OPTIMIZING WORK STEALING ALGORITHMS WITH SCHEDULING CONSTRAINTS

BY

JONATHAN JOSIAH LIFFLANDER

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Doctoral Committee:

Professor Laxmikant V. Kalé, Chair
Dr. Sriram Krishnamoorthy, PNNL
Professor David Padua
Professor Vivek Sarkar, Rice University
Professor Marc Snir

Abstract

The fork–join paradigm of concurrent expression has gained popularity in conjunction with work-stealing schedulers. Random work-stealing schedulers have been shown to effectively perform dynamic load balancing, yielding provably-efficient schedules and space bounds on shared-memory architectures with uniform memory models. However, the advent of hierarchical, non-uniform multicore systems and large-scale distributed-memory architectures has reduced the efficacy of these scheduling policies. Furthermore, random work stealing schedulers do not exploit persistence within iterative, scientific applications.

In this thesis, we prove several properties of work-stealing schedulers that enable online tracing of the tasks with very low overhead. We then describe new scheduling policies that use online schedule introspection to understand scheduler placement and thus improve the performance on NUMA and distributed-memory architectures. Finally, by incorporating an inclusive data effect system into fork–join programs with schedule placement knowledge, we show how we can transform a fork–join program to significantly improve locality.

To the spirits that have guided me.

Acknowledgments

Prima facie, I would like to express my gratitude to my advisor Professor Sanjay Kale for his continuous support of my PhD and related research efforts. He has guided me intellectually with patience, motivation, and immense knowledge. His confidence, perseverance, and unwavering passion for parallel computing has enabled my research career, and led me through a journey of scholarly pursuit.

Throughout my PhD, Dr. Sriram Krishnamoorthy has served as a mentor, providing intellectual support and motivation, teaching me the art of high-impact research. He has co-authored numerous papers with me, working side-by-side to enable much of the research contained within this thesis.

In the Parallel Programming Laboratory, I have been strongly supported both professionally in my research and through friendship from my colleagues: notably, Nikhil Jain, Harshitha Menon, and Phil Miller. These three colleagues have greatly impacted my journey of professional and personal development.

My entire family has fully supported me through the process of obtaining my PhD in numerous ways. I would like to express my deep gratitude to my wife, Stephanie Ai-Ping Lee, for her unwavering support throughout my PhD. She has provided invaluable emotional support, empowering me to work countless hours toward this monumental scholarly pursuit. My brother, Daniel Lifflander, has provided a strong foundation for my research. As a fellow computer scientist, he understands the process of writing code, and more importantly the value of developing theoretical principles that last. His friendship, support, and open ear to my ever-evolving ideologies have been invaluable during my PhD. I am greatly indebted to my father, John Lif-

flander, my mother, Carol Lifflander, and my sister, Anna-Grace Lifflander, for their enduring support in this endeavor and guidance. My father-in-law, John Lee, and mother-in-law, Helen Lee, have both provided significant encouragement and confidence in me throughout this entire process. My brother-in-law, Daniel Lee, has been an incredible friend to me, always supporting me in any way possible. My sister-in-law, Christina Lee, has always shown confidence in my research abilities and supported me.

I would like to thank my best friend, Zacharia Persson, for his continuous support of my PhD. His strong confidence in my research and assistance in refining ideas has contributed significantly to my intellectual wellbeing and provided a foundation for my research.

Finally, I would like to thank all the PPLers for which I have interacted and have played a significant role in my journey: Esteban Meneses, Osman Sarood, Ehsan Totoni, Anshu Arya, Yanhua Sun, Eric Bohm, Issac Dooley, Aaron Becker, David Kunzman, Filippo Gioachin, Abhinav Bhatele, Gengbin Zheng, Pritish Jetley, Akhil Langer, Xiang Ni, Michael Robson, Ronak Buch, Bilge Acun, and Eric Mikida.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

The demand for increased computational capacity has corresponded with increased hardware complexity in commodity and high-end computing systems. Processor frequency limitations have engendered the multi-processor era, leading to hierarchical, non-uniform memory access (NUMA) subsystems, and a host of intricate architecture designs that ostensibly enable large-scale science to be performed efficiently. However, it is challenging for domain scientists to maintain complex codes, co-evolving them with new computational abstractions while ensuring the correctness of their algorithms. Strategic oversight from stake-holders may inhibit momentum toward new concurrent programming models, as the staying power of new models may be unpredictable, especially in research contexts.

Two decades of extensive research on fork–join (or, *task-parallel*) programming idioms have demonstrated their utility in expressing concurrency, while maintaining performance-portability and theoretical optimality under specific conditions. The fork–join idiom has proliferated both commercially on multi-core systems and, more experimentally, on large-scale supercomputers. Many runtimes and languages now support fork–join, including OpenMP 3.0 [2], Cilk [3], X10 [4], Habanero Java [5], NESL [6], Intel Thread Building Blocks [7], Java Concurrency Utilities [8], etc.

The Cilk language—an exemplar fork–join model—augments the C language with two linguistic constructs: spawn and sync. With the spawn construct, a procedure or function can be described as concurrent with the following continuation in the enclosing function scope. The sync keyword expresses the limits the concurrency within that scope, ensuring that all recur-

sively spawned functions are completed. With these two keywords, fork–join concurrency can be expressed concisely.

Extensive theoretical work on random work-stealing schedulers in a uniform multi-core setting has proven that work-stealing, in conjunction with strict forkâĂŞjoin models, produces provably-optimal schedules with understandable space and time bounds. Moreover, work stealing has been shown to be effective for dynamic load balancing. Compared to other load balancers that are activated periodically, and thus cannot address immediate imbalances that may arise, random work stealing can balance loads within a phase of the application.

Overdecomposition—exposing more work units than the available hardware threads—allows such models to diminish load imbalance, even in the face of unpredictable, dynamically varying loads. Instead of decomposing the application directly to the available parallelism, the application is written to expose more concurrency, enabling work units to be shuffled by the runtime system as the application unfolds. In the models studied, applications will typically expose work units of a medium grain size: larger than a few FLOPs, but orders of magnitude more concurrency than the available hardware threads.

This thesis addresses several challenges that arise in scheduling in the context of random work stealing:

- Random work stealing produces efficient schedules at runtime depending on load. However, these schedules are different every time the application is executed, making them difficult to understand and analyze. The sheer number of work units exposed inhibits naïve methods of capturing and storing the schedule. Effectively tracing the application, or capturing where and when each work unit is executed, can enable many optimizations and analyses both online and offline. Can a random work-stealing schedule be captured without explicitly storing where each work unit is executed?

- On shared-memory NUMA architectures, random work-stealing can inflate execution time because as work units are stolen between processor cores, the memory domain that work units access may be different from where the data is mapped. Different memory polices, such as first-touch or interleaved, change the nature of the initial data mapping further

complicating the problem. Can random work stealing schedulers be adapted to be aware of NUMA locality without explicitly mapping work units?

- Traditionally, random work stealing has been confined to shared-memory architectures. Many challenges inhibit adapting work-stealing algorithms to distributed-memory contexts. Random work stealing typically starts with a single task on one processor and spreads the work throughout the machine by steals, which may limit an application's scalability with high ramp-up costs. Random steals may also interrupt execution when implemented with message passing. Additionally, migrating work units may be prohibitively expensive. Can random work stealing be applied to distributed-memory contexts in a way that is comparable to/competitive with persistence-based load balancers? Can random work stealing be applied to distributed-memory contexts and how does it compare with persistence-based load balancers?

- The fork–join model limits the expression of concurrency to series-parallel DAGs (directed acyclic graphs). Many of the advantages (i.e. efficiency proofs) stem from this limitation, but it also inhibits certain locality-based optimizations. A wealth of compiler research has studied automatic locality optimizations for for-loop codes (e.g. Pluto), but this research not applicable to recursive programs written in fork–join models. Can the performance advantages gained through state-of-the-art compile-time locality optimization be achieved through runtime reordering of task trees in fork–join models?

This thesis tackles the preceding challenges in the context of random work stealing and addresses the associated questions. In Chapter 4, we prove several properties of help-first and work-first scheduling polices, presenting tracing algorithms that capture work-stealing schedules with very low overhead. In Chapter 5, we devise retentive work stealing, a new scheduling policy that scales to large distributed-memory machines, and compare it to persistence-based load balancers. In Chapter 6, we present constrained work stealing algorithms that improve the performance of random work stealing in NUMA contexts. Finally, in Chapter 7, we present dynamic splicing: runtime algorithms that recursively interleave distinct phases of an application to accrue significant cache locality benefits.

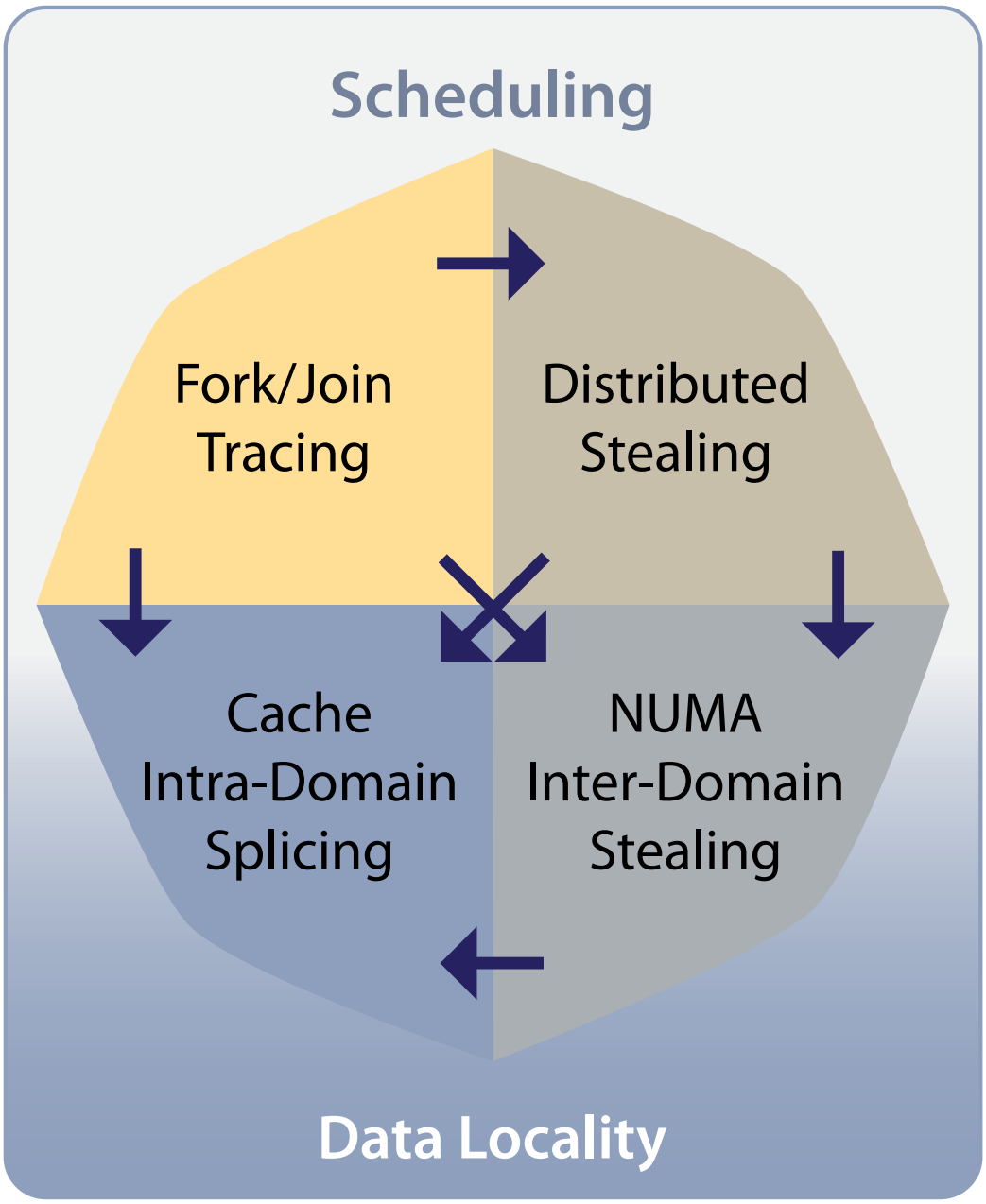


Figure 1.1: Topics covered in thesis.

CHAPTER 2

Background and Notation

For the work presented in this thesis, we shall make several assumptions about the nature of fork–join expression of concurrency and the underlying work-stealing schedulers that map them to hardware resources.

2.1 Spawn-Sync and Async-Finish Concurrency

An async/spawn statement identifies the associated statement as a task (formally defined in Section 2.2), the basis unit of concurrent execution. A task identified by the async/spawn statement can be executed in parallel with the enclosing task, referred to as the parent task. A finish/sync statement identifies the bounds of concurrency. All computation enclosed by a finish statement, including nested concurrent tasks, is 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 sync 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 complete.

2.2 Notation

We define a *continuation step* (or simply *step*) to be the dynamic sequence of instructions with no interleaving async/spawn, finish/sync, 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/spawn or finish statement, representing all encapsulated continuation steps. In shared-memory environments, all continuations of a task occupy the same storage and transform the state from one continuation 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/spawn 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.

2.3 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.

Work stealing schedulers differ in the task creation strategy, victim selection policy, and action taken when a concurrent task is encountered. We will consider random work stealing, where the victim is chosen at random, in the ensuing discussion, though the presented algorithms can support other victim selection policies as well. Lazy task creation postpones the creation of task descriptors to reduce overhead, but does not change the fundamental scheduling strategy.

We study and build upon two scheduling policies that are outlined in Figure 2.1 for async-finish programs identified by Guo et al. [9]. In the *work-first* scheduling 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 or sync statement, the current task's continuation is pushed onto the deque to complete processing of the tasks nested within the finish scope. Finish scopes or sync statements 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].

```
@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);
```

(a) 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 management

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

(b) Work-first scheduler

Figure 2.1: Basic actions in the two scheduling policies

CHAPTER 3

Related Work

3.1 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 [10, 11] and for optimizing transactional memory conflict detection [12]. 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 [1] does not exploit the steal relationship—a beneficial approach that we demonstrate later in this chapter.

Work stealing has typically been studied empirically [13]). Tallent and Mellor-Crummey [14] presented blame shifting to relate lack of parallel slack in work stealing to code segments. They assume global information about the number of active and idle workers.

3.2 Load Balancing

Load imbalance is a well-known problem and has been widely studied in the literature. Applications involving regular data structures, such as dense

matrices, achieve load balance by choosing a data distribution (e.g., multipartitioning [15]) and carefully orchestrating communication. These approaches are specialized for regular computations and do not directly extend to other classes.

Iterative calculations on less regular structures (e.g., sparse matrices [16] or meshes [17]) employ an inspector-executor approach to load balancing where the data and associated computation balance is analyzed at runtime before the start of the first iteration to rebalance the work (e.g., CHAOS [18]) Typical approaches to such start-time load balancing employ a partitioning scheme [19]. Scalable parallelization of such partitioners [20, 21] is non-trivial.

Overdecomposition is instantiated in a variety of forms by different compilers and runtimes, including Cilk [22], Intel Thread Building Blocks [7], OpenMP [23], Charm++ [24], Concurrent Collections [25], and ParalleX [26]. Many application frameworks that understand domain-specific data structures implicitly employ this approach [27].

Charm++ [24] supports a variety of persistence-based load balancing algorithms. The typical approach involves gathering statistics for objects, measuring the amount of imbalance, and executing the corresponding rebalancing algorithms in either a centralized or hierarchical [28] fashion. Such hierarchical persistence-based load balancers are employed in several scalable applications [29]. Unlike the hierarchical schemes considered by Zheng et al. [28], we focus on the development of a localized rebalancing algorithm that also incurs lower space overheads due to greedy rebalancing.

Irregular algorithms whose workload cannot be predicted at start-time, or work partitioned into sub-units, pose a significant challenge to the above load balancing approaches. Applications in this class include state-space search, combinatorial optimization, and recursive parallel codes. Work stealing is a popular approach to load balancing such applications. Cilk [22] is a widely-studied depth-first scheduling algorithm for fully-strict computations with optimal space and time bounds. It has been shown to scale well on shared memory machines and implementations are available for networks of workstations [30] and wide-area networks [31]. Prior work on extending work stealing to distributed-memory adapted the algorithm to employ remote memory access (RMA) operations using ARMCI [32], demonstrating scaling to 8192 cores [33]. An implementation in X10 extended this algorithm

to reduce the interference caused by steal operations [34]. These approaches employed work stealing in a memoryless fashion. We present work stealing for distributed-memory machines using a threaded active message library developed on MPI, demonstrating scaling to significantly higher core counts.

Work stealing has not been evaluated at this scale (100,000+ cores on multiple platforms) for any application on any hardware platform using any prior algorithm. The largest prior demonstration was on up to 8192 cores [33]. The domains employing work stealing and persistence-based load balancing have traditionally been disjoint. We are not aware of any prior work comparing the effectiveness of these two schemes for iterative applications, or any other application domain.

3.3 NUMA-Aware Scheduling

Cilk [35] employs random work stealing with a work-first execution strategy. Guo et al. [36] studied help-first scheduling policies to improve the load balance achieved in practice. Hierarchical place trees [37] and related approaches [38, 39] adapt the work stealing to promote localized steals, indirectly improving data locality. These schemes preferentially access local data but can result in different remote accesses across phases. Parallel depth-first scheduling [40] improves locality of access to shared caches in nested-parallel computations, rather than across sequentially composed nested-parallel computations. We consider the complementary problem of locality optimization across phases.

Locality-aware scheduling is supported in X10 [41] through explicit invocation of task execution at the location of specific data elements. This approach imposes the burden of data distribution and load balance on the programmer. The property we exploit in incremental optimization of data placement is referred to as the *principle of persistence*—the same computation structure is repeated and can be optimized for. Charm++ [42] explicitly associates computation with data objects and performs persistence-based load balancing [43] that is coupled with data migration. Our approach does not impose such tight binding of computation and the data it operates upon.

Nikolopolous et al. [44] studied reusing loop schedules to improve memory affinity for OpenMP looping constructs. Olivier et al. [45] observed that non-

locality memory accesses lead to an inflation in an OpenMP task's execution time. They present API support that explicitly specifies locality domains and placement of tasks on them. Explicit data placement and layout specifications [46, 47, 48] and modifications to the random work stealing policy [49] also have been considered for OpenMP task programs.

The Pochoir compiler [50] performs scheduling across iterations for stencil computations. The transformed code is generated in Cilk and scheduled as a Cilk program. While time tiling improves data reuse, the Pochoir compiler does not specifically optimize for data locality. Our approach to constraining work stealing computations with data locality can be used to improve the scheduling of computations generated by a compiler such as Pochoir.

Grain size control has been studied in many different contexts. Static compile-time approaches [51] have been employed. Charm++ has adaptive grain size control for state space search [52]. Other work has focused on only splitting grain sizes when a worker is in need of work, which is feasible with a managed runtime [53, 54, 55]. Lazy task creation has been used to increase granularity [56], while other work has focused on increasing the steal granularity [57, 58], which is different than our approach.

Our approach is applied to the Cilk work-first scheduling runtime and can be adapted to other fork/join models. It can be directly applied to other work-stealing models, such as a help-first scheduler. However, for more divergent models, efficient tracing and constrained execution algorithms will be required to effectively implement our methodology.

3.4 Cache Locality Optimization and Effect Systems

3.4.1 Compile-time Analysis and Transformation for Fork/join Programs

Burstall and Darlington presented a system of rules for transforming recursive programs [59]. Nandivada et al. [60] presented a transformation framework to optimize exposed concurrency in task-parallel programs. Neither takes data locality into account. The work by Rugina et al. [61, 62, 63] on static analysis of pointer and array index references in recursive programs can aid the construction of the effects used in this paper.

3.4.2 DSLs and Library Composition

Domain-specific languages (DSLs) allow information related to data locality and dependences to be expressed at a higher-level of abstraction enabling aggressive optimizations [64]. Pochoir [50] is an example stencil DSL that exploits language-level information to generate optimized stencil programs in Cilk. Chandra et al. [65] present the Cool language, an extension of C++ for task-parallel programs, that allows specification of affinity and migration hints to the runtime scheduler, effectively an approach to user-controlled NUMA-like locality. Active libraries exploit domain information and associated optimizations without the need for distinct language syntax [66]. Just as in DSLs, active libraries result in the generation of optimized code based on transformation of the source code being analyzed. Use of library annotations to optimize across libraries without requiring their source code has been extensively studied [67, 68, 69]. Unlike our runtime approach, these approaches typically exploit the library annotations to generate optimized implementations at compile-time.

3.4.3 Runtime Locality-aware Scheduling of Fork/join Programs

Our scheduling strategy—explore the continuation of an invoked function for splicing—is similar to the help-first scheduling strategy presented by Guo et al. [36]. Guo et al. focus on exposing additional parallelism for load balancing rather than optimizing data locality. Parallel depth-first schedulers are designed to enable constructive cache sharing among concurrent tasks in a fork-join programs [70]. Guo et al. presented SLAW—scalable locality-aware work stealing—to schedule tasks to places based on locality hints [71]. These runtime approaches optimize locality within a phase, represented by a single recursive function or task.

3.4.4 Scheduling Based on Effect Information

TWEJava [72] exploits effect information to extract deterministic parallelism. Legion [73, 74] and HJp [75] exploit information on memory accesses by tasks to ensure deterministic parallelism. These approaches focus on au-

tomatic extraction of parallelism and determinism guarantees, but do not tackle data locality. Specifically, Legion [74] does not attempt to refine the dependences to the finest possible granularity, which is required for cache locality optimization, to minimize runtime costs. Pen and Pai [76] employ a hardware-software approach to managing last level cache for input annotated task-parallel programs using the OmpSs task-parallel programming model. Jo and Kulkarni [77] exploit access information to schedule concurrent threads operating on shared data to optimize inter-thread data locality for tree traversals. Philbin et al. [78] is most related to our work. They extract sub-computations in a sequential program into parallel threads, which are then executed in an order that minimizes cache misses. However, their strategy does not tackle dependences, does not consider interleaved execution of the threads to further reduce cache misses, cannot handle recursive programs, and does not inter-operate with a dynamic parallel scheduler such as Cilk.

3.4.5 Thread/task Management

The splicing optimization can be implemented using various user-level thread libraries that support lightweight context switching (e.g., scott [79], qthreads [80], boost [81]). Our management of dependences involving delayed steps is similar to various task graph schedulers (e.g., Nabbit [82], CnC [83], Supermatrix [84], and X10 [85]).

CHAPTER 4

Tracing Work Stealing Schedulers

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.

In this chapter, 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 [86]. 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 chapter, 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 [1] and supporting retentive stealing without requiring explicit enumeration of tasks for distributed-memory, described in Section 5.7.

The following are the primary contributions of this chapter:

- 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; and
- reduction in the cost of data race detection using an algorithm that maintains and traverses the dynamic program structure tree [1].

4.1 Problem Statement

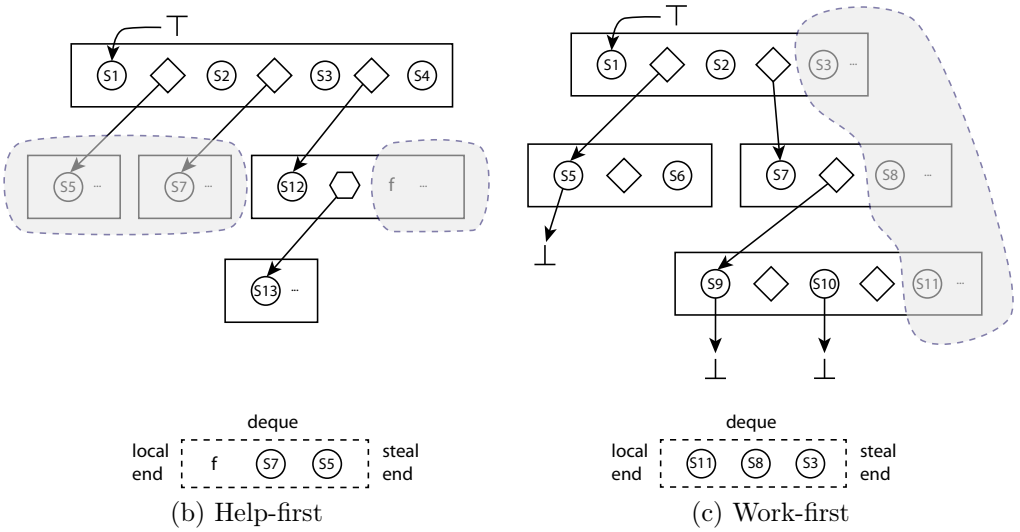
An example async-finish program is shown in Figure 4.1a. In the figure, `s1`, `s2`, etc. are continuation steps. Tasks spawned using `async` statements need to be processed before the execution can proceed past the immediately enclosing `finish` statement. `async` statements that are not ordered by a `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.

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.

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

```
fn() {
  s1;
  async { s5; async x; s6; }
  s2;
  async {
    s7;
    async { s9; async x; s10;
              async x; s11; .. }
    s8; ..
  }
  s3;
  async { s12; finish {s13;..} ..}
  s4;
}
```

(a) async-finish program



(b) Help-first (c) Work-first

Figure 4.1: 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.

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 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 4.1. 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 2.1. 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.

4.2 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 4.1b shows a snapshot of the help-first scheduling of the program in Figure 4.1a. The steps in the outermost task, represented by `fn()`—`s1`, `s2`, `s3`, and `s4`—are processed before any other steps.

Observation 4.2.1. *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 4.2.2. *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 4.2.3. *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 4.2.4. *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 4.2.2, 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 4.2.3, this task's partial continuation will not be stolen until all the parent's older siblings are stolen. By lemma 4.2.2 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 4.2.5. *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 4.2.2, t_2 is processed before t_1. By lemma 4.2.3, 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 4.2.2, task t must be an older sibling of the parent task q. From lemma 4.2.3, 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 4.2.2, q is processed before t. q has not been processed, hence we have a contradiction. □

Lemma 4.2.6. *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 4.2.5, q is last task processed on level l. By lemma 4.2.2, 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 4.2.7. *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 4.2.4, 4.2.5, and 4.2.6, 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 4.2.8. *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 4.2.7) (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 4.2.4) 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 4.1b. 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.

In Figure 4.2, we present another example program, along with how the program is unfolded in snapshots as the scheduler run. Then, based on steals occurring from the steal end of the deque, we show how the steal tree is built.

4.3 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

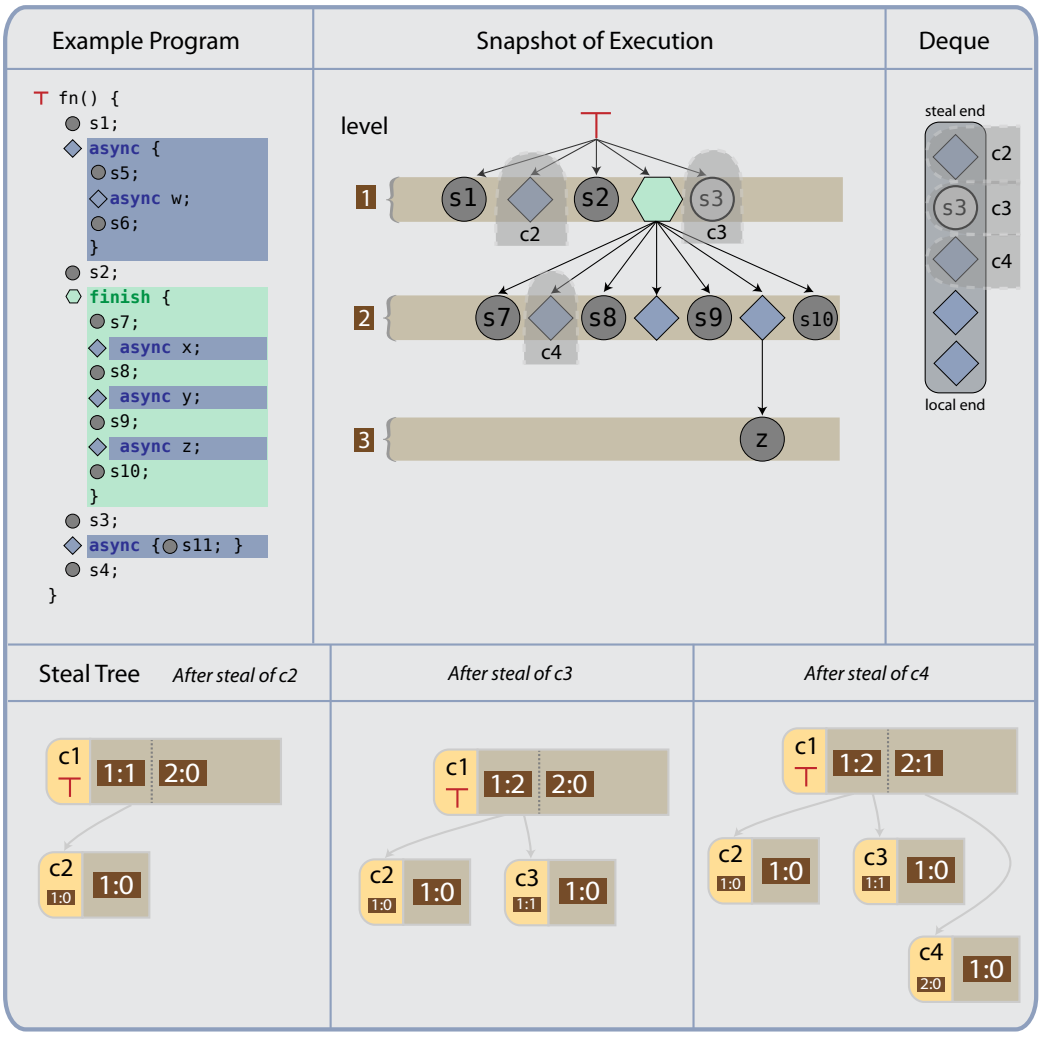


Figure 4.2: Illustration of Steal Tree being formed as steals (c1,c2,c3) occur.

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 4.3.1. *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 4.3.2. *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 4.3.1 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 4.3.3. *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 4.3.2, 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 4.3.2), 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 4.1c. The thread began execution from step `S1` and has completed execution of `S`1, `S`2, `S`5, `S`6, `S`7, `S`9, and `S`10. It is currently executing the task spawned by `S`10 with the the deque consisting of steps `S`3, `S`8, and `S`11—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 `S`3, `S`8, followed by `S`11. Note that the execution in the work-first scheduling policy is *left-to-right*, while the steals are *right-to-left*.

4.4 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 4.2 and 4.3. 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 4.3.

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

```
struct ContinuationHdr {
  int level; //this task's async level
  int step; //this continuation's step
};
struct Task : ContinuationHdr { ... };
struct Cont : ContinuationHdr { ... };
struct WorkingPhaseInfo {
  // victim stolen from
  int victim;
  //step stolen at each level, init to −1
  vector<int> stepStolen;
  // list of thieves
  vector<int> 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
@startWorkingPhase(Cont c):
  // level of starting frame
  c.level = 0;

//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 4.3: Common data structures and level management for all algorithms

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.

4.4.1 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.

4.4.2 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 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)
```

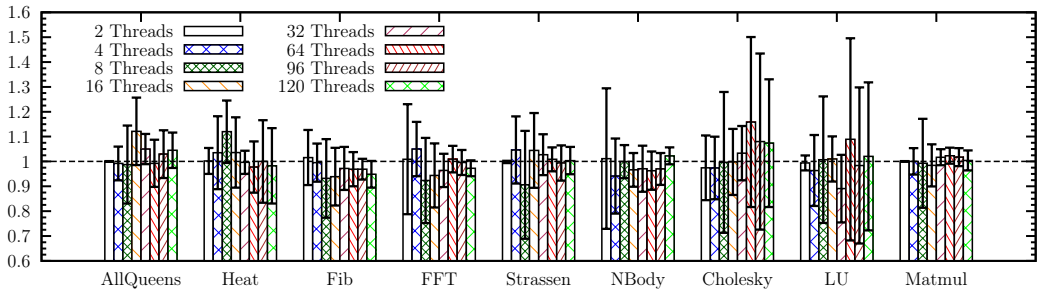


Figure 4.4: The ratio of mean execution time with tracing versus without tracing with a sample size of 15 on the POWER 7 architecture using the shared-memory Cilk runtime. The error bars represent the error in the difference of means at 99% confidence, using a Student's t-test.

4.4.3 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 4.5 and 4.7 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.

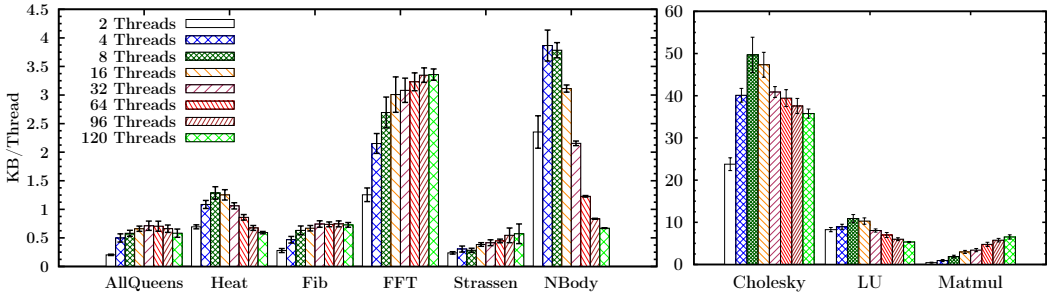


Figure 4.5: The storage overhead in KB/thread with our tracing scheme using the shared-memory Cilk runtime on the POWER 7 architecture. The error bars represent the standard deviation of storage size with a sample size of 15.

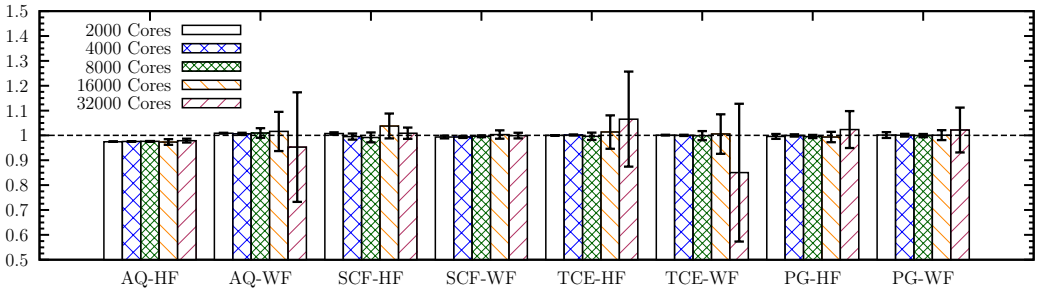


Figure 4.6: The ratio of mean execution time with tracing versus without tracing with a sample size of 15 on Cray XK6 Titan in distributed-memory. The error bars represent the error in the difference of means at 99% confidence, using a Student's t-test.

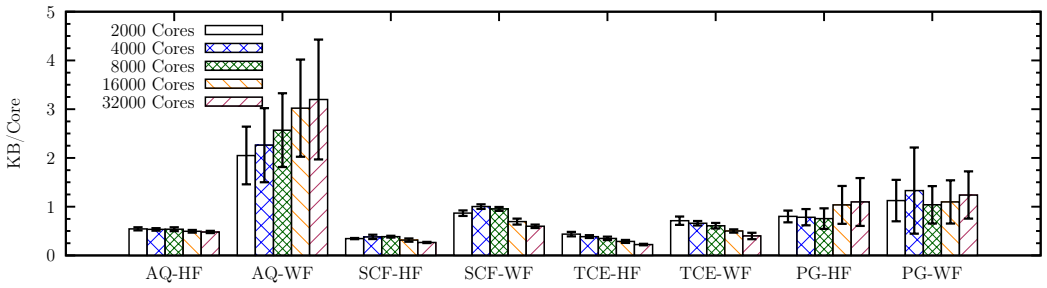


Figure 4.7: The storage overhead in KB/core with our tracing scheme for distributed-memory on Cray XK6 Titan. The error bars represent the standard deviation of storage size with a sample size of 15.

4.5 Applications

4.5.1 Data-race Detection for Async-Finish Programs

Raman et al. [1] 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. [1] 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 subtree 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 [1] 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
```

4.6 Experimental Evaluation

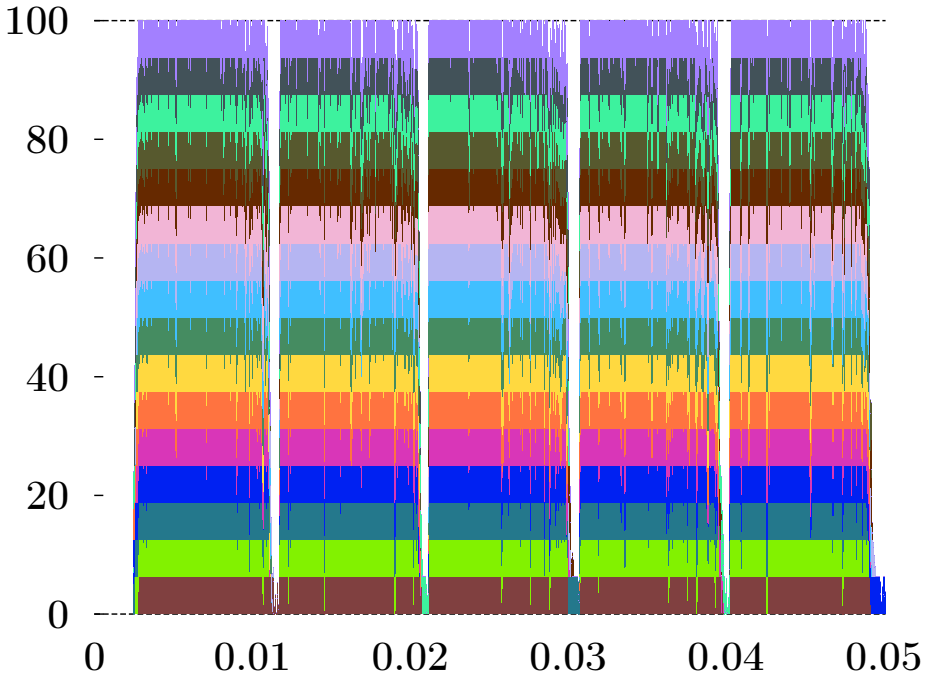
The shared-memory experiments were performed on a POWER 7 system composed of 128 GB of memory and a quad-chip module with eight cores per chip running at 3.8 GHz and supporting four-way simultaneous multithreading. The distributed-memory experiments were performed on the OLCF Cray XK6 'Titan', an 18688-node 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. The distributed-memory experiments use the implementation discussed in detail in Section 5.4.

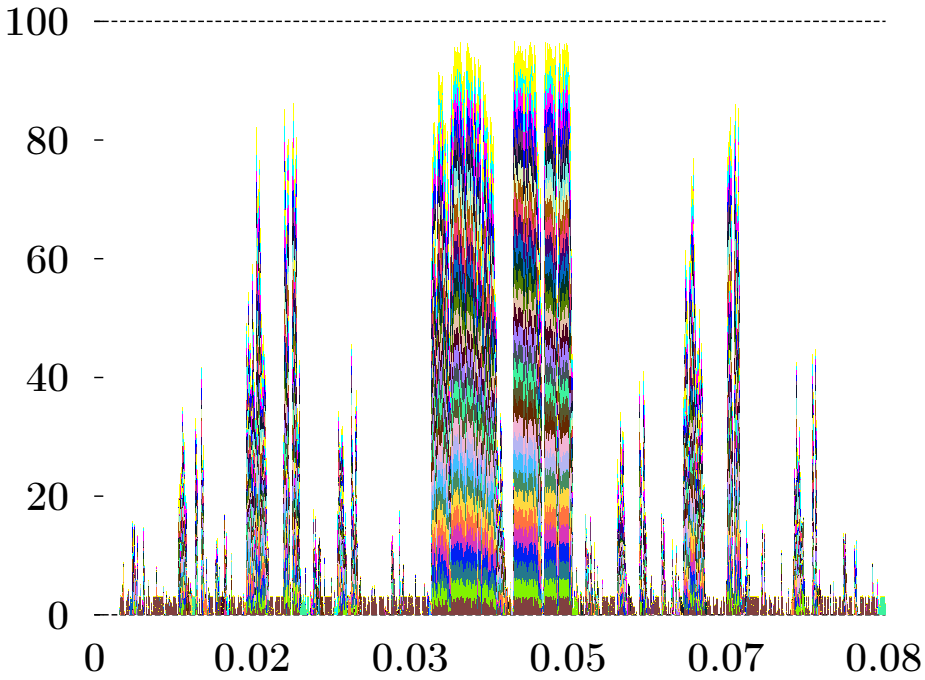
4.6.1 Tracing on Shared-Memory Systems

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 4.2. 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 [87]; 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 4.4. 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 [88]. 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



(a) 16 threads of the iterative Heat benchmark for 5 iterations



(b) 32 threads of the LU benchmark

Figure 4.8: Percent utilization (y-axis) over time in seconds (x-axis) using the steal tree traces, colored by a random color for each thread.

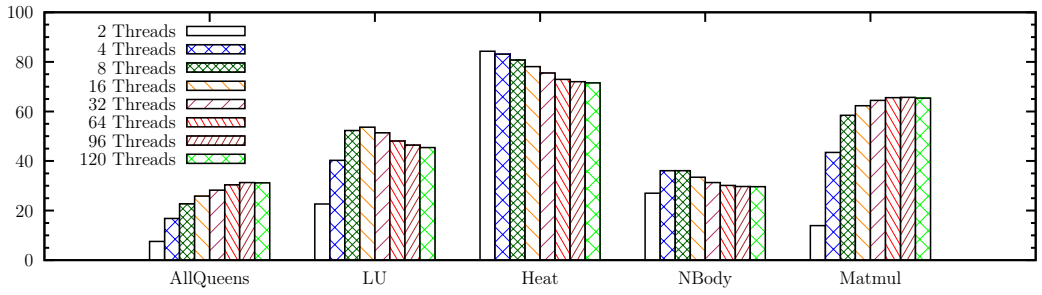


Figure 4.9: The percent reduction attained by exploiting our traces to reduce the tree traversals of the DPST (dynamic program structure tree) to detect races in a shared-memory application [1].

high variability between runs.

Figure 4.5 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 4.8. 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.

4.6.2 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 4.2). 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 described in detail in Section 5.5.1. PG is a work stealing implementation of the pair-wise sequence alignment application designed by Wu et al. [89]. We refer the readers to the corresponding paper for details.

We execute the four benchmarks under both work-first and help-first scheduling policies. Figure 4.6 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 4.7 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 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.

4.6.3 Space Comparison Study using the StealTree

To study the benefits of tracing with the proposed framework using the steal tree, we have selected five shared-memory benchmarks in Cilk for an in-depth analysis of the space overheads. For each configuration experimentally evaluated, we have selected two problem sizes for each benchmark and three

Benchmark	Configuration	Sequential Cutoff
AllQueens	nq = {14,15}	b = {6,3,1}
Heat	nt = 5, n = $\{16k^2,32k^2\}$	cols = {8,4,1}
Fib	n = {43,35}	b = {10,5,1}
Matmul	n = $\{3000^2,4000^2\}$	b = {64,16,8}
Cholesky	(n,zeros) = $\{(2k^2,20000),(1k^2,10000\}$	b = {8,4,2}

Table 4.1: Benchmarks for space comparison (Figure 4.10).

sequential cutoffs, shown in Table 4.1. We limit the depth of recursive expansion to the sequential cutoff for each benchmark and invoke a sequential kernel. Hence, as the sequential cutoff increases less concurrent tasks will be created, reducing the space requirements.

For the AllQueens and Fib benchmark, the sequential cutoff is invoked by sequentially executing a kernel when that cutoff is reached while recursively subdividing the problem. For Matmul and Cholesky, the sequential cutoff is the matrix block size that is used after recursively subdividing the matrix. For Heat, which uses a 1-D column decomposition, the sequential cutoff is the number of columns that are executed sequentially after recursive subdivision.

In Figure 4.10, we show a space comparison with and without the steal tree. Each bar is the mean storage of five runs and the error bars represent standard deviation. The 'Enum' bars represent the amount of storage (in kB) that is required to explicitly trace the tasks by storing the processor on which each task executed. The 'StealTree' bars show the amount of space required to store the entire steal tree, using the formulas in Section 4.4.3. The graphs show a consistent pattern for the benchmarks tested: explicit enumeration of tasks requires orders of magnitude more space than using the steal tree to obtain an equivalent trace of the computation. The steal tree results are not independent of the number of processor cores, because the trace becomes more fragmented as more steals occur to keep all the workers busy. The total space requirements approximately increase linearly with the number of cores. Hence, we can predict that even on much larger shared-memory machine, the storage costs will most likely be significantly lower than explicit enumeration.

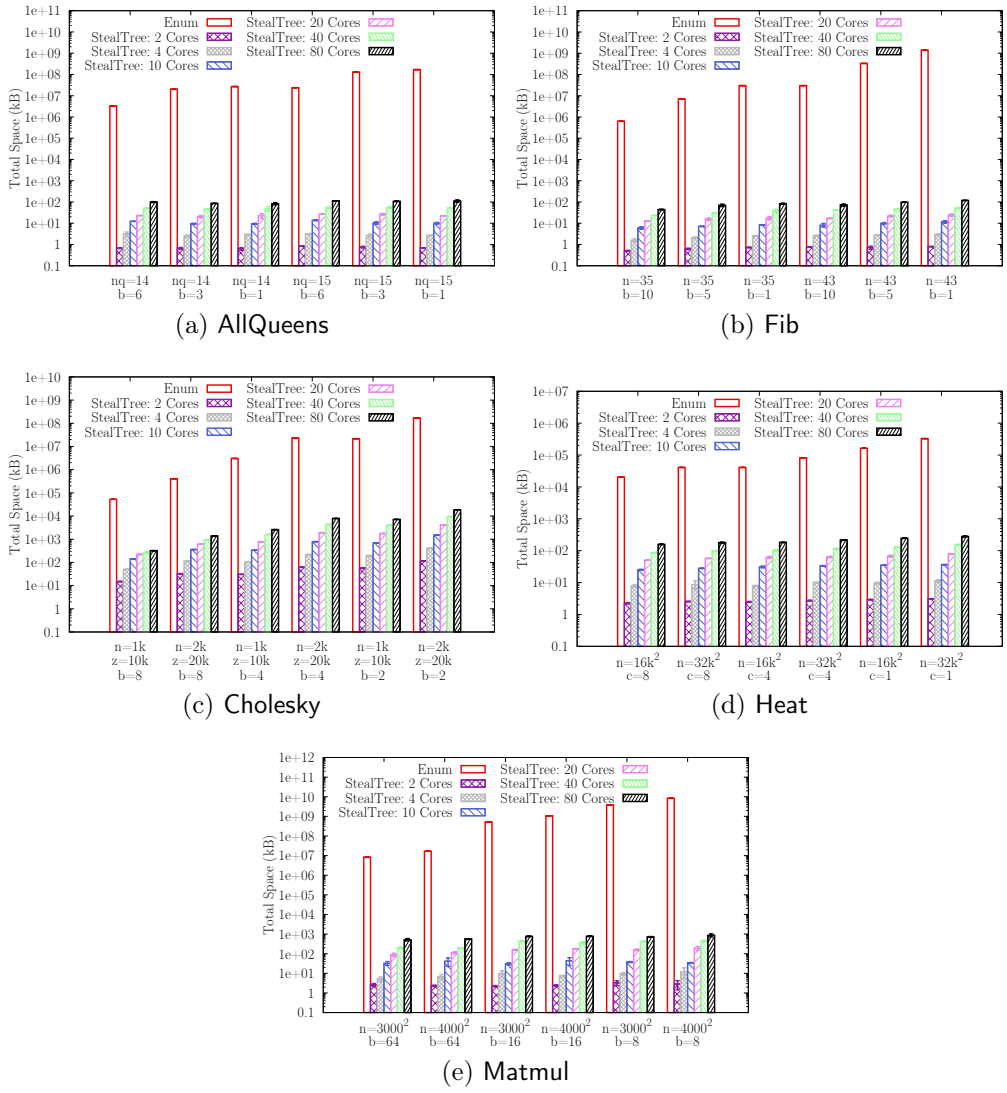


Figure 4.10: Space comparison between naïvely tracing tasks using explicit enumeration (Enum) and using the proposed tracing framework (the steal tree). Using the steal tree to trace each application requires orders of magnitude less storage than naïve tracing.

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 4.2: Benchmark configurations.

4.6.4 Optimizing Data-Race Detection

For several benchmarks in Cilk, we show in Figure 4.9 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.

CHAPTER 5

Retentive Work Stealing

Applications often involve iterative execution of identical or slowly evolving calculations. Many such applications exhibit significant complexity and runtime variation to preclude effective static load balancing, requiring incremental rebalancing to improve load balance over successive iterations.

A popular approach to addressing the load balancing challenge is *overdecomposition*. Rather than parallelize to a specific processor core count, the application-writer exposes parallelism by decomposing work into medium-grained tasks. Each task is coarse enough to enable efficient execution (tiling, vectorization, etc.) and potential migration, while being fine enough to expose significantly higher application-level parallelism than is required by the hardware. This allows the middleware to manage the mapping of the tasks to processor cores, and rebalance them as needed.

This chapter focuses on balancing the computational load across processor cores for overdecomposed applications, where static or start-time approaches are insufficient in achieving effective load balance. Such applications require periodic rebalancing of the load to ensure continued efficiency.

Applications that retain the computation balance over iterations, with gradual change, are said to adhere to the *principle of persistence*. Persistence-based load balancers redistribute the work to be performed in a given iteration based on measured performance profiles from previous iterations. We present a hierarchical persistence-based load balancing algorithm that attempts to localize the rebalance operations and migration of tasks. The algorithm greedily rebalances "excess" load rather than attempting a globally optimal partition, which could potentially incur high space overheads.

Work stealing is an attractive alternative for applications with significant load imbalance within a phase, or applications with workloads that cannot be easily profiled. Work stealing ameliorates these problems by actively attempting to find work until termination of the phase is detected. This approach has been successfully employed in domains where the load imbalance cannot be computed a priori or varies significantly across consecutive invocations [90]. We present an active-message-based work stealing algorithm optimized for iterative applications on distributed memory machines. The algorithm minimizes the number of remote roundtrip latencies incurred, reduces the duration of locked operations, and takes into account data transfer time when stealing tasks across the communication network. *Retentive* work stealing augments this algorithm with knowledge of execution profiles from the previous iteration to enable incremental rebalancing.

The scalability and overheads incurred by these algorithms are evaluated using candidate benchmarks. We observe that the persistence-based load balancer produces effective load distributions with low overheads. We demonstrate work stealing at over an order of magnitude higher scale than prior published results. While more scalable than widely believed, work stealing does not perform as well as persistence-based load balancing. Retentive stealing, which borrows from the persistence-based load balancer to retain information from the previous iteration, is shown to adapt better to iterative applications, achieving higher efficiencies and lower stealing overheads.

The primary contributions of this chapter are:

- a hierarchical persistence-based load balancing algorithm that performs greedy localized rebalancing;
- an active-message-based retentive work stealing algorithm optimized for distributed memory machines;
- first comparative evaluation of persistence-based load balancing and work stealing;
- most scalable demonstration of work stealing — on up to 163,840 cores;
- demonstration of the benefits of retentive stealing in incrementally rebalancing iterative applications; and

- application of tracing using the steal tree presented in Chapter 4 to reduce the cost of retentive work stealing.

5.1 Challenges

An effective load balancer should achieve good load balance at scale while incurring low overheads. In particular, the cost of rebalancing should be related to the degree of imbalance incurred and not the total work. In an iterative application, the load balancing overhead should decrease as the calculation evolves towards a stable state, increasing only when the application induces additional load imbalance.

Persistence-based load balancers cannot adapt to immediate load imbalance and incur periodic rebalancing overheads. If a processor runs out of work too early in a phase, it needs to wait until the end of the phase for the imbalance to be identified and corrected. Minimizing task migration for such load balancers can be beneficial by retaining data locality and topology-awareness that guided the initial distribution.

Work stealing algorithms typically employ random stealing to quickly propagate available work. This interferes with locality optimizations and topology-aware distributions. Termination detection is a challenge at scale on distributed-memory machines. While hierarchical termination detectors approximate the cost of tree-based reduce operations in principle, they incur higher costs in practice. Termination detectors run concurrent with the application, introducing additional overheads throughout the application's execution.

The cost of stealing itself is significant on a distributed-memory machine due to the associated communication latency and time to migrate the stolen work. Work stealing also ignores prior rebalancing, incurring repeated stealing costs across iterations. Due to these reasons, work stealing has traditionally been confined to shared-memory systems. In addition, given all the "noise" introduced by work stealing, it is typically employed in applications that incur significant load imbalance and are not amenable to an initial distribution. Typical approaches to distributed-memory load balancing consider hierarchical schemes due to the perceived limitations associated with work stealing.

5.2 Programming Model

The algorithms presented in this chapter are implemented in the context of a MPI-based runtime library. The runtime acts both as a user-level library for distributed memory task-based execution and as the runtime target for language constructs such as X10 async [91] that support non-SPMD execution modes. All processes in a group or an MPI communicator collectively switch between SPMD and task processing phases. The code snippet below illustrates the repeated execution of a task processing phase as part of an iterative application. Throughout the chapter, we employ C++ style notation except for a few shorthands in place of detailed implementation-specific API.

```
TslFuncRegTbl *frt = new TslFuncRegTbl();
TslFunc tf = frt->add(taskFunc);
TaskCollProps props;
props.functions(tf,frt)
      .taskSize(sizeof(Task))
      .localData(&procLocalObj,sizeof(procLocalObj))
      .maxTasks(localQueueSize);
UniformTaskCollSplit utc(props); //collective
for (..) {
  Task task(..); //setup task
  utc.addTask(&task, sizeof(Task)); //local operation
}
while(..) {
  utc.process(); //collective
  utc.restore(); //implicit collective
}
```

The fundamental construct in the computation is a task collection seeded with one or more tasks. The library supports several task collection variants, each specialized to exploit specific properties of the task collection known at runtime. The above example shows a task collection, called UniformTaskCollectionSplit, which optimizes for a collection of tasks of identical size, with a known upper bound on the number of tasks on any individual processor core, and additional information common across all tasks (provided as the opaque struct procLocalObj). Function pointers associated with a task execution are translated into portable handles using the function registration table, TslFuncRegTbl. These properties are used to construct a task collection object that implements the split queue work stealing algorithm, explained

in Section 5.4. The choice of task collection can be explicitly specified by the programmer or chosen by the runtime depending on the task collection properties specified. The task collection objects are collectively created on a per-process or a per-thread basis.

The task collection objects are then seeded with tasks to begin execution. This allows for distributed locality-aware initialization. The copy overhead of task insertion in the above illustration can be avoided through in-place task initialization. The task collection, once seeded, is processed in a collective fashion using the process() method. Tasks, during their execution, can create additional tasks to be executed as part of the same or another task collection. The process() method returns when all tasks, seeded and subsequently created, have been executed and the task collection is empty, determined through a distributed termination detection algorithm. The process() method is the runtime equivalent of an X10 finish statement and corresponds to a *terminally-strict* sub-computation.

In an iterative computation, the task queue can be restored to its state prior to invocation of process() using the restore() method. This resets the termination detector and the task distribution enabling re-execution of the task collection. Execution profiles from the previous execution of the task collection can be used to adapt the seeding of tasks and scheduling algorithms employed for subsequent iterations.

5.3 Persistence-based Load Balancer

On each processor core, the persistence-based load balancer collects statistical data on the durations of each task executed in the current iteration. This load database is then used to rebalance the load for the next iteration after the current iteration has terminated.

The first step in utilizing the persistence properties that an application might exhibit is collecting data. Each task that is executed by a core is timed and stored in a local database. Storing the exact duration of each task (assuming a double-precision timer), requires $\Theta(n)$ doubles and $\Theta(n)$ task descriptors, where n is the number of tasks that each core executes. To reduce the amount of storage required, the scheduler times each task, truncates the duration, and bins it with other tasks that are of approximately

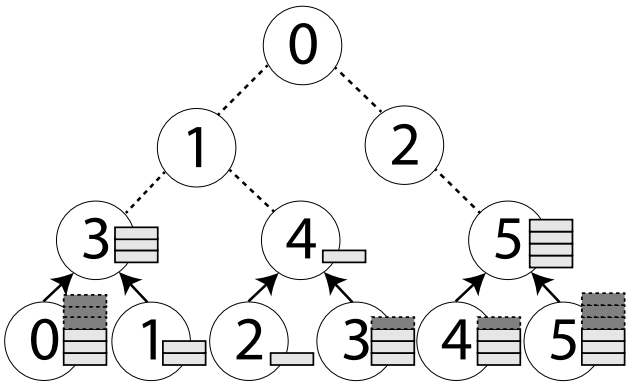
Algorithm 1: Centralized load balancer

```
1  begin
2      peLoad ⇐ { this processor's load };
3      lbD ⇐ { database of tasks };
4      localTaskPool ⇐ { empty task pool };
5      sumLoad ⇐ { distributed reduction };
6      avgLoad ⇐ sumLoad / number of cores;
7      while peLoad > C · avgLoad do
8          task ⇐ removeSmallestTask(lbD);
9          addTask(localTaskPool, task);
10         peLoad -= getDuration(task);
11     sendTo0(localTaskPool, peLoad);
12     if processor = 0 then
13         taskPool ⇐ { received tasks };
14         peLoads ⇐ { load for each PE };
15         makeMaxHeap(taskPool);
16         makeMinHeap(peLoads);
17         while t ⇐ removeMaxTask(taskPool) do
18             assign(t, getMinPE(peLoads));
19         { send out new tasks to each PE };
20     else
21         { receive new load from PE 0 };
```

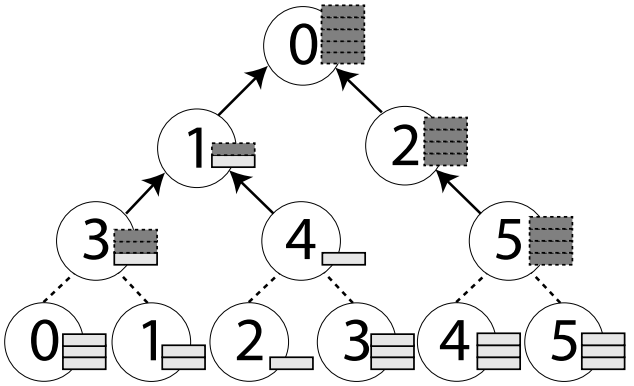
the same duration. This reduces the storage to $\Theta(n) + b$, where n is the number of task descriptors and b is the number of bins. These bins are kept in a sorted structure, allowing the load balancer to access the database in roughly duration order.

5.3.1 Centralized Load Balancing

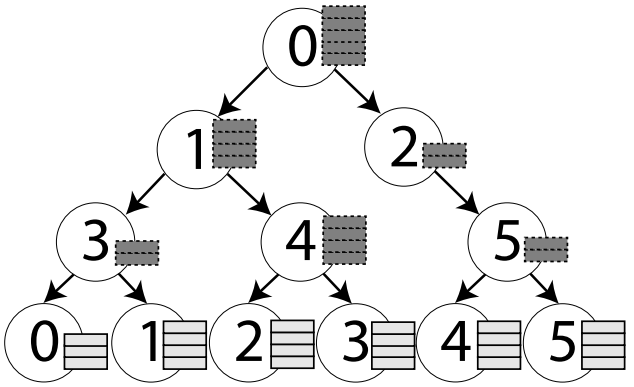
The baseline strategy, referred to as the centralized scheme, performs load balancing on one core, much like RefineLB as described by Zheng et al. [28]. Algorithm 1 details this strategy. First, the average load is calculated in parallel using a sum-reduction. If a core's load is sufficiently above the average load, the core removes the shortest duration task from its load database and moves it into a local task pool, until its load is below a constant, referred to as C, times the average load. These tasks (descriptors that compactly represent the task) in the local task pool, along with the core's new load, are



(a) All the leaves of the tree send their excess load (any load above a constant, referred to as C, times the average load) to their parent, selecting the work units from shortest duration to longest.



(b) Each core receives excess load from its children and saves work for underloaded children until their load is above a constant D times the average load. Remaining tasks are sent to the parent.



(c) Starting at the root, the excess load is distributed to each child applying the centralized greedy algorithm: maximum-duration task is assigned to minimum-loaded child.

Figure 5.1: The hierarchical persistence-based load balancing algorithm for 6 cores. The rectangles represent work units; a shaded rectangle with a dotted border indicates the work unit moves during that step.

sent to core 0, which attempts to redistribute the load.

Core 0 receives the donated tasks from overloaded cores, storing them into a task pool. It also receives each core's total load. From this data it creates a min-heap of the loads of each core and a max-heap of the durations in the task pool. It then assigns the longest task to the most underloaded core until the task pool is empty. These assignments are then sent to the cores.

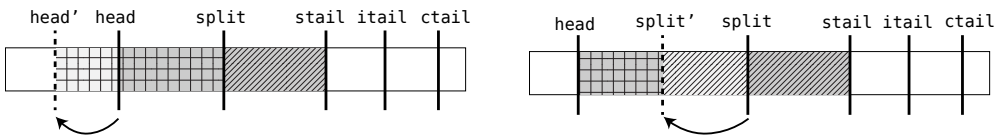
To maximize the scalability of this approach, the local task pool from each core is collected on core 0 using MPI_Gatherv, and new assignments are redistributed using MPI_Scatterv.

5.3.2 Hierarchical Load Balancing

Algorithm 2 details the procedure and Figure 5.1 illustrates the structure of the load-balancing tree. The cores are organized as an n-ary tree where every core is a leaf. First, the average load across all cores is determined. Each core locally chooses its shortest tasks to donate while reducing its anticipated load to be below the average times a threshold.

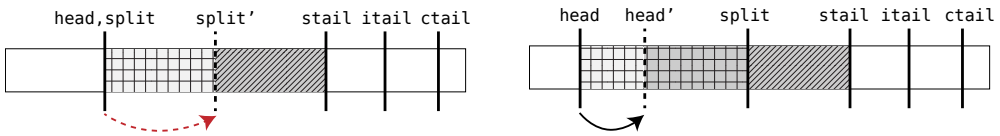
An underloaded core contains an empty set of tasks to be donated. Each core then sends its load information together with the set of donated tasks to its parent. Each core above the first level in the tree then receives the tasks and load from each child. These cores redistribute the donated tasks to balance the lightly loaded cores using the same procedure as the centralized algorithm. Each core then sends the total surplus or deficit in its sub-tree, together with donated tasks left over from the distribution, to its parent. This algorithm is repeated recursively up the tree to the root. The root redistributes leftover tasks down the tree to further improve load balance. Any tasks provided by a core's parent are redistributed down the tree to the leaf nodes, where they are enqueued for the next iteration.

This algorithm is greedy since it locally optimizes for the children of a node as work moves up the tree, assigning tasks to underloaded children, and down the tree to distribute the work. The average child load (total load of that subtree divided by the subtree size) at each node is used to make a local decision about which child to assign work. Such a greedy approach reduces communication and the amount of storage required at each level by considering only the immediate children and assigning them work immediately. This



(a) addTask() (b) releaseToShared()

Figure 5.2: Routines related to adding tasks into the deque



(a) acquireFromShared() (b) getTask()

Figure 5.3: Routines for a worker getting tasks from its deque

also reduces the amount of time to rebalance at each level because fewer tasks must be considered. In addition, tasks are assigned to locally deficit cores, better preserving data and topology locality than the centralized rebalancing scheme.

On the other hand, the quality of the load balance may diminish when the tree is extremely unbalanced because local decisions based on averages do not cause work to be migrated aggressively. Also, when there is a great disparity in the work unit size, large work units may be assigned to a locally deficit core rather than the most underloaded core, not effectively addressing the load imbalance problem.

Note that tasks themselves can be distributed directly from the donating core to the designated core, rather than through the tree. The greedy algorithm can also be applied to other organizations of the cores (e.g., torus) to better match the underlying communication network's topology.

5.4 Retentive Work Stealing

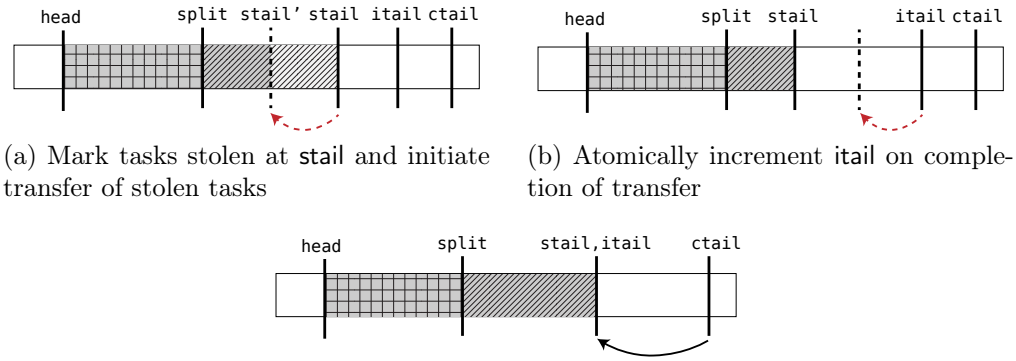
In this section, we present our implementation of distributed memory work stealing. Work stealing begins with all participating cores seeded with zero or more tasks. A core with local work takes the role of a worker, processing local tasks, as long as they are available. Once local work is exhausted, a worker becomes a thief, searching for other available work. A thief randomly chooses a victim and attempts to steal work from the victim's collection of tasks. On

Algorithm 2: Hierarchical greedy load balancer

```
1  begin
2      peLoad ⇐ { this processor's load };
3      lbD ⇐ { database of tasks };
4      localTaskPool ⇐ { empty task pool };
5      sumLoad ⇐ { distributed reduction };
6      avgLoad ⇐ sumLoad / number of cores;
7      while peLoad > C · avgLoad do
8          task ⇐ removeSmallestTask(lbD);
9          addTask(localTaskPool, task);
10         peLoad -= getDuration(task);
11     sendToParent(localTaskPool, peLoad);
12     { wait for children };
13     if processor is not root then
14         { wait for children };
15         taskPool ⇐ { received tasks };
16         peLoads ⇐ { load for each child PE };
17         makeMaxHeap(taskPool);
18         makeMinHeap(peLoads);
19         while t ⇐ removeMaxTask(taskPool) do
20             ∧
21                 getMinPELoad(peLoads) < D · avgLoad;
22             assign(t, getMinPE(peLoads));
23     else
24         { wait for children };
```

	HF-Be256	**HF-Be512 (20/40)**	**TCE**
Total tasks	84.9M	21.7B / 1.36B	470K
Non-null tasks	213K	9.10M / 862K	470K

Figure 5.4: Number of tasks in first (all tasks) and subsequent iterations (non-null tasks). The chunk size for HF-Be512 is shown in parentheses.



(a) Mark tasks stolen at stail and initiate transfer of stolen tasks

(b) Atomically increment itail on completion of transfer

(c) Worker updates ctail using the invariant in Equation 5.1

Figure 5.5: Steps in a steal operation

a successful steal, a thief returns to the worker state, continuing to process its local tasks. This procedure repeats until all workers have exhausted their tasks and termination is detected.

The randomness in the choice of the victim ensures quick distribution of work even in highly imbalanced cases (e.g., only one worker starts with all the work). If sufficient parallel slack is present, generally more cores will be in worker state rather than searching for work as a thief. Therefore, work stealing implementations, as pioneered by Cilk, attempt to avoid the overheads of synchronization or atomic operations on the executing worker, forcing much of the synchronization overhead on the thieves.

Shared memory implementations employ a deque in which the worker inserts the tasks on one end (referred to as the *head*), while thieves steal from the other end (the *tail*). Fence instructions are employed by the worker to ensure that its insertions at the head are visible to potential thieves in the correct order. A thief obtains a lock on the deque to preclude other concurrent steals. More details of the algorithm can be found in Blumofe et al. [22]

We employ a distributed-memory work stealing algorithm that considers the differing costs involved in distributed-memory machines.

```
Task taskBuf[BSIZE];//array holding tasks on the deque
Lock lock; //lock to arbitrate access to the deque
int head; //head: accessed only by worker
volatile int split; //split: worker reads/writes; thief reads
volatile int stail; //position of next steal; thief reads/writes; worker reads
volatile int itail; //intermediate tail: worker reads, thief writes
```

```
int ctail; //completed tail: accessed only by worker
Initial values:
  lock.unlock();
  head = split = stail = itail = ctail = 0;
```

A distributed-memory implementation of work stealing requires tasks to be copied to local memory, rather than just obtaining a pointer. On many architectures, such data transfer is more efficient from a contiguous memory rather than an arbitrary data structure (such as a linked list of tasks). We therefore employ a bounded-buffer circular deque implementation on a fixed-sized array.

The operations on the deque are depicted in Figures 5.2, 5.3, and 5.5. The dotted arrows correspond to locked or atomic operations, while the vertical dotted lines depict updated values for the variables being modified.

Each worker (a processor core in our case) allocates such a deque and associated state variables. The reuse of taskBuf allows it to reside in "registered" memory, enabling efficient data transfer. Rather than allowing all tasks in the deque to be stolen by a thief, we employ a split deque. All tasks between head and split are local to the worker owning the deque and cannot be stolen by a thief. This enables a worker to add and remove tasks at the head, without the potential need for fence instructions. The tasks past the split are in the shared portion of the deque and are available for stealing by thieves.

Individual remote memory operations (obtaining a lock, adjusting indices, releasing a lock, etc.) incur significant latencies. This not only increases the cost of a steal but slows down work propagation by precluding other steals. In order to enable contesting thieves to make quick progress, a split-phase stealing protocol is employed so that stolen tasks can be concurrently transferred while other thieves make progress. The shared portion between split and stail represents tasks that are available to be stolen. The shared portion between stail and ctail corresponds to stolen tasks that are being copied into the thieves' local memories.

The state of the deque can be identified by the following:

```
bool full() {
  return head==ctail && (split!=head || split!=ctail);
}
bool sharedEmpty() { return split==stail; }
bool localEmpty() { return head==split; }
```

```
int sharedSize() { return (split−stail+BSIZE)%BSIZE;}
int localSize() { return (head−split+BSIZE)%BSIZE;}
```

Note that the state of the deque queried without holding a lock is speculative if any of the variables associated with computing the state is marked as volatile. The only exception is split, which can only be modified by the worker and therefore can be read by it without holding the lock.

Adding a task into the deque by a worker (addTask()) involves inserting the task at the head. The method also resets the space available for insertion by adjusting ctail. This employs the invariant:

$$(\text{itail==stail}) \equiv \text{no pending steals on taskBuf[stail..ctail]} \tag{5.1}$$

```
void releaseToShared(int sz) {
  memfence();
  split = (split+sz)%BSIZE;
}
void addTask(Task task) {
  do {
    if(itail==stail) ctail = stail;
  } while(full());
  taskBuf[head] = task;
  head = (head+1)%BSIZE;
  if (sharedEmpty())
    releaseToShared(localSize() / 2);
}
```

When a worker observes the shared portion of its deque to be empty, it releases tasks from its local portion, shown by the routine releaseToShared(). Note that a memory fence operation is required while adding or getting tasks only when releasing work to the shared portion of the deque.

```
bool acquireFromShared() {
  lock();
  if(sharedEmpty()) return false;
  int nacquire = min(sharedSize()/2,1);
  split = (split − nacquire + BSIZE) % BSIZE;
  unlock();
  return true;
}
bool getTask(Task *task) {
```

```
  if(localEmpty())
    if(!acquireFromShared()) return false;
  *task = taskBuf[head];
  head = (head − 1 + BSIZE) % BSIZE;
  if (sharedEmpty())
    releaseToShared(localSize() / 2);
  return true;
}
```

Extracting a task from the deque first involves transferring tasks from the shared to the local portion, if the local portion is empty. The task at the head is then returned.

When a worker runs out of local work (getTask() returns false), it attempts to steal work from a random worker proc. This is implemented as an active message executed on the victim. The block of code executed as an active message is annotated by @proc in the code snippet. The victim's deque is locked and any tasks to be stolen are marked (using stail) before unlocking the deque. The stolen tasks are then transferred to the thief through an asynchronous operation (sendResponse()). When that operation is complete (detected by other runtime system components), the completion is noted in the deque by atomically updating itail. Note that multiple thieves could initiate the transfer of stolen tasks and complete the transfers out-of-order. Hence, ctail, which denotes completed steals, cannot be updated by a thief upon completion of its transfer, but needs to be updated by the worker using the invariant shown in Equation 5.1.

```
bool steal(int proc) {
  if(hasTerminated()) return false;
  //post a recv for incoming response
  @proc { //active message executed on proc
    lock();
    if(sharedEmpty()) { sendResponse(NULL); }
    int nsteal = min((sharedSize()+1)/2, BSIZE−stail);
    int oldtail = stail;
    int newtail = (stail + nsteal) % BSIZE;
    int nshift = newtail − oldtail;
    stail = newtail;
    unlock();
    sreq = sendResponse(taskBuf[oldtail..(oldtail+nsteal)]);
    when sreq.localComplete() {
      atomic itail += nshift;
```

```
    }
  }
  //wait on recv
  if(recvSize()>0) {
    lock();
    // adjust head, split, ...
    unlock();
    return true;
  }
  return false;
}
```

The stolen tasks are directly transferred from the victim's deque to the thief's deque, without additional copy operations. In the meanwhile, any attempts to steal from the thief would fail since the state variables denote the deque to be empty. On completion of a successful data transfer, and receipt of non-zero number of tasks, the state variables are updated under a lock to denote the availability of tasks.

```
void process() {
  while (!hasTerminated()) {
    Task task;
    while (getTask(&task)) {
      executeTask(&task);
    }
    bool got_work = false;
    while(got_work == false && !hasTerminated()) {
      do {
        p = rand() % nproc();
      } while (p == me());
      got_work = steal(p);
    }
  }
}
```

The overall execution procedure is shown above. All participating processor cores execute this routine. Each core executes all local tasks in the worker role, then turns into a thief searching for work. On finding work, the thief turns back into a worker. This cycle is repeated until termination is detected.

The tasks executed by each core are logged throughout the execution. In the next iteration, the processing begins with each core's local queue seeded

with tasks executed by it in the previous iteration. Given that work stealing attempts to dynamically balance the load each time it runs, this retained task distribution is anticipated to be more balanced than the initial seeding.

Essentially by retaining the previous execution profile, the advantages of persistence-based load balancing are applied to work stealing, and depending on the degree to which the persistence principle applies, the number of attempted and successful steals diminish in subsequent iterations with this optimization (shown in Figures 5.9 and 5.10). This strategy also maintains the primary advantage of work stealing: the ability of the runtime to adapt to major dynamic imbalances during an iteration.

```
set<Task> executedTasks;
void executeTask(Task task) {
  executedTasks.insert(task);
  // execute task
}
void restore() {
  foreach task in executedTasks {
    addTask(task);
  }
  executedTasks.clear();
}
while(..) {
  process();
  restore();
}
```

Note that the balance determined by the work stealing algorithm includes associated overheads. Therefore significant imbalance, together with work stealing, can still be expected in subsequent iterations. In Section 5.5, we show that this approach incrementally improves the load balance while also reducing the work stealing overheads.

Termination detection is done using Francez's termination detection algorithm [92], involving a tree in which thieves propagate termination messages in the form of rounds up and down the tree. Any round is invalidated by a thief that was a victim of a steal since the last round. Termination is detected when all workers have turned thieves, participated in the termination detection procedure, and none of them have been stolen from since the last round. The organization of the workers/thieves into a tree results in a

theoretical logarithmic overhead of termination detection after all tasks have been executed.

5.5 Experimental Evaluation

The experiments were performed on three systems: Cray XE6 NERSC Hopper [93], IBM BG/P Intrepid [94], and Cray XK6 Titan [95]. Hopper is a 6384-node system, each node consisting of two twelve-core 2.1GHz AMD 'MagnyCours' processors and 32GB DDR3 memory. Titan is a 18688-node system, each node consisting of one sixteen-core 2.2GHz AMD 'Bulldozer' processor and 32GB DDR3 memory. Hopper and Titan employ the Gemini interconnection network with a peak of 9.8GB/s bandwidth per Gemini chip. The Intrepid system consists of 40960 nodes, each with one quad-core 850MHz PowerPC 450 processor and 2GB DDR2 memory.

Our codes were compiled using the Intel compiler suite versions 12.0.4.191 and 12.1.1.256 on Hopper and Titan, respectively. Cray MPICH2 XT versions 5.3.3 and 5.4.1 were used on Hopper and Titan, respectively. On Intrepid, our codes were compiled with IBM XL C/C++ version 9.0. All communication was performed using two-sided MPI operations (`MPI_Isend()`, `MPI_Irecv()`, and `MPI_Wait()`), except the collectives employed in the persistence-based load balancing, as specified in Section 5.3. We developed a thread-based implementation with one thread pinned to each core throughout the execution, all of them sharing data, with MPI initialized in `MPI_THREAD_MULTIPLE` mode. We evaluated various configurations by varying the number of worker threads and "server" threads, and found that the best performance (in all the configurations we evaluated) was achieved with 23 worker threads and 1 server thread on Hopper, 15 worker threads and 1 server thread on Titan, and 3 worker threads and 1 server thread on Intrepid. We report all our results for these configurations. Given the server thread is still employed in communication progress for the application, we report all results as if the application is utilizing all the cores.

We evaluated the following schemes:

- **StealAll** Work stealing of all tasks in the calculation
- **Steal** Work stealing non-null tasks (same as StealAll for TCE)

- **StealRet** Retentive work stealing on non-null tasks
- **PLB** Persistence-based load balancing
- **Ideal** Ideal scaling expected, for comparison

5.5.1 Benchmarks

The algorithms presented were evaluated using the following two benchmarks:

5.5.1.1 Tensor Contraction Expressions

Tensor Contraction Expressions (TCE) [96] comprise the entirety of Coupled Cluster methods, employed in accurate descriptions of many-body systems in diverse domains. Tensors are generalized multi-dimensional matrices organized into dense rectangular tiles. A tensor contraction can be viewed as a collection of tile-tile products. The sparsity in the tensors, which can be algebraically determined through inexpensive local integer operations, induces inhomonegenity in the computation of dense tile-tile contractions, which vary widely in their computational requirements, spanning in structure from inner products to outer products.

5.5.1.2 Hartree-Fock

The Hartree-Fock (HF) method is a single-determinant theory [97] that forms the basis for higher-level electronic structure theories such as Møller-Plesset perturbation theory and Coupled Cluster theory. The benchmark consists of the two-electron contribution component of Hartree-Fock, the computationally dominant part of the method. The work to be performed is divided into smaller units based on the user-specific tile size. The work to be performed is determined by the schwarz screening matrix. Unlike the TCE benchmark, the sparsity induced by the schwarz matrix depends on the specific input and cannot be determined at compile-time. The screening produces variability in the execution time of individual tasks and potentially results in *null* tasks, i.e., tasks that do not perform any work. These null tasks are pruned in subsequent iterations.

5.5.1.3 Experimental Setup

For the Hartree-Fock benchmark, we considered two different molecular systems for evaluation, one consisting of 256 beryllium atoms (HF-Be256) and the other 512 atoms (HF-Be512). The matrices involved in the Hartree-Fock calculation (schwarz, fock, and dens matrices) were block-cyclically distributed amongst the cores with a tile size of 40 for evaluation on Hopper. A tile size of 20 was used on Titan and Intrepid to expose additional parallelism and enable evaluation on larger core counts. The number of total and non-null tasks is shown in Table 5.4. Note that the number of non-null tasks quadruples when the number of atoms is doubled. The tasks themselves do not necessarily incur the same execution time due to the sparsity induced by the schwarz matrix.

For TCE, we evaluate the following expression:

$$C[i,j,k,l] += A[i,j,a,b] * B[a,b,k,l]$$

Each dimension is split into four spatial blocks. Indices i, j, k, and l are organized into spatial blocks $240, 180, 100$, and 210; indices a and b are of size 84 and are organized into spatial blocks $20, 24, 20$, and 20. The tensors are partitioned into dense blocks with a tile size 20 and distributed in the global address space. Detailed explanation of the benchmark can be found in Baumgartner et al. [96]

5.6 Experimental Results

We observed that applications converged faster to the best achievable efficiency on Hopper and present five iterations for each application. Both schemes required many more iterations to converge to the best possible efficiency on Titan and Intrepid. We present results for the first fourteen iterations on these systems. Complete results for all configurations considered are not presented due to space limitations.

Persistence-based load balancing is typically performed only when significant load imbalance is detected to amortize the cost of load balancing. We load balance every iteration to quickly evaluate the effectiveness of load balancing. For the experiments, the load balancer constants C (Algorithm 1)

and D (Algorithm 2) were set to 1.003; the branching factor used in the hierarchical version was 3. The results presented do not include the load balancing cost, which is evaluated separately in section 5.6.2.

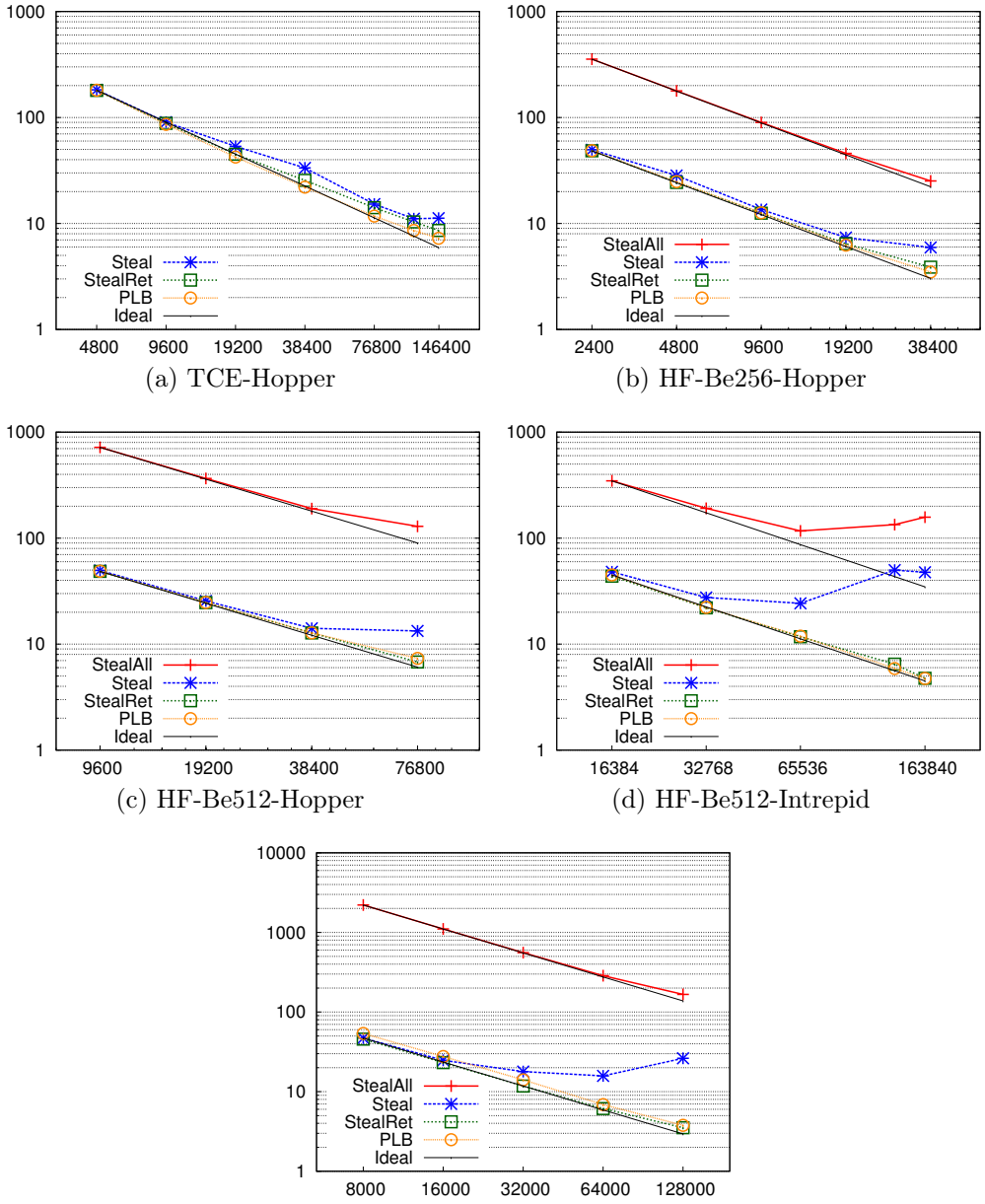
5.6.1 Scalability and Efficiency

The execution times of the various schemes for both applications are shown in Figure 5.6. For the HF benchmark, the execution time for stealing all tasks corresponds to the zero-th iteration of all runs, before pruning the null tasks, while other times presented correspond to the last iteration for the respective runs. Lines marking ideal speedup (with respect to the smallest core count shown in the corresponding graph as the baseline) are shown for comparison. For the HF benchmark, executing all tasks is significantly more expensive than executing only the non-null tasks. This is primarily due to the communication and computation associated with identifying the null tasks (checking the schwarz matrix). The schemes also scale well with increase in core count. Traditional work stealing scales better when executing all tasks, given the increased degree of available parallelism. This demonstrates that random work stealing can scale to large core counts provided sufficient parallelism.

When executing only the non-null tasks, work stealing is less strong-scalable than the other approaches evaluated. Persistence-based load balancing and retentive stealing achieve the best performance. The TCE benchmark exhibits super-linear scaling due to the working set fitting in the cache. This effect is countered by load imbalance at higher scales. Retentive work stealing appears to address the problems associated with work stealing as we increase the number of processor cores but keep the problem size fixed.

In order to better evaluate the observed performance and evolution of the schemes with iterations, we plotted their efficiency for each of the iterations on the non-null tasks. For each problem size, the efficiency is measured with respect to the best performance achieved by any of the schemes at any core count considered for that problem size.

The efficiency of retentive work stealing is shown in Figure 5.8. The efficiency of traditional work stealing is that achieved by the first iteration. Retentive work stealing steadily improves its efficiency with subsequent iter-



(a) TCE-Hopper

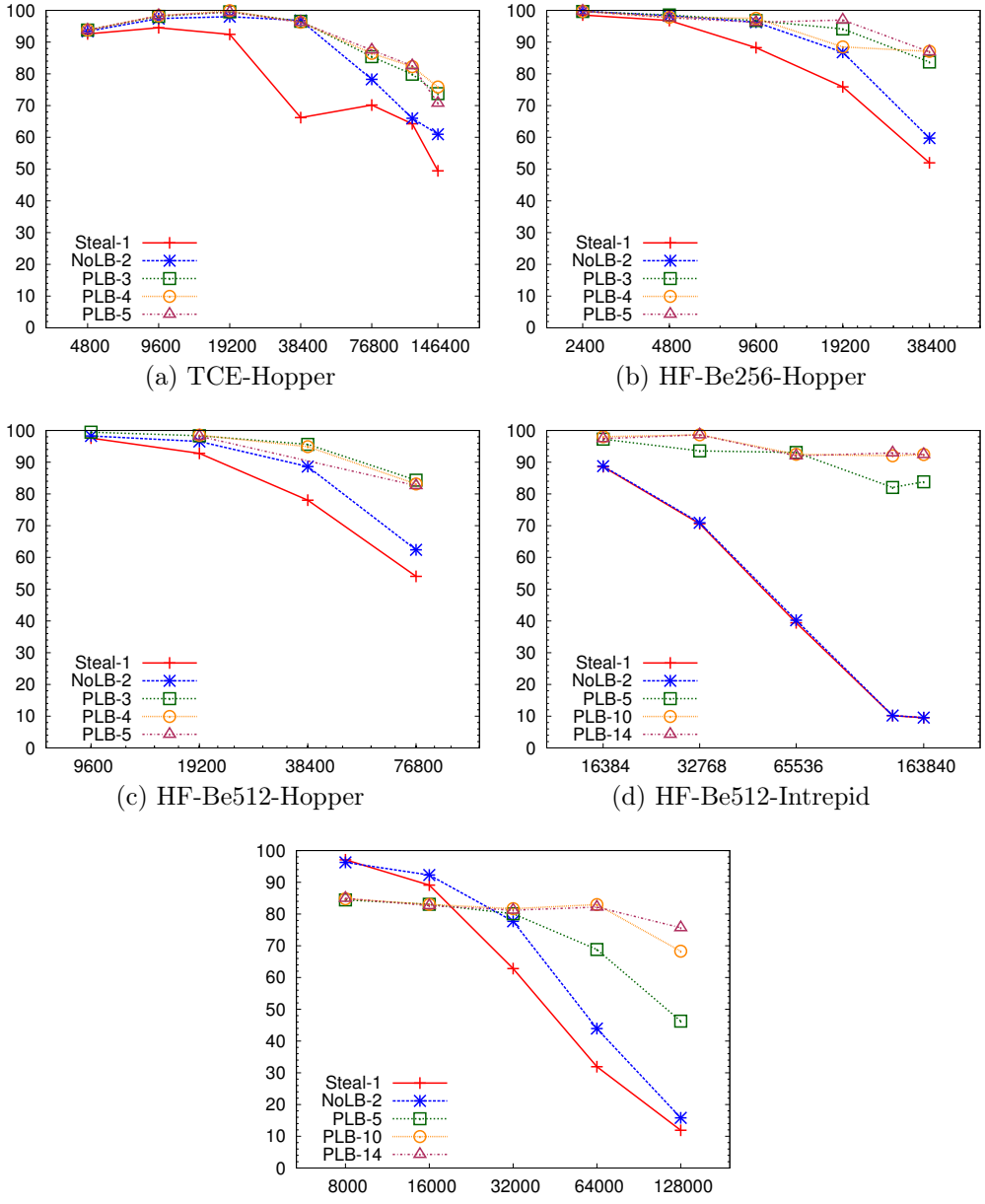
(b) HF-Be256-Hopper

(c) HF-Be512-Hopper

(d) HF-Be512-Intrepid

(e) HF-Be512-Titan

Figure 5.6: Execution times for first and last iteration. x-axis — number of cores; y-axis — execution time in seconds



(a) TCE-Hopper

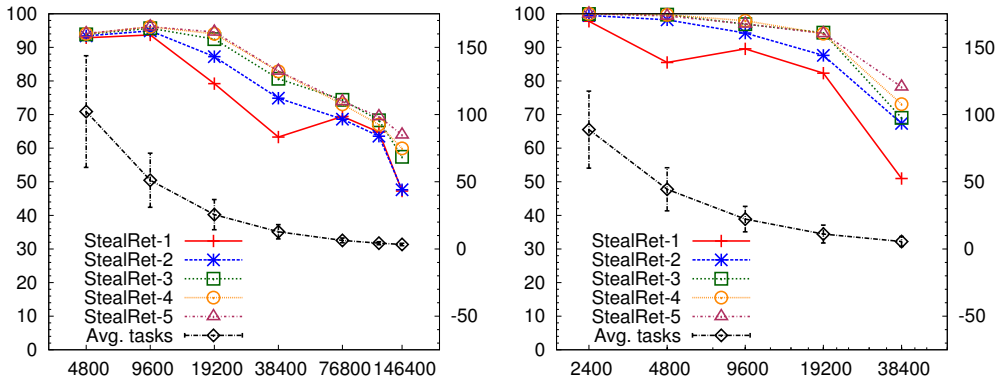
(b) HF-Be256-Hopper

(c) HF-Be512-Hopper

(d) HF-Be512-Intrepid

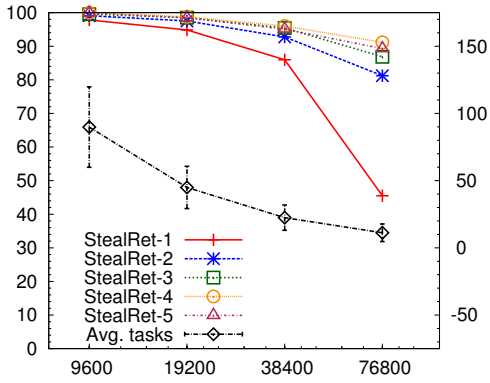
(e) HF-Be512-Titan

Figure 5.7: Efficiency of persistence-based load-balancing across iterations for the three system sizes, relative to the ideal anticipated speedup. x-axis — number of cores; y-axis — efficiency

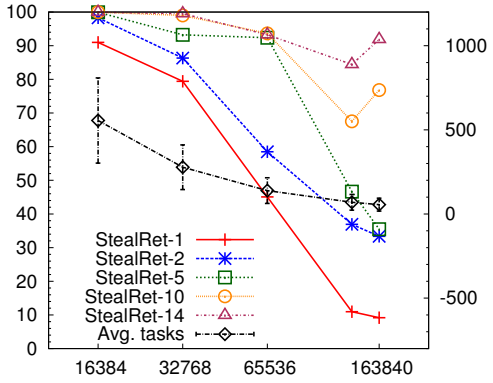


(a) TCE-Hopper

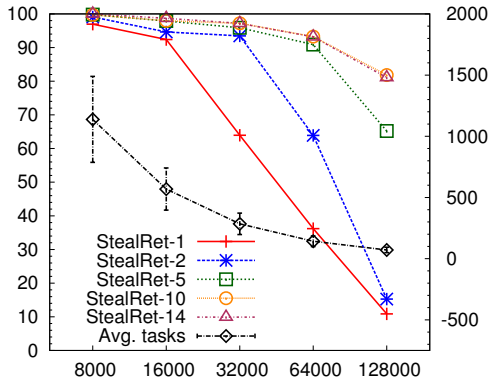
(b) HF-Be256-Hopper



(c) HF-Be512-Hopper

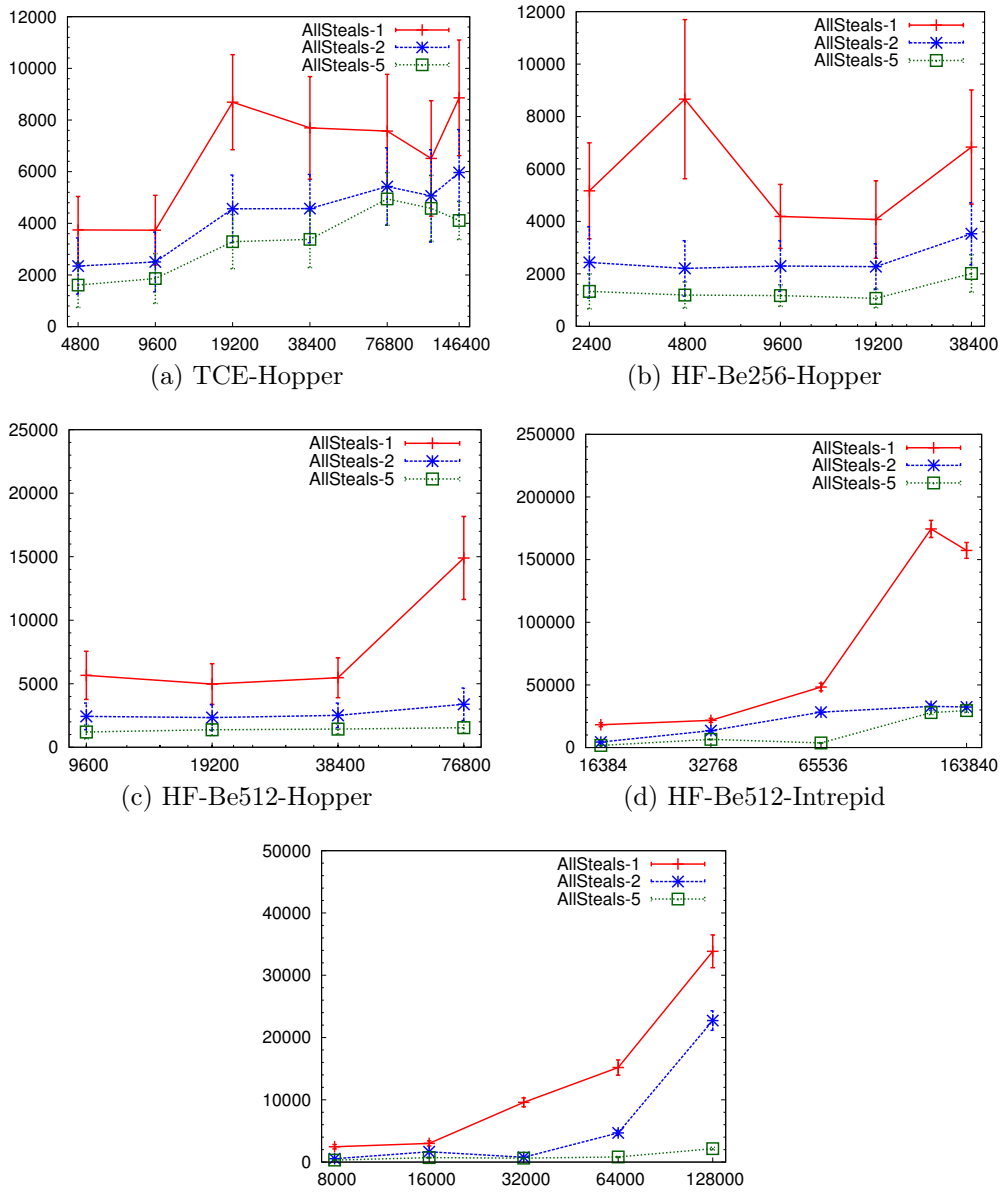


(d) HF-Be512-Intrepid



(e) HF-Be512-Titan

Figure 5.8: Efficiency of retentive work stealing across iterations relative to ideal anticipated speedup and tasks per core. x-axis — core count; left y-axis — efficiency; right y-axis — tasks per core (error bar: std. dev.)



(a) TCE-Hopper

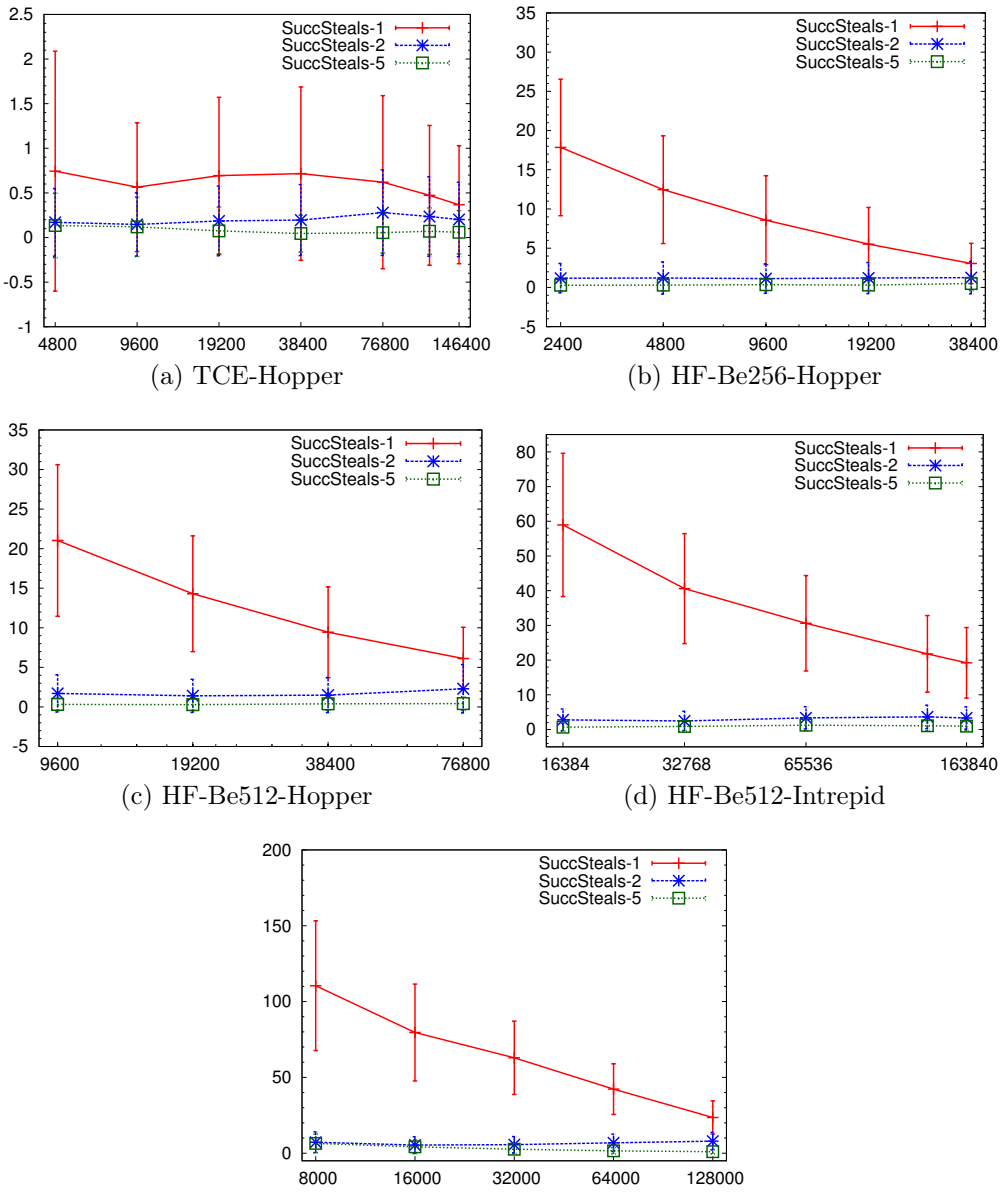
(b) HF-Be256-Hopper

(c) HF-Be512-Hopper

(d) HF-Be512-Intrepid

(e) HF-Be512-Titan

Figure 5.9: Average (error bar: standard deviation) number of attempted steals for the first, second, and fifth iteration of retentive stealing. x-axis — number of cores; y-axis — average number of steals.



(a) TCE-Hopper

(b) HF-Be256-Hopper

(c) HF-Be512-Hopper

(d) HF-Be512-Intrepid

(e) HF-Be512-Titan

Figure 5.10: Average (error bar: standard deviation) number of successful steals for the first, second, and fifth iteration of retentive stealing. x-axis — number of cores; y-axis — average number of steals.

Iter	Avg / Std All	Avg / Std Success	Avg / Std All	Avg / Std Success
	4800 Cores		9600 Cores	
1	3745/1294	0.7/1.3	3735/1350	0.6/0.7
2	2347/1086	0.2/0.4	2507/1155	0.1/0.4
5	1610/868	0.1/0.4	1864/964	0.1/0.3
	19200 Cores		38400 Cores	
1	8691/1840	0.7/0.9	7694/1987	0.7/1.0
2	4563/1306	0.2/0.4	4570/1325	0.2/0.4
5	3291/1049	0.1/0.3	3381/1095	0.04/0.2
	78600 Cores		146000 Cores	
1	7573/2196	0.6/1.0	8859/2238	0.4/0.7
2	5426/1496	0.3/0.5	5967/1663	0.2/0.4
5	4944/1019	0.1/0.2	4109/739	0.1/0.2

(a) TCE-Hopper

Iter	Avg / Std All	Avg / Std Success	Avg / Std All	Avg / Std Success
	2400 Cores		4800 Cores	
1	5168/1831	18/8.7	8662/3034	12/6.9
2	2436/1356	1.2/1.9	2209/1056	1.2/2.1
5	1333/667	0.3/0.7	1192/498	0.3/0.7
	9600 Cores		19200 Cores	
1	4190/1221	8.6/5.7	4073/1476	5.5/4.7
2	2298/966	1.1/1.9	2275/873	1.2/2
5	1171/412	0.3/0.8	1066/360	0.3/0.7
	38400 Cores			
1	6837 / 2178	3.1 / 2.6		
2	3528 / 1186	1.2 / 2.1		
5	2015 / 715	0.5 / 0.9		

(b) HF-Be256-Hopper

	9600 Cores		19200 Cores	
1	5661/1897	21/9.6	4975/1602	14/7.3
2	2428/1047	1.7/2.4	2336/1020	1.4/2.1
5	1202/549	0.3/0.8	1375/457	0.3/0.7
	38400 Cores		76800 Cores	
1	5471/1571	9.4/5.7	14899/3267	6.1/4
2	2512/965	1.5/2.2	3382/1278	2.3/3.1
5	1430/412	0.4/0.8	1537/439	0.4/0.9

(c) HF-Be512-Hopper

	16384 Cores		32768 Cores	
1	18220/1569	59.0/20.6	21736/1615	40/15
2	4386/633	2.8/3.1	13522/844	2.5/2.8
5	1697/231	0.7/1.1	6548/373	0.9/1.3
	65536 Cores		131072 Cores	
1	48315/3026	30/14	174496/6860	21/11
2	28249/1583	3.4/3.2	32904/1793	3.7/3.4
5	3694/318	1.2/1.6	27914/1620	1.1/1.5
	163840 Cores			
1	157353/6307	19/10		
2	32368/1813	3.3/3.2		
5	29479/1516	1/1.4		

(d) HF-Be512-Intrepid

	8000 Cores		16000 Cores	
1	2461/356	110/42	2994/410	79/31
2	538/113	7.3/6.9	1652/142	5.4/5.5
5	320/87	6.5/6.1	714/84	4.2/4.5
	32000 Cores		64000 Cores	
1	9592/713	63/24	15180/1225	42/17
2	770/118	5.6/5.3	4680/412	6.8/5.7
5	631/76	2.6/3.1	814/86	1.6/2.1
	128000 Cores			
1	33844/2634	24/11		
2	22725/1571	8.0/5.8		
5	2155/217	1.0/1.4		

(e) HF-Be512-Titan

Figure 5.11: Average (Avg) number and standard deviation (Std) of attempted and successful steals for retentive work stealing for TCE, HF-Be256, and HF-Be512 benchmarks on Hopper Cray XE6, Intrepid IBM BG/P, and Titan Cray XK6.

(a) HF-Be256-Hopper, 2400 cores: iterations are Steal, StealRet (last iteration), and PLB; corresponding timings are 50, 48.9, and 49 seconds.

(b) HF-Be256-Hopper, 9600 cores: iterations are Steal, StealRet (final iteration), and PLB; corresponding timings are 13.6, 12.6, and 12.2 seconds.

Figure 5.12: Utilization over time for all worker threads over time. Green region — task; Blue region — steal; x-axis — time; y-axis — percent utilization.

ations; for HF-Be512 the best points achieve over 90% efficiency on 76,800 cores of Hopper, over 91% efficiency on 163,940 cores of Intrepid, and over 81% efficiency on 128,000 cores of Titan. Given that the effectiveness of work stealing is influenced by the parallel slack, we plot the average number of tasks, with error bars showing the standard deviation across different processor cores, for each core count. This plot shows the similarity of task distributions across the problem sizes. For each problem size, the efficiency begins to deteriorate when less than 10 tasks per processor core are available on average. For a given number of tasks per processor core, the efficiency decreases with core count.

5.6.2 Cost and Effectiveness of Persistence-Based Load Balancers

In this section, we evaluate the effectiveness of the centralized and hierarchical persistence-based load balancers using micro-benchmarks constructed from the execution times for each task in the HF-Be256 run on Hopper. In order to simulate various degrees of load imbalance and stress the load balancers, the tasks are sorted by duration and distributed to the cores in a round-robin fashion, favoring every n-th core by giving it m tasks. For example, when $n = 2$ and $m = 4$ every other core receives 4 tasks, instead of 1 task.

Load Balancer Execution Time (seconds)						
		2400	4800	9600	19200	38400
n	m	C / H	C / H	C / H	C / H	C / H
1	1	1.57 / 1.42	1.98 / 1.12	5.76 / 1.08	14.8 / 1.92	35.6 / 1.04
2	2	4.83 / 0.07	6.33 / 0.16	8.42 / 0.81	13.0 / 0.77	43.8 / 1.00
2	4	7.40 / 0.35	10.0 / 0.66	13.0 / 0.33	18.3 / 0.31	37.7 / 1.14
2	8	9.24 / 0.34	11.4 / 0.30	13.0 / 0.33	18.4 / 1.62	45.2 / 4.40
4	2	1.40 / 0.08	2.00 / 0.30	3.82 / 0.48	16.9 / 0.82	39.5 / 0.75
4	4	2.84 / 0.09	4.13 / 0.20	6.65 / 0.56	13.4 / 0.36	37.0 / 0.65
4	8	4.34 / 0.28	6.27 / 0.97	9.07 / 0.58	15.1 / 0.26	43.0 / 0.85
8	2	0.45 / 0.04	0.75 / 0.11	2.73 / 0.25	15.0 / 0.79	36.2 / 0.36
8	4	0.98 / 0.06	1.52 / 0.07	3.03 / 0.19	11.9 / 0.65	38.1 / 0.55
8	8	1.88 / 0.06	2.70 / 0.10	5.01 / 0.49	11.4 / 0.38	36.7 / 0.63

Table 5.1: Execution time (seconds) on Hopper for rebalancing 12 initial distributions of tasks of the HF-Be256 system, comparing centralized (C) and hierarchical (H) persistence-based schemes. The ideal execution times are 48, 24, 12, 6, and 3 seconds, respectively.

By varying m and n, the performance of both schemes is evaluated under severe amounts of load imbalance. As the problem is scaled on more cores, due to the number of tasks remaining constant, much more load imbalance arises because there is more variance in duration between the tasks that each core selects. For example, when $n = 4$ and $m = 4$, on 2400 cores, the core with the highest load has 252% the average amount of work; on 38400 cores, the highest loaded core has 691% the average. As illustrated by these data points, the imbalance increases with the number of cores, necessitating that substantially more tasks be migrated.

Table 5.1 shows execution times and Table 5.2 shows the quality of the load balance achieved by both schemes on Hopper. The quality of load balance is measured as percentage deterioration in the execution time over the ideal load balance. The execution time for a given distribution of tasks is defined by the load on the core with the maximum load, while the ideal is the average of the load across all cores. Since the ideal load balance is not always achievable, we compare the hierarchical algorithm with the centralized algorithm.

The centralized load balancer achieves a consistently good load balance, given its global view of the computation at all scales. The variations in the load balance quality are due to the differences in tasks assigned to different

Load Balancer Quality (percent over ideal)						
		2400	4800	9600	19200	38400
n	m	C / H	C / H	C / H	C / H	C / H
1	1	0.03 / 0.06	0.03 / 0.18	0.03 / 0.4	0.03 / 2.0	5.8 / 12
2	2	0.03 / 0.14	0.03 / 0.54	0.03 / 1.1	1.7 / 6.6	5.8 / 12
2	4	0.03 / 0.26	0.03 / 0.68	0.03 / 4.0	0.03 / 7.1	5.8 / 12
2	8	0.03 / 0.28	0.03 / 2.2	0.03 / 3.7	0.1 / 7.3	5.8 / 18
4	2	0.03 / 0.17	0.03 / 0.53	0.03 / 1.3	2.8 / 5.7	5.8 / 11
4	4	0.03 / 0.29	0.03 / 0.60	0.03 / 2.4	1.3 / 7.4	5.8 / 15
4	8	0.03 / 0.29	0.03 / 0.58	0.03 / 3.9	0.03 / 6.9	5.8 / 13
8	2	0.03 / 0.30	0.03 / 0.73	0.03 / 1.2	2.9 / 5.4	6.6 / 13
8	4	0.03 / 0.50	0.03 / 1.0	0.03 / 3.1	2.3 / 7.6	5.8 / 12
8	8	0.03 / 0.49	0.03 / 1.0	0.03 / 4.6	0.9 / 7.9	5.8 / 15

Table 5.2: LHS — execution time (seconds) on Hopper for rebalancing 12 initial distributions of tasks of the HF-Be256 system, comparing centralized (C) and hierarchical (H) persistence-based schemes. RHS — load balance quality, computed as the maximum percentage over the ideal execution time (perfectly balanced load). The ideal execution times are 48, 24, 12, 6, and 3 seconds, respectively.

cores. The hierarchical algorithm suffers from worse load balance quality due to the greedy rebalancing employed. At scale, the few tasks available per core exacerbates the challenges encountered by greedy rebalancing. However, it is still competitive given sufficient number of tasks to rebalance. The execution times of the load balancers clearly demonstrate the benefits of the hierarchical approach.

5.6.3 Quantifying Work Stealing Overheads

In order to study the improvements in performance obtained by retentive stealing, we measured the number of attempted and successful steals for the various problems. Figure 5.9 and Table 5.11 show the number of attempted steal operations for the first, second, and fifth iterations. We observe that the number of steals, of any form, does not increase dramatically with an increased process count. For example, on iteration 1 of HF-Be512 on Hopper, the average number of attempted steals decreases from 5661 on 9600 cores to 5471 on 38400 cores.

If sufficient parallelism is present, the number of steals is influenced more by the problem size and initial task distribution than the number of concurrent executing cores. When the degree of application parallelism dries out, as happens in our application in the highest scale considered for each problem, steal attempts significantly increase. For iteration 1 of HF-Be512 on Hopper, increasing the number of cores from 38400 to 76800 increases the average number of attempted steals from 5471 to 14898. This trend is due to a lack of parallel slack. The high standard deviation shows that different cores are provided with different initial loads and attempt a varying number of steals to find work.

In order to examine work stealing overheads when there is sufficient parallel slack, we collected performance statistics for the HF benchmark on 300 cores of Titan not using retentive stealing. We found that the tasks take up over 85% of the total iterative time, and only 15% of the time is due to stealing and most of this time falls during the ramp-down time. In this experiment, because the application has sufficient parallel slack, the performance is good. Figure 5.13 and 5.14 depict a usage graph and time profile graph of the execution; red bars indicate stealing time whereas blue indicate task execution time. A histogram, shown in Figure 5.15, (log scale) of task duration shows for this particular problem how the the task duration is distributed. Note the long tail that ranges into over 2 seconds for the longest tasks. This contributes to the long ramp-down time during the end of the iteration.

The number of attempted steals provides insight into the improvements in the performance achieved by retentive stealing. As the load balancing becomes iteratively refined with retentive stealing, the number of attempted steals decreases and stabilizes across process counts. In addition, the gradual balancing of the load is accompanied by a lower standard deviation. Intuitively, when each processor core finishes the work initially assigned to it, all cores reach a similar state and the entire application is close to completion. For iteration 5 of HF-Be512 on Hopper, increasing the number of cores from 38400 to 76800 after retentive stealing has been applied for several iterations, only increases the average number of attempted steals from 1430 to 1537.

The number of successful steals, shown in Figure 5.10 and Table 5.11, confirms our intuition from the number of attempted steals. A successful steal depends on the availability of pending work, which decreases with increase

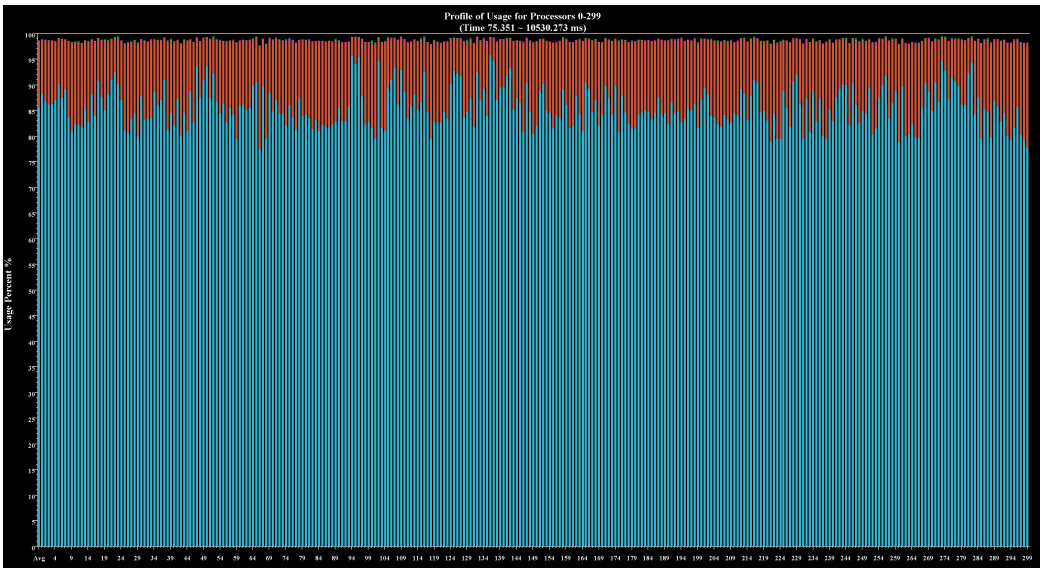


Figure 5.13: Usage graph of HF-Be32 on 300 Cores of Titan. Red represents stealing time and blue represents task execution time.

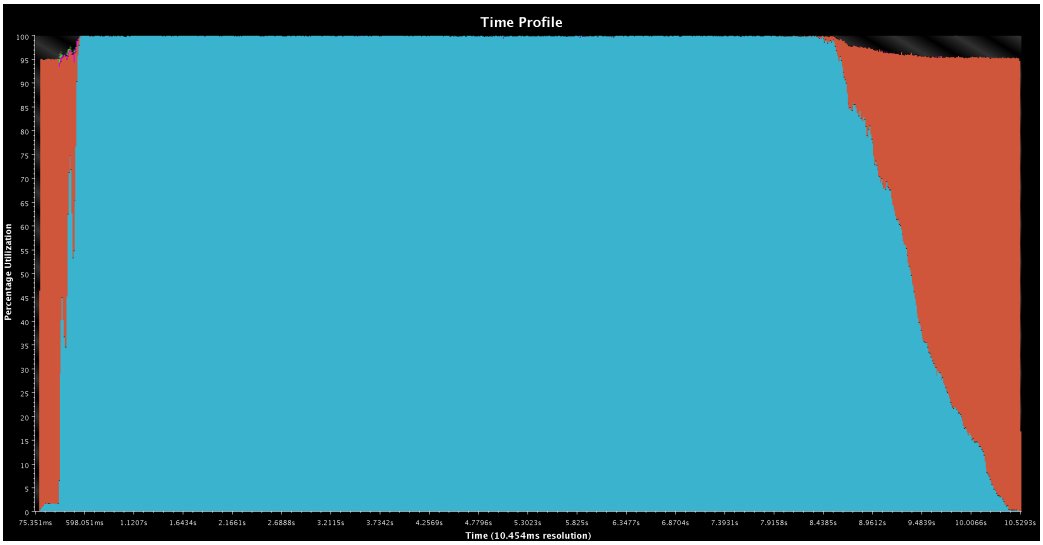


Figure 5.14: Time profile graph of HF-Be32 on 300 Cores of Titan. Red represents stealing time and blue represents task execution time.

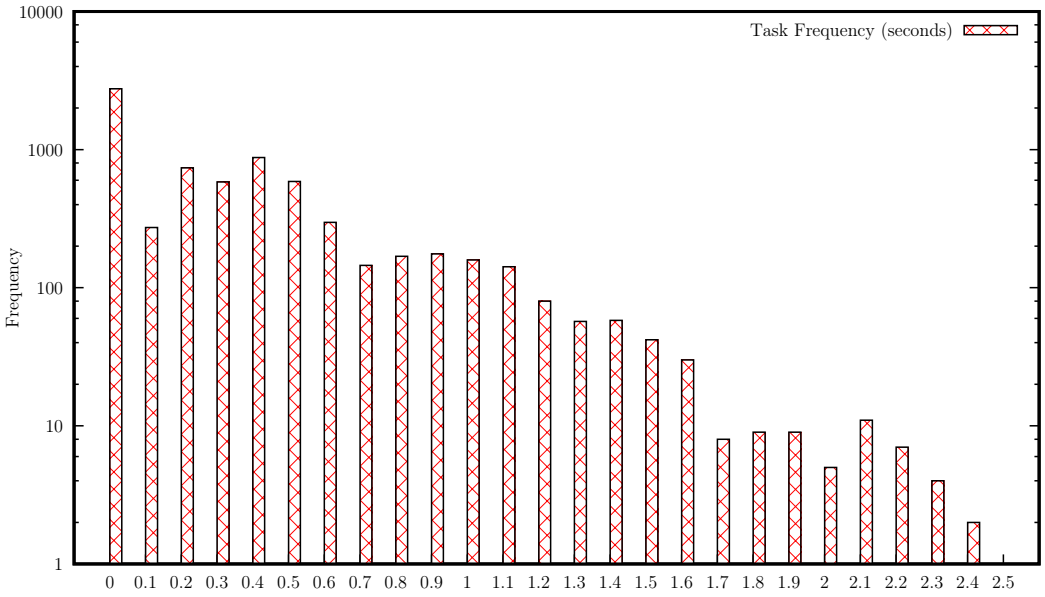


Figure 5.15: Histogram of task size for HF-Be32.

in core count. More importantly, retentive stealing balances the load well enough that for the last iteration the number of successful steals is very small. For iteration 5 of HF-Be512 on Hopper, for core counts 9600, 19200, 38400, and 76800, the average number of successful steals are 0.3, 0.3, 0.4, and 0.4, respectively.

The efficiency of persistence-based load balancing, using the hierarchical load balancer, is shown in Figure 5.7. The first iteration plotted on the graph is the efficiency of work stealing after the null tasks have been pruned. The significant change in the work distribution from iteration 0 to 1 renders persistence-based load balancing ineffective, resulting in large observed execution times. Instead, we resorted to work stealing in the first iteration, in this evaluation.

For the second iteration, to show the overhead of work stealing, tasks are seeded based on the previous iteration, but work stealing is disabled and no load balancer is used. Efficiency improves slightly in this case because the overheads associated with stealing and termination detection are removed. In subsequent iterations, the hierarchical load balancer is executed before the start of the iteration, using the data collected from the previous iteration. The first time the hierarchical load balancer is run, load balance improves significantly; for example, on Hopper efficiency increases by 7% on 76800 cores for the TCE calculation, 24% on 38400 cores for the HF-Be256 system,

and 22% on 76800 cores for the HF-Be512 system. The second application of the load balancer improves the performance slightly in the HF-Be256 case on Hopper. The third application does not seem to have much impact on Hopper, but Titan and Intrepid require more applications of the load balancer before HF-Be512 converges.

Figure 5.12 shows the processor utilization over time for two HF-Be256 runs on 2400 and 9600 cores of Hopper. The graph demonstrates that for the first iteration of the computation, before retentive stealing is applied, performance is low due to many cores spending time trying to steal tasks. As the problem is strong scaled, the amount of time spent stealing increases dramatically. Retentive work stealing reduces this time in both cases, increasing the amount of time spent executing tasks. With retention, work stealing performs almost as well as persistence-based load balancing.

5.7 Applying the Steal Tree to Retentive Work Stealing

While retentive stealing addresses the performance issues of work stealing in the distributed-memory context, this approach relies on 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 in Chapter 4. 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, 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 the following section, we show experimentally that these overheads do not adversely impact performance and the storage required stays well below the amount required to explicitly enumerate all tasks.

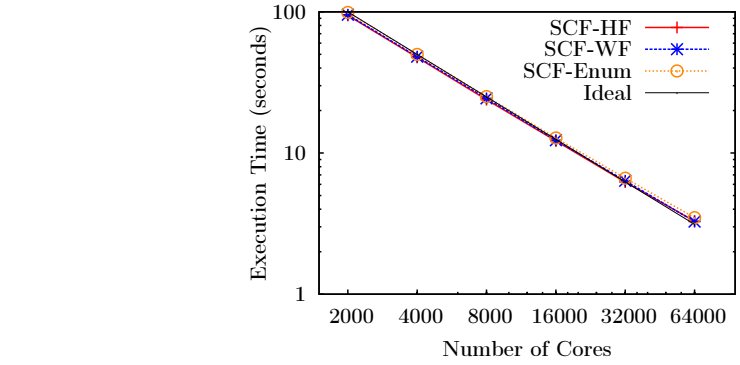
5.7.1 Experimental Results using the Steal Tree with Retentive Stealing

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 5.16a and 5.17a 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 the previous section. We graph the full task enumeration scheme used earlier in this chapter as TCE-Enum and SCF-Enum. We are able to reproduce this result without incurring the overhead of enumerating every task and storing them.

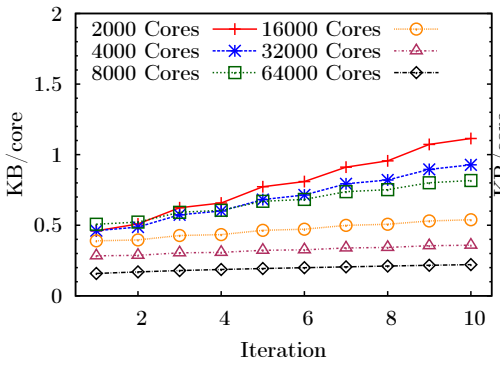
For SCF, fully enumerating the tasks requires 20.7 KB/core on 2K cores; for TCE, full enumerating the tasks requires 8.3 KB/core on 2K cores. For both benchmarks, the steal tree uses significantly less storage per core compared to enumerating all the tasks. For instance, on 2K cores of SCF with help-first 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.

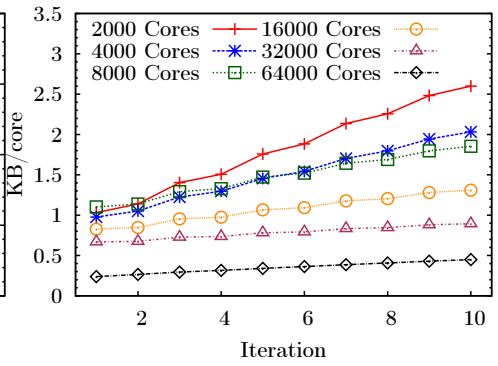
In Figures 5.16b, 5.16c, 5.17b and 5.17c we graph the amount of storage per core required over time for retentive stealing because steals in subsequent iterations cause more partitioning of the work. We observe that the convergence rate is application- and scale-dependent. For the SCF benchmark, the 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.



(a) Scaling

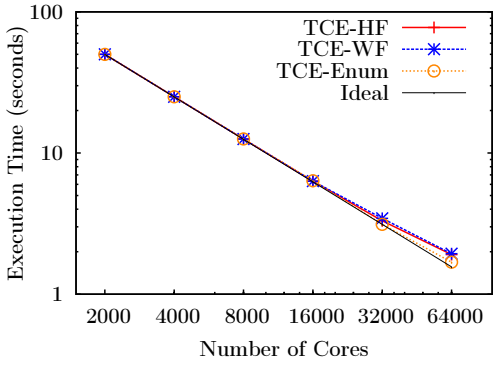


(b) Help-First

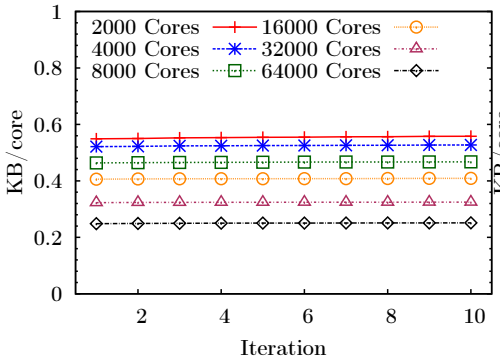


(c) Work-First

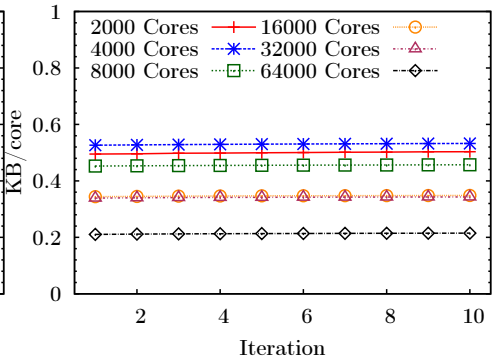
Figure 5.16: Retentive stealing using our tracing algorithms on recursive specification of the SCF benchmark on Cray XK6 Titan.



(a) Scaling



(b) Help-First



(c) Work-First

Figure 5.17: Retentive stealing using our tracing algorithms on recursive specification of the TCE benchmark on Cray XK6 Titan.

CHAPTER 6

NUMA Locality for Work Stealing

A fork/join parallel program scheduled using work stealing is typically oblivious to data locality and incurs data access penalties on multi-core systems with non-uniform memory and caches. Aligning the work performed, thus the data accesses, by a given processor across similar phases of a computation reduces its data access costs, potentially improving performance. However, this needs to be achieved without interfering with the essential dynamic load balancing benefits of fork/join programs scheduled using work stealing.

In this chapter, we present an approach to improve data locality across the phases of a fork/join program while ensuring load balance. We begin with the efficient extraction of an initial load balanced schedule for a given phase in the form of a *steal tree*. We present user-specified and automated approaches to steal tree extraction. User specification allows precise control over partitioning, while automated extraction minimizes user effort. The actions of the work stealing scheduler in subsequent phases are then constrained to match this initial schedule. A fixed schedule ensures data locality across phases, but it might not be effective in supporting phases with similar but not identical characteristics. Therefore, we design three constrained scheduling algorithms that follow a given schedule with varying degrees of fidelity: strict ordered, to precisely follow a given schedule; strict unordered, to improve schedule flexibility when waiting on steals; and relaxed, to permit additional steal operations while respecting the given schedule. We demonstrate the usefulness of these algorithms by devising two optimizations for fork/join programs.

Programs with non-cache resident working sets operating on systems with non-uniform memory access latencies can further benefit from data redistri-

bution. We treat data distributions as execution schedules of a fork/join initializer. This allows us to align data distributions with computation phases. We show these elements can be flexibly combined to improve the data locality and, in turn, the scalability of fork/join programs.

The performance of fork/join programs is significantly impacted by the amount of work encapsulated in each task. Fine-grained tasks allow effective load balancing while potentially incurring significant task management overheads. Coarse-grained tasks can reduce these overheads but potentially suffer from load imbalance due to lack of sufficient parallelism. Manual tuning of task granularity is a non-trivial challenge. We demonstrate the use of constrained work-stealing algorithms to dynamically coarsen the tasks in an iterative fashion.

These algorithms are implemented in the context of the Cilk runtime. The experimental evaluation demonstrates that initial schedules can be efficiently constructed, the constrained work-stealing algorithms effectively combine data locality and load balance, and these can be combined to greatly improve overall performance. The evaluation also demonstrates that dynamic coarsening automatically adapts the execution to achieve the performance benefits of manual coarsening while providing sufficient parallelism to ensure load-balanced execution.

The primary contributions of this chapter are:

- programmatic support for specifying work-stealing schedules to allow user guidance on data locality;
- constrained work-stealing algorithms that expose varying degrees of exactness and adaptivity;
- demonstration of data locality and dynamic coarsening optimizations; and
- implementation and detailed evaluation in the context of Cilk, demonstrating low overheads and performance improvements up to a factor of 2.5x on 80 cores compared to traditional random work stealing.

6.1 Background

In this section, we briefly describe the Cilk scheduler and its data access behavior. More detailed descriptions can be found elsewhere [35, 98].

A Cilk *closure* refers to the data structured in Cilk used to store the partially executed state of a task, or the state of the local variables in the function invocation. Execution begins with one of the threads executing the closure corresponding to the main() function. The actions of the scheduler can be described by the following loop:

```
void Cilk_Scheduler():
  foreach (w in workers):
    while (/*not terminated*/):
      Closure cl = try_steal_from_deque(w);
      if (!cl) cl = Closure_steal(w, random_victim());
      if (cl) execute_closure(cl);
void execute_closure(Closure* cl):
  // execute the corresponding closure starting at continuation
```

While shown to be provably space- and time-efficient, Cilk does not take data locality into account. In particular, we consider two challenges associated with executing fork/join programs that incur non-trivial data access costs. First, the lack of locality-awareness across computation phases significantly impacts performance. For example, we evaluated the execution time for performing a parallel memory copy between two 8 GB arrays on an 80-core system after both arrays have been initialized (system configuration is detailed in Section 6.3.3.4). The memory copy is organized as concurrent tasks operating on contiguous blocks of the array. A statically scheduled OpenMP loop aligned the initialization and copy operations and took 169 ms to perform the memory copy. Conversely, implementations using Cilk and OpenMP tasks took 436 ms.

Second, the performance of such programs is very sensitive to task granularity. The preceding results were obtained using a 512 KB block size, the best performing Cilk and OpenMP tasks version. Other block sizes, ranging from 4 KB (page size) to a few MBs, performed worse than this. While compute-bound programs can be optimized in terms of the smallest task granularity that maximizes sequential performance, optimizing data access costs imposes additional challenges.

6.2 Overview

In this section, we present an overview of our approach to data locality optimization for fork/join programs. Our objective is to match the actions of the work-stealing scheduler across different phases of a fork/join program. For example, making the same worker thread execute the initialization and copy tasks on a given data block would improve data locality and thus performance, resulting in the same performance as the OpenMP statically scheduled loops.

Here, we focus on a runtime approach to data locality optimization by matching the task execution schedules across different phases with user guidance. This involves efficient construction of a *template* schedule and algorithms to constrain the actions of the work-stealing scheduler to execute subsequent phases to follow a previously constructed template schedule.

We exploit the fact that the schedule for a fork/join program scheduled using work stealing can be described in terms of steal operations involved. These steal operations can be combined to construct a *steal tree* that represent a phase's execution schedule. We present two approaches to construct the template schedules. The first approach involves efficiently tracing the execution of a phase to extract the corresponding steal tree. The second approach involves user-specified partitioning of the work, where the user constructs a synthetic steal tree as the program is executed. Automated extraction of the steal tree (in Section 6.3.1) minimizes user effort but cannot immediately optimize for data locality and load balance. User-specified steal trees (discussed in Section 6.3.2) provide direct control to the user while requiring additional user effort.

We describe three constrained schedulers (discussed in Section 6.3.3) that present different trade-offs between faithfully preserving the template schedule and continuing to improve load balance. The two strict schedulers preserve the steal tree provided and can avoid the costs associated with work stealing. The relaxed replay scheduler incrementally load balances the computation, starting from the template schedule.

The following code snippet illustrates the optimization approach:

```
spawn initialization(); sync;
// s <- extract schedule from initialization
for (i = 0; i < numIters; i++)
```

```
if (!converged):
  // s <- use relaxed scheduler with s on kernel
  spawn kernel(); sync;
  // use strict scheduler with s on data relocation code
else:
  // use strict scheduler with s to maintain locality
  spawn kernel(); sync;
```

In this snippet, the user is interested in matching the schedules for the initialization (the initialization() spawn) and iterative kernel (the kernel() spawn) phases. The user extracts the schedule for the initialization phase into s. This schedule may not be optimal due to data locality inefficiencies in the initialization phase itself. In addition, the extracted schedule might not result in a load-balanced execution for the kernel phase. Therefore, the user employs the relaxed replay scheduler for the kernel phase. The data used in the kernel is redistributed to match this new relaxed schedule. When the performance has sufficiently stabilized (converged is true), the user disables work stealing and data redistribution and switches to the strict scheduler to ensure data locality for subsequent kernel phases.

To manipulate the data distribution, the user must write a fork/join initializer, a code that traverses the data with the same spawn/sync structure as the kernel, but instead copies and hence reinitializes the data. We use the fork/join initializer with the strict ordered scheduler so the data locality matches how the kernel was executed.

The following code snippet summarizes the application programmer interface to construct, manipulate, and replay schedules. The rest of the chapter examines the API and the associated algorithms in detail.

```
// extract Steal Tree from previous spawn
StealTree extractSchedulePrevious();
// map continuation to worker thread after next spawn
void designateAfterNextSpawn(int worker);
// apply ordered scheduler to next spawn
void applyStOWS(StealTree t);
// apply unordered scheduler to next spawn
void applyStUWS(StealTree t);
// apply relaxed scheduler to next spawn
void applyRelWS(StealTree t);
// prune the Steal Tree
void pruneTree(StealTree t, int percent);
```

6.3 Detailed Design

6.3.1 Automated Schedule Extraction

Cilk programs often are fine-grained to maximize the concurrency exposed to the runtime. Therefore, efficiently recording the schedule may be expensive to capture and store in terms of time and space. Thus, we use the steal tree described in Chapter 4 to record the schedule for the entire program. We use the theory presented in that chapter but employ a different design. Our goal beyond that chapter is to allow the extraction of a steal tree at any spawn in the computation. To implement this, we associate and track the information needed to construct the steal tree with every closure. When a steal occurs, we partition the steal tree at the closure, creating a new branch for the stolen continuation. For each steal, the steal tree stores a pointer to a child tree indexed by the continuation that was stolen.

```
struct StealTree:
  int thd; // the worker that ran this continuation
  int cont; // continuation starting point (parent's branch index)
  int seq; // unique sequence number for this worker−(thd,seq)
  map<int, StealTree> br; // the branches at each stolen continuation
struct Closure:
  // internal Cilk data, function ptr, etc.
  int curSpawn; // current spawn in scope being executed
  int level; // global spawn−tree level
  StealTree tree; // the steal tree for this closure
struct ThreadLocalData:
  int curSequence;
ThreadLocalData data[NUM_WORKERS];
```

The StealTree data structure completely specifies the mapping of continuations to workers, and the seq field specifies the order in which the continuations were executed. When a steal occurs, the newly allocated Cilk Closure is populated with a new StealTree. The corresponding branch in the old StealTree, indexed by the current spawn, is set to point to the new StealTree.

6.3.2 User-Specified Schedule Construction

In addition to extracting the schedule from a given phase of the computation, we present an approach to programming construct steal trees. In particular, the user can specify a mapping of a given continuation to a worker using the `designateAfterNextSpawn()` call. The following example specifies that the continuation beginning at `spawn y()` is to be executed by worker 1, and the continuation beginning at `spawn z()` is to be executed by worker 2.

```
void fn():
  designateAfterNextSpawn(1); // map cont. after x to worker 1
  spawn x();
  designateAfterNextSpawn(2); // map cont. after y to worker 2
  spawn y();
  spawn z();
```

The call to designate a continuation should precede the spawn whose continuation is being designated. User specification of the exact order in which the continuations mapped to a worker need to be executed is a non-trivial challenge. The strict unordered scheduler addresses this concern. A worker, on encountering this function, steals from itself and inserts the created closure into the steal tree as follows:

```
int designation[NUM_WORKERS] = {−1..−1};
void designateAfterNextSpawn(int worker):
  designation[get_current_worker()] = worker;
void pushed_spawn(int worker, int spawn, int curLev, Closure c):
  int contThd = designation[worker];
  designation[worker] = −1;
  if (contThd != −1 && contThd != worker):
    check_validity(c, curLev);
    // steal continuation from self
    Closure cont = Closure_steal(worker, worker);
    // transfer current steal tree to stolen continuation
    cont.tree = c.tree;
    // discard transferred steal tree from current continuation
    c.tree = c.tree.br[spawn];
    donate_continuation(contThd, cont);
```

We ensure that programmatically created steal trees follow the same properties as a steal tree constructed at runtime by the Cilk work-stealing sched-

uler. Principally, if a continuation in a task is mapped to a worker, the parent task (the task that spawned this one) also should have a stolen continuation based on the proofs in Chapter 4. This is true when the number of nested steals at this point equals the number of nested spawns minus one. If this is not true, then the parent task did not have a stolen continuation.

```
void check_validity(Closure cl, int curLevel):
  assert(curLevel == cl.level+1);
```

6.3.2.1 Schedule Extraction API

At this point, the schedule constructed can be extracted using the routine extractSchedulePrevious(). This routine returns a StealTree structure representing the steal tree rooted at the immediately preceding spawn. Note that the statements following the spawn can be executed while the spawned task, and those spawned transitively, continue to be executed. Therefore, a call to extract the steal tree must be preceded by a sync to ensure that the steal tree has been completely constructed before it is extracted. Given that the steal tree is requested *a posteriori*, we construct the steal tree for every spawn throughout the computation. An optimized implementation could include a split-phase design if a specification of the intent to extract a steal tree triggered the steal tree construction for a spawn. However, we observe that the overheads of steal tree construction are marginal in practice due to the fact that the steal tree construction operations are proportional to the number of steals, which are a small fraction of the total number of tasks in a fork/join program with sufficient concurrency.

6.3.3 Constrained Work Stealing

We have implemented three different scheduling algorithms that constrain a work-stealing scheduler to a template schedule with varying levels of fidelity. Depending on how refined and effective a schedule is for a given computation, it may need to be exactly followed or revised to adapt to changes in the environment (e.g., changes in the locality of data accessed). In Figure 6.1, we depict how the three types of constrained schedulers can improve a default schedule:

- **StOWS** The strict, ordered work-stealing scheduler exactly follows a template schedule t by guiding each worker to execute the tasks in the order prescribed by t.

- **StUWS** The strict, unordered work-stealing scheduler approximates a template schedule t by guiding each worker to execute the same tasks it executed in t, but it allows them to greedily deviate in order (ensuring that all dependencies are followed).

- **RelWS** The relaxed work-stealing scheduler approximates a template schedule t by guiding each worker to execute the tasks it executed in t in any order, while allowing further steals when a worker is idle.

Constraining the execution to a template schedule requires coordinating the workers so that each worker steals the same closures as dictated by the schedule. Depending on the level of fidelity, the order may matter or further steals may be allowed. A possible method to implement this involves coordinating the thieves so they steal from the same victims as specified during the designated working phase. However, this may slow down the victim if it has to wait for the thief to steal due to perturbations in the execution or schedule variations from lower levels of fidelity (unordered or relaxed work stealing). Hence, we have implemented all of the scheduling algorithms using a donation protocol: when a continuation marked as stolen in the template schedule is encountered, the victim steals the next continuation from itself and donates the closure to the worker designated in the steal tree.

The common operation in constrained work stealing executes a closure in the current constrain mode. Once a spawn is encountered and pushed on the stack, if there exists a steal point in the steal tree for the continuation after the spawn, the worker steals from itself and passes the stolen continuation to the worker designated as thief. Henceforth, we shall refer to the thief that steals a continuation in a template schedule as the *designed worker* for that continuation.

```
enum constrain_mode {RWS, StOWS, StUWS, RelWS};
constrain_mode current_mode = RWS;

void execute_closure(Closure cl, constrain_mode m):
  // set global work stealing constrain mode
```

```
  current_mode = m;
  // execute continuation

// executed after a spawn is encountered and pushed on the stack,
// incrementing the current level
void pushed_spawn(int worker, int spawn, int curLevel, Closure cl):
  if (current_mode != RWS):
    int contThd = cl.tree.br[spawn+1].thd;
    if (contThd != worker):
      // steal continuation from self
      Closure cont = Closure_steal(worker, worker);
      // transfer current steal tree to stolen continuation
      cont.tree = cl.tree;
      // discard transferred steal tree from current continuation
      cl.tree = cl.tree.br[spawn];
      if (current_mode == RelR || current_mode == StUR):
        donate_continuation(contThd, cont);
      else if (current_mode == StOR):
        int seq = cont.tree.br[spawn+1].seq;
        seqs[contThd][seq].cur = cont;
        seqs[contThd][seq].ready = true;
```

The spawn/sync structure of the computation being constrained does not have to exactly match the template schedule. While the structure can be arbitrarily different, it can only deviate in two ways to be semantically valid.

Rule 6.3.1. *The template schedule and constrained computation may vary in task depth. The template may have steal points beyond the computational structure, or the computation may have deeper tasks. Deeper tasks in the computation are constrained by the deepest parent task in the template with a steal point.*

Rule 6.3.2. *To match, the spawn/sync structure must be aligned until the position of the continuation. This could be followed by an arbitrary task structure of spawns and syncs.*

6.3.3.1 Strict Ordered (**StOWS**) Scheduling

This scheduling policy is used when the template schedule exactly matches the computation and is known to provide good load balance and data locality. This policy exactly reproduces a template schedule: the mapping of a

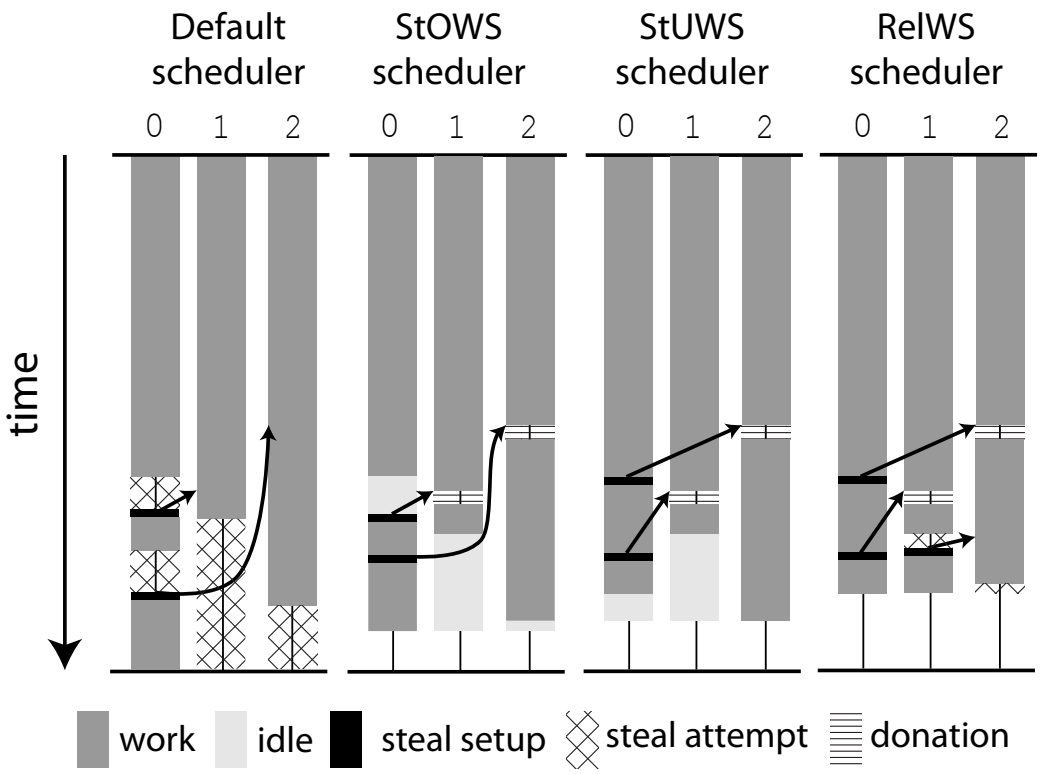


Figure 6.1: Example computation scheduled with the default scheduler and then modified with three types of constrained work stealing. Default scheduler: all workers begin busy with work then attempt to steal when they finish. StOWS reproduces this schedule without having to search for work. StUWS is able to revise the order on thread 0, reducing the time. RelWS performs an additional steal, further balancing the workload.

continuation to a worker and the order in which each worker executes the continuations. Compared to the default Cilk scheduler, the StOWS scheduler has the advantage that workers do not have to search for work, which reduces the execution overhead. The donations do not incur much overhead because they require little coordination between the workers.

To reproduce the order of previous execution, each worker creates an ordered list of steal tree branches that it executed previously from the designated spawn. This list is built by traversing the steal tree at that spawn and finding the minimum and maximum sequence numbers for each worker. This operation can be parallelized by performing a parallel tree traversal and using Cilk++ reducers [98] or an equivalent Cilk implementation to find the global minimum and maximum for each worker in the subtree.

Once the ranges are found, sequence arrays are built for each worker for following the sequence of closures to execute during StOWS:

```
struct Sequence:
  boolean ready = false;
  Closure cur;

Sequence seqs[NUM_WORKERS];

void buildWorkerSequences():
  foreach (w in NUM_WORKERS):
    int phases = range[w].max − range[w].min;
    seqs[w] = allocate_init(phases);
```

During StOWS, workers do not perform random work stealing. Instead, they start at the beginning of their sequence arrays and wait for the next element in the sequence to become ready (ready field set to true) and the closure to be passed to that worker by setting the cur field. The first closure is given to the thread that executed the root of the subtree. After a thread pushes a spawn, it checks the steal tree to determine if there was a steal, and donates the continuation if there was by setting the cur field in the proper location in the sequence specified by the steal tree.

The following algorithm describes the StOWS scheduler. The initial closure is passed to the starting sequence. Then each worker waits until the next closure activates in the sequence.

```
void StOWS_Scheduler(Closure starting):
  buildWorkerSequences(seqs);
```

```
int startThd = starting.tree.thd;
int startSeq = stating.tree.seq;
seqs[startThd][startSeq].ready = true;
seqs[startThd][startSeq].cur = initial;
foreach (w in workers):
  for (i = 0; i < length(seqs[w]); i++):
    while (!seq[w][i].ready)
      ;
    execute_closure(seq[w][i].cur, StOR);
```

6.3.3.2 Strict Unordered (**StUWS**) Scheduling

The strict unordered scheduling policy is used when a template schedule provides a good mapping of tasks to threads, but the ordering in the schedule must be refined to maximize performance or the ordering is unspecified (e.g., a user-specified schedule construction). The advantage of this algorithm over the other schedulers is that it allows the system to adapt to small perturbations in the execution without incurring the overhead of attempting and coordinating steals.

The strict unordered scheduler ensures that each worker executes the same work as the template schedule, following the computational dependencies, but relaxes the order in which concurrent stolen tasks are executed. More specifically, if two concurrent tasks were executed by a worker in the previous schedule, they will both be executed by the same worker, but possibly in a different order than what the template schedule dictates.

Unordered execution is achieved by using the steal tree to determine the mapping of tasks to threads, but ignoring the sequence information. To store donations from multiple concurrent workers, each worker maintains a bounded buffer that is protected by a lock. If the bounded buffer is empty and the worker is idle, the worker spins, waiting for work to arrive. The bounded buffer may grow in size up to a system-imposed limit. When the limit is reached, any donating workers spin until a closure is removed from the buffer. The management of the bounded buffer causes this scheme to incur slightly more time and space overhead compared to StOWS.

The following algorithm describes the scheduler. The initial closure is deposited into the designated bounded buffer, then each worker checks the buffer for any new closures. If the continuation following a spawn is desig-

nated to be stolen, the worker donates it to the appropriate worker's buffer.

```
void StUWS_Scheduler(Closure starting):
  int startThd = starting.tree.thd;
  donate_continuation(startThd, starting);
  foreach (w in workers):
    while (/*not terminated*/):
      while (/* no continuations ready*/)
        ;
      Closure cl = try_extract_continuation(w);
      if (cl):
        execute_closure(cl, StUR);
```

6.3.3.3 Relaxed (**RelWS**) Scheduling

Due to environmental changes, such as the data locality of tasks, growing load imbalances, or execution perturbations due to noise, a template schedule may need to be revised. We have developed the RelWS algorithm for approximating a template schedule by following it as much as possible, but deviating from it when a worker is idle, indicating that the schedule has a deficiency at this point.

Similar to StUWS, RelWS does not follow the order and uses the bounded buffer to transfer continuations between workers. When the bounded buffer is empty, the worker becomes a thief. If the worker performs a successful steal, it continues to follow the template schedule for the stolen continuation. Any descendent continuations from this stolen continuation that are marked as stolen in the template schedule continue to be treated as steals and get donated to the designated worker. Therefore, each overriding steal only modifies, at most, one branch of the tree.

The primary advantage of RelWS is that it can adapt to changes that may arise. However, it does incur the most overhead of the three polices due to its use of the bounded buffer and the stealing overhead when the buffer is empty.

The following algorithm shows how the relaxed scheduler functions:

```
void RelWS_Scheduler(Closure starting):
  int startThd = starting.tree.thd;
  donate_continuation(startThd, starting);
  foreach (w in workers):
```

```
while (/*not terminated*/):
  Closure cl;
  if /* ready continuations of w not empty */:
    cl = try_extract_continuation(w);
  if (cl):
    execute_closure(cl, RelR);
  else:
    cl = Closure_steal(w, random_victim());
  if (cl):
    execute_closure(cl, RelR);
```

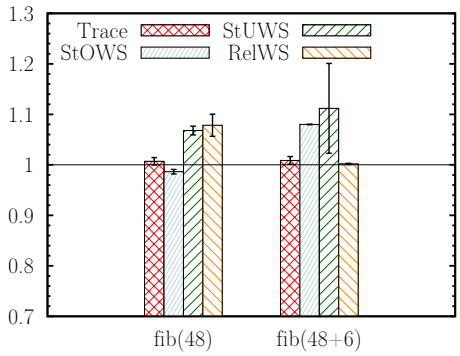
We now evaluate the overhead of building the steal tree and adaptability of using the constrained schedulers with the recursive Fibonacci benchmark (fib) implemented in Cilk. For all of the experiments conducted with fib, we calculate the 48th Fibonacci number, unless specified differently. When we reach the depth of fib(30), we invoke a sequential kernel.

6.3.3.4 Experimental Setup

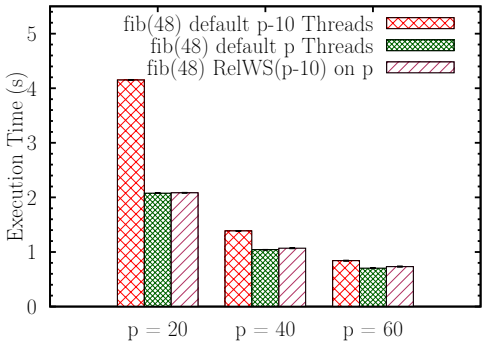
All of the experiments in this chapter were performed on an Intel 80-core machine, composed of eight 2.27 GHz E7-8860 processors, each with 10 cores. They are connected via Intel QPI 6.4 GT/s, and the machine has 2 TB of DRAM. All our codes were compiled with GNU GCC version 4.3.4, using the MIT Cilk 5.4.6 translator [35] or just with GCC and OpenMP 3.0 (version 200805). For the OpenMP results, we tried using ICC with the Intel OpenMP implementation, but found no significant scaling difference. The machine runs Red Hat Linux version 4.4.7-3 and has been configured to use a page size of 4096 bytes. All of our codes set the affinity of created threads that pins each thread (in Cilk or OpenMP) to a specific core during the execution. The first 10 threads created are pinned to a single socket.

6.3.3.5 Overhead Evaluation

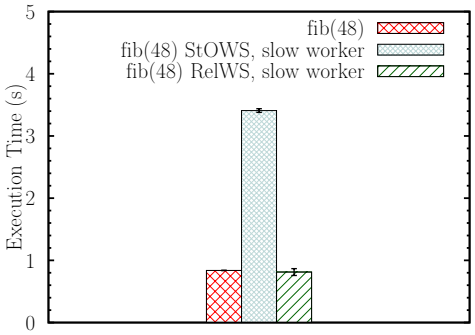
In Figure 6.2a, the first set of bars plots the normalized execution time compared to executing fib without tracing for the four different configurations. Building the steal tree, shown as "Trace" on the plot, incurs very little overhead and is within the standard deviation. We observe that the strict ordered scheduler speeds up execution by 1.4%, but unordered and relaxed work steal-



(a) Normalized ratio (inset left) of tracing and the three constrained scheduling schemes for fib on 80 cores compared to baseline. Normalized ratio (inset right) using fib(48) as a template schedule for fib($48+6$) compared to native execution.



(b) We execute fib(48) on $p-10$ and p cores and compare this to using RelWS on the schedule for $p-10$ cores. RelWS adapts well to this situation.



(c) The resulting execution when one worker out of 80 is arbitrarily slowed down. The strict ordered scheduler increases the execution time by nearly 4 times. RelWS adapts achieving close to the same speed compared to a normal execution.

Figure 6.2: The overhead of the fib microbenchmark with tracing and three scheduling schemes (6.2a) along with benchmarks to show how RelWS adapts to dynamic variations (6.2b, 6.2c).

ing incur an execution time penalty of about 6.8% and 7.8% with standard deviations of 0.8% and 2.2%, respectively. This matches our expectation that the strict ordered scheduler slightly improves performance if the computation is sufficiently load balanced and the schedule is appropriate, but unordered or relaxed schedulers may impose overheads if they are not needed to refine the schedule.

6.3.3.6 Adaptability Evaluation

In the second set of bars, we test the efficacy of RelWS by using a schedule from a smaller problem size for a larger problem to observe how it adapts. We first execute fib(48), extract the schedule, and use that schedule as a template for fib($48 + 6$). We compare this to running fib($48 + 6$) with the default Cilk scheduler. We find that using the strict ordered scheduler incurs a performance penalty of around 8% due to scheduling deficiencies. This is from the mismatch between the template schedule and the work being performed.

The strict unordered scheduler causes high average overhead but with a large standard deviation, indicating that the execution time is unpredictable, because the unordered scheduler has varying performance depending on the order of execution. However, RelWS recovers the lost performance entirely by adapting to the new problem size, achieving performance close to the native schedule.

In Figure 6.2b, we vary the number of working threads to further show the flexibility of the RelWS scheduler. The first bar in each set plots the execution time from fib(48) with $p - 10$ threads. The second bar shows the execution time with p threads. In the third bar, we show the execution time using the schedule produced with $p - 10$ threads as a template for p threads. We observe the performance almost matches a schedule natively generated by Cilk for the scenario with RelWS. In the case of $p = 20$, the performance is within 0.35% of the native. For $p = 40$, it is 2.6%, and for $p = 60$, it is 3.97%.

In the final fib experiment shown in Figure 6.2c, we present the baseline execution time for fib(48). Then, we arbitrarily slow down a single worker by inflating the size of every task at the bottom of the tree for only those tasks the slow worker executes. This is performed by enlarging every task the slow

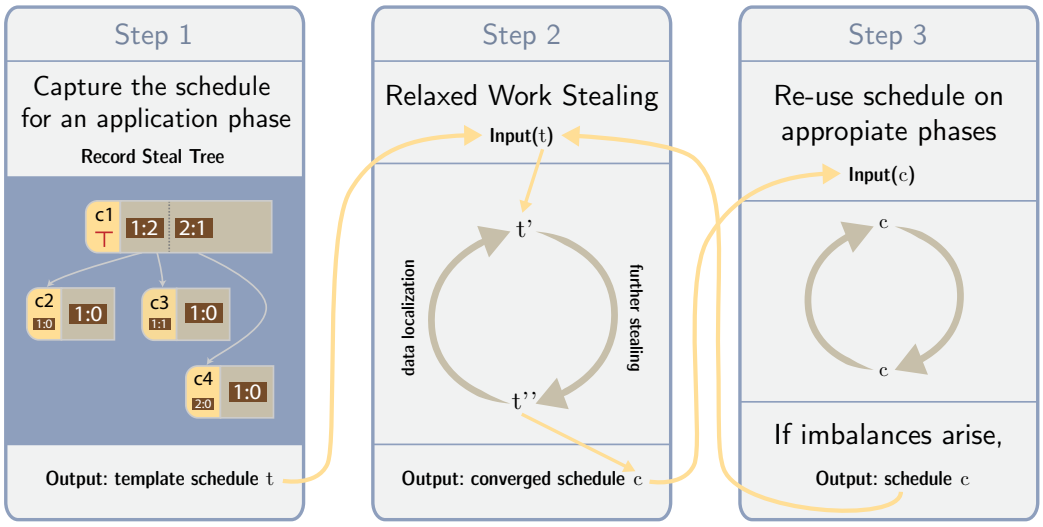


Figure 6.3: Overall process applied to iteratively improve locality.

worker executes from fib(n) to fib($n+3$). Using the strict ordered scheduler with a slow worker increases the execution time by a factor of 4. Using RELWS on this same schedule restores the performance almost entirely by stealing work away from the slow worker.

6.4 Whole Program Data Locality Optimization

We demonstrate the usefulness of the algorithms presented for optimizing data locality for six benchmarks. To iteratively optimize data locality, we start with a template schedule that may be extracted from the data initialization code, depending on whether of not the initialization code has similar structure to the kernel. If not, the initial template schedule is derived from applying random work stealing on the kernel code. After we apply RELWS to the template schedule for the kernel, we redistribute the data by invoking a fork/join initializer that copies and reinitializes the data constrained by the strict ordered scheduler. We iteratively apply this method to gradually localize data and correspondingly load balance the schedule. For the benchmarks tested, we found that the schedules and data distributions converge quickly, within about three to five iterations. This process is shown in Figure 6.3.

```
spawn initialization(); sync;
StealTree t = extractSchedulePrevious();
for (i = 0; i < numIter; i++):
```

```
if (!converged):
  applyRelWS(t); spawn kernel(); sync;
  t = extractSchedulePrevious();
  applyStOWS(t); spawn forkjoin_initializer(); sync;
else
  applyStOWS(t); spawn kernel(); sync;
```

For iterative computations, we employ two metrics to determine when the execution can be considered to have converged to a good schedule: (1) idle time and (2) number of non-uniform memory access (NUMA)-remote accesses. Time spent waiting for a closure to be ready or looking for work is considered idle time. A schedule can be considered to have converged in terms of load balance if the idle time does not improve in subsequent iterations. The number of accesses to NUMA-remote accesses can increase the task execution time. This can be measured by tracking each memory access. Alternatively, we indirectly measure improvements in data locality in terms of the decrease in the task execution times. When the total time spent executing the tasks stabilizes, we consider the schedule to have converged in terms of NUMA-remote accesses.

For non-iterative applications, we use the strict unordered scheduler to constrain the schedule to the data locality induced by the initialization structure. Similar to the iterative code, the user extracts the schedule from the initialization then calls `applyStUWS(...)` before the spawn of the non-iterative computational kernel.

Table 6.1 shows the problems and configurations for each benchmark. For each application, we used the block size that performed best. The heat benchmark solves the two-dimensional (2D) heat equation and is included with the MIT Cilk package as a test benchmark. The finite-difference time-domain (fdtd) benchmark is a grid-based 2D finite difference time domain method from the PolyBench Benchmark Suite [99]. The floyd-warshall benchmark finds the shortest paths in weighted graphs. While floyd-warshall is akin to matrix-matrix multiplication, the in-place nature and arithmetic operations involved complicate tiling along the lines of matrix multiplication. The correct recursive version of floyd-warshall, similar to the Cilk version for matrix-matrix multiplication, is effectively serial due to the dependencies involved [100]. Therefore, we implemented a version that performs one in-place outer-product update, implemented as 2D recursive loops, per iteration. The

conjugate gradient (cg) benchmark is a sparse numerical solver that uses the conjugate gradient method and was taken from the NAS Parallel Benchmark suite [101], implemented in C and OpenMP [102]. The multi-grid (mg) benchmark is an implementation of the multi-grid numerical method for solving partial differential equations using a hierarchy of calculations at varying resolutions. The pattern of the computation is a V-cycle, where calculations are performed from coarsest to finest then back to coarsest. The benchmark was taken from the NAS Parallel Benchmark suite [101], implemented in C and OpenMP [102]. The parallel prefix benchmark performs a prefix sum on an array of doubles in parallel [103].

For each benchmark, we demonstrate data locality optimization using (a) user-specified work partitioning with constrained work stealing and (b) iterative optimization using RelWS.

The benchmarks can be classified into four groups:

6.4.1 Iterative, Matching Structure

The heat, fdtd, and floyd-warshall benchmarks have a similar structure for initialization and their corresponding kernels, so the template is extracted from the initialization loops and RelWS is used for five iterations until convergence. For the user-specified work partitioning, the programmatically constructed steal tree is used for both phases.

6.4.2 Iterative, Differing Structure

The cg benchmark has a more complex access pattern across phases. Therefore, we start with random work stealing on the kernel and iteratively refine that schedule. For user-specified work partitioning, we programmatically construct multiple steal trees that match each phase.

6.4.3 Non-iterative, Matching Structure

The parallel prefix sum benchmark is not iterative. Hence, we extract the steal tree from the initializer and use it to constrain all of the phases of the parallel prefix kernel, scheduling them with StUWS. For user-specified work

Benchmark	Problem	Configuration	Tasks
heat	nx = ny = 32768	block = 64x8192	2k
floyd-warshall	n = 32768	block = 64x4096	4k
fdtd	ey = ex = hz = 32768	block = 64x8192	2k
cg	NA=2^{21}, NNZ=15	rows = 1024	2k
mg	N{X,Y,Z}=1024,LM=11	block=16x16x4MB	64–4k
scan	N = 256 MB	block = 512	512

Table 6.1: Benchmark configurations (for mg, the number of tasks depends on the level).

partitioning, we programmatically construct a steal tree that is used for all of the phases.

6.4.4 Iterative, Multiple Structures

The V-cycle in the mg benchmark results in phases of several different sizes, corresponding to different grid resolutions. We evaluate three approaches to optimize this benchmark. In the first scheme, we extract the steal tree from one kernel at each grid resolution and use that to iteratively optimize the data locality for all other kernels at the same grid resolution. In the second scheme, we extract the steal tree from the finest grid resolution and use that to iteratively optimize the data locality of all kernels at all grid resolutions using unordered work stealing. In the third scheme, we programmatically construct a steal tree for each grid resolution and use that to constrain execution.

For each benchmark, we implemented two OpenMP schemes: one using parallel-for loops with a static schedule and the other with OpenMP tasks. We found that OpenMP static scheduling performed better than OpenMP dynamic or guided for all of the benchmarks. For the OpenMP tasks, we used recursive tasks, similar to a recursive Cilk implementation. The Cilk first-touch and interleaved curves on each graph are the result of running the baseline Cilk code with either the first-touch or interleaved memory policies enforced by the `numactl` Linux utility. For of the OpenMP task versions, we used the interleaved memory policy because it performed better. For all the constrained work stealing versions and OpenMP static, we used the default first-touch policy.

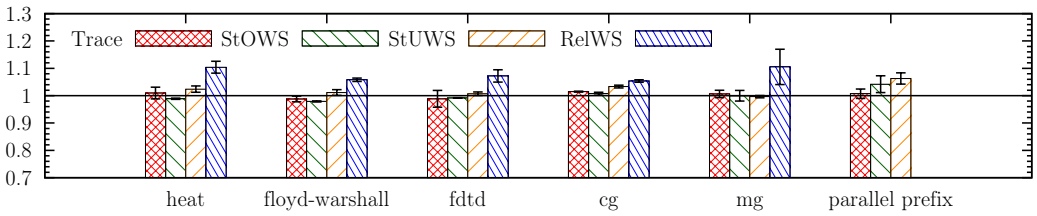


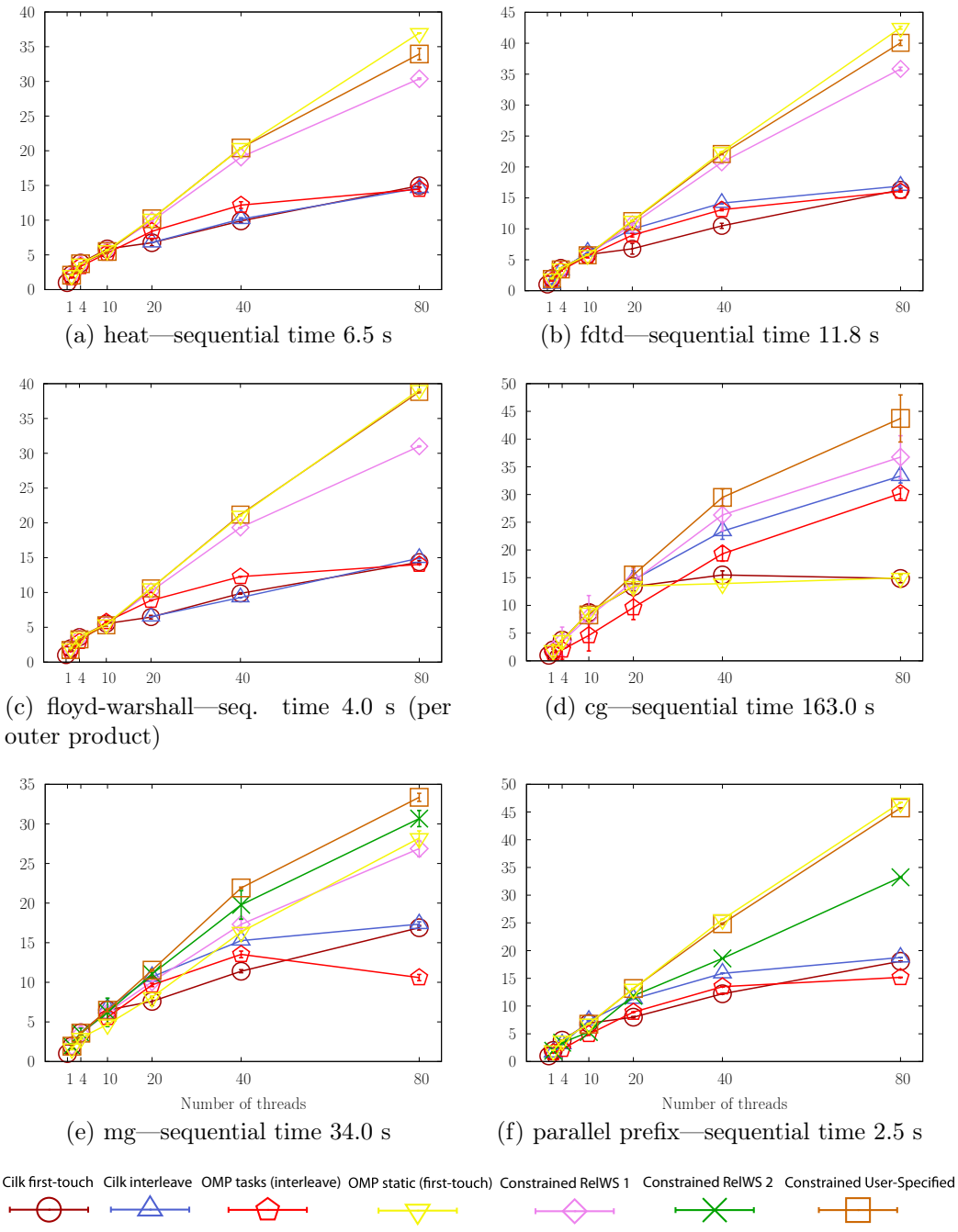
Figure 6.4: Normalized execution time of four configurations (mean({Trace, StOWS, StUWS, RelWS})/mean(Baseline)) compared to the default Cilk scheduler. Error bars are relative standard deviation with a sample size of 5.

6.4.5 Empirical Evaluation

6.4.5.1 Measuring Overheads

We first measure the overhead of tracing and the constrained schedulers. We compare the execution time using a baseline Cilk (MIT Cilk version 5.4.6) to a modified version of Cilk that traces the computation using the steal tree. Figure 6.4 shows the normalized execution time compared to the baseline Cilk without tracing on 80 cores. We also present the normalized execution time for the three types of constrained work stealing. We observe that tracing incurs very low overhead. The heat benchmark incurs the most overhead, about 1.5% with a standard deviation of 0.2%. The strict ordered scheduler, which exactly reproduces the execution, speeds up execution in some cases. For example, the floyd-warshall benchmark has a 2.1% decrease in execution time. The strict unordered scheduler executes any ready task without regard for the original order executed. We expect this may incur some overhead in cases were ordering is important within the composed schedule. The scan benchmark shows the most overhead, about 6.3% with a 2% standard deviation. Finally, RelWS has the most overhead due to following the template schedule and overriding steals. The heat benchmark has the most overhead, incurring 10.4% with a 2.2% deviation. Although the benchmarks exhibit overhead with RelWS, we intend to use it primarily to adapt schedules. Hence, the overhead will be amortized once the adaptation is complete.

Figure 6.5 shows the speedup of all six benchmarks on up to 80 threads. In the speedup plots, we do not include the data redistribution overhead because this cost will be amortized once the schedule converges. The "Constrained Iter. RelWS" label corresponds to the result of using our iterative data locality optimization scheme over five iterations. The "Constrained User-



(a) heat—sequential time 6.5 s

(b) fdtd—sequential time 11.8 s

(c) floyd-warshall—seq. time 4.0 s (per outer product)

(d) cg—sequential time 163.0 s

(e) mg—sequential time 34.0 s

(f) parallel prefix—sequential time 2.5 s

Figure 6.5: Speedup achieved strong scaling to 80 cores with respect to a single thread (legend shown above). Cilk first-touch is the baseline using the default Cilk scheduler with no tracing overhead and first-touch. Compared to Cilk interleaved, OMP schedule(static), OMP tasks on all the loops. Speedup shown for constrained work stealing with a user-specified partitioning, and automatic data locality optimization using RelWS. Each point is the mean of five runs. Error bars are the standard deviation.

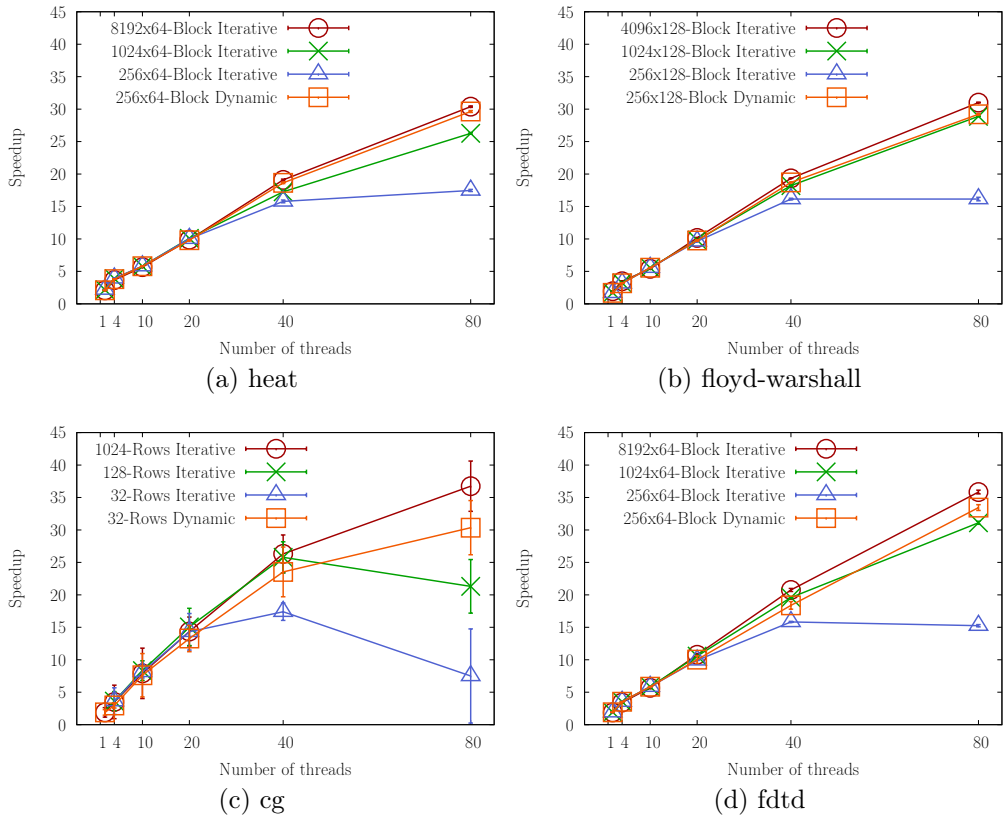


Figure 6.6: Speedup achieved for three grain sizes using relaxed work stealing iteratively. These are compared with using a dynamic grain size, starting with a small user-specified grain size.

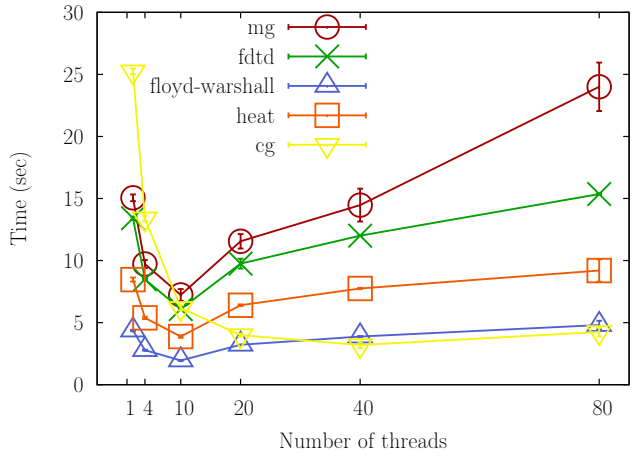


Figure 6.7: Execution time for data redistribution with the fork/join initializer exploiting first-touch. Each point is the mean and standard deviation of five runs.

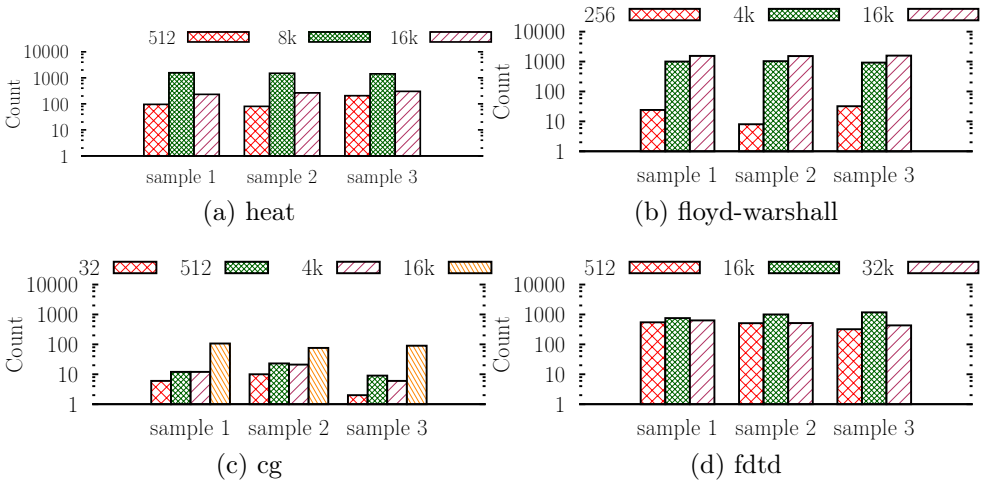


Figure 6.8: Histograms that depict the dynamic grain size distribution after convergence for three samples per benchmark.

Specified" label corresponds to the result of constraining the scheduler using a user-specified partitioning with StUWS. The "Constrained StUWS" label corresponds to the result of extracting the steal tree from one phase followed by StUWS in subsequent phases.

We observe that maximum speedup obtained for any benchmark is about 45x. This is due to the lack of memory bandwidth available within a single NUMA domain. This can be observed by the sub-linear scalability of all the benchmarks up to 10 threads. Beyond 10 threads, which constitutes one NUMA domain, an increase in threads is matched by a corresponding increase in the number of memory controllers, and hence the aggregate memory bandwidth.

We observe that Cilk first-touch, Cilk interleaved, and OpenMP tasks achieve the lowest scalabilites of all the schemes, achieving around a 15x speedup on 80 cores. This is due to the lack of locality awareness in these schedulers. For the cg benchmark, the Cilk interleaved and OpenMP tasks interleaved perform better, achieving more than a 20x speedup, due to the less-regular access patterns in the benchmark.

The constrained user-specified scheme and OpenMP static scheme often perform the best, achieving a speedup up to 46x. This is due to the potential for a perfect match in access patterns across the different phases of the computation. We observe that OpenMP static performs significantly worse for cg and mg, achieving speedups of 10x and 28x, respectively. This is due to the

Benchmark	heat	fdtd	floyd	cg	mg	prefix
KB/Thread RelWS	1.9	1.9	1.75	1.6	1.2	N/A
KB/Thread User-Specified	1.5	1.5	1.5	1.2	5.8	0.6

Table 6.2: Space utilization for Steal Tree.

non-trivial specification required to match the data access pattern across the different phases. The user-specified partitioning scheme can express these complex relationships, consistently achieving high speedups.

The iterative data locality optimization scheme consistently improves upon the baseline Cilk schemes, achieving a speedup of up to 2.5x over the Cilk first-touch scheme. In many cases, it approaches the performance of the user-specified and OpenMP static schemes.

For the mg benchmark, unordered work stealing for all grid resolutions, based on the steal tree from the finest resolution, performs surprisingly well (even better than OpenMP static), achieving a speedup of 30.7x. For the parallel prefix sum benchmark, despite the infeasibility of iterative data locality optimization, we observe a significant speedup (1.8x) compared to OpenMP tasks and the Cilk schemes.

RelWS may cause the steal tree to become more partitioned over time, incurring storage costs for the template schedule that also increase over time. We observe that the size of the steal tree converges quickly, along with the performance. In table 6.2, we show the average amount of memory required to store the steal tree on 80 cores after five iterations. We find that the standard deviation of five runs is negligible. For mg, we present the sum of all the steal trees sizes for each resolution. We also show the amount of memory required to store the user-specified steal tree on 80 cores. The highest amount of memory required is 464 KB total memory for 80 threads when using a user-specified steal tree for mg.

To redistribute the data, we tried explicitly migrating pages using the move_pages system call but found it to be more expensive. Instead, we used the fork/join initializer that copies and reinitializes the data in a constrained manner. Figure 6.7 plots the amount of time taken to execute the fork/join initializer to redistribute the data based on the current template schedule to localize the accesses. We observe that the amount of time taken scales with thread count until we are beyond a single NUMA domain (10 threads).

Benchmark	heat	fdtd	floyd	cg	mg	prefix
Baseline Cilk	123	53	86	570	1319	100
Locality-Optimized	138	68	104	594	1336	117

Table 6.3: Lines of code (using David WheelerâĂŹs SLOCCount) with and without locality optimization API.

Beyond this point, the time taken increases likely due to limitations in memory controller bandwidth or kernel contention in allocating pages in parallel. The time varies based on the amount of data in each benchmark that must be redistributed.

6.4.6 Productivity

To demonstrate the productivity of our approach, we measured the lines of code (using David WheelerâĂŹs SLOCCount) with and without our locality optimization. For each benchmark, the iterative data locality optimization includes API calls for extracting the template schedule, invoking constrained work stealing to schedule the work, and calling the fork/join initializer to redistribute the data. The line counts including the entire source code for each benchmark, are shown in Table 6.3.

We observe that adding our locality optimization only requires around 20 additional lines of code, and this does not increase with the benchmark size.

6.5 Dynamic Task Coarsening

Finding the ideal grain size for a given application is a challenging problem. Selecting a grain size too large will lead to load imbalance, while a small grain size will increase runtime overheads. A coarser grain size also enables the use of efficient sequential implementations as the base case, further improving performance. Ideally, the user could specify the minimum grain size allowed, and the system could adapt that to the largest grain size that maintains a good load balance. We describe a method to automatically select grain size using the algorithms described in this chapter. The programmer selects a small grain size, and the system automatically coarsens it to be sufficiently

large to amortize runtime overheads and improve NUMA locality.

The key observation enabling this optimization is that all parts of the steal tree do not equally contribute to locality and load balance. Steals higher up in the steal tree correspond to large portions of work and fundamentally characterize a schedule. Steals deeper in the tree typically correspond to smaller amounts of work and result from the work stealing scheduler reacting to minor load imbalances. As such, these steals are not fundamental to ensuring data locality or load balance. Even worse, such steals fragment the schedule, interfere with coarse-grained data distribution and work partitioning, and preclude efficient sequential implementations of coarser-grained tasks.

We observe that the load imbalance addressed by these steals deeper in the tree can be addressed by the RelWS scheduler. Thus, we begin with a schedule derived from a random work stealing scheduler to derive a template schedule. However, the lower steals in the steal tree are *pruned* before applying the schedule to subsequent phases. Performing this procedure iteratively, we encourage and retain stealing of coarser units of work, making the steals move higher in the steal tree. We continue this until the schedule does not improve in subsequent iterations. The resulting tree has many more coarse-grained steals than the initial schedule. A code snippet employing this approach follows:

```
#define PRUNE_ITER 5
spawn initialization(); sync;
StealTree t = extractSchedulePrevious();
for (i = 0; i < numIter; i++):
  if (i < PRUNE_ITER):
    pruneTree(t,85); applyRelWS(t); spawn kernel(); sync;
    t = extractSchedulePrevious();
    applyStUWS(t); spawn forkjoin_initializer(); sync;
  else if (i == PRUNE_ITER):
    pruneTree(t,85); applyStUWS(t); spawn kernel(); sync;
    t = extractSchedulePrevious();
  else:
    applyStOWS(t); spawn kernel(); sync;
```

In the algorithm, we call pruneTree(t,85) before invoking the constrained scheduler. This prunes the steal tree in level order, retaining the top 15% of the steal tree. After using the relaxed scheduler for a few iterations, we use the strict unordered scheduler that prunes the steal tree for the last time,

composing a final schedule. After this, we use the final schedule as a template for the strict ordered scheduler.

The `pruneTree()` function includes a parameter p that indicates what percentage of steal points the system should prune from the steal tree. The pruning is implemented by traversing the steal tree nodes in level order and removing the specified percentage of steal points from the bottom of the tree. We achieve this by (a) counting the number of steal points in the steal tree, (b) traversing the steal tree and marking the top $100 - p\%$ steal points as persistent, and (c) deleting all of the non-persistent steal points.

```
void pruneTree(t, p):
  int count = StealTreeNumPoints(t); //total number of steals
  int numPersistant = count * (100−p)/100;
  for (i = 0; i < t.numLevels and numPersistent > 0; i++):
    for (node in t.level): node.persist = true; numPersistent−−;
  for (node in t): if (!node.persist): node.delete();
```

Under strict (ordered or unordered) work stealing, a coarser sequential kernel can be employed for a given task if it is guaranteed that no task transitively spawned by it can be stolen. Each task queries the steal tree to check this condition and appropriately chooses between spawning subtasks and performing a coarser sequential computation. This enables the runtime to dynamically coarsen the tasks and improve performance without impacting load balance.

Figure 6.6 shows the results of applying dynamic coarsening to some of the benchmarks. Due to space limitations, we only show the result for three of the six benchmarks. We plot the speedup obtained using the iterative relaxed method presented previously with three different block sizes. We observe that smaller block sizes perform much worse. The graphs indicate that increasing the block size improves performance. In fact, for our experimental evaluation (featured in the previous section), we used the best-performing block size, which is the largest shown for each benchmark. Increasing the block size beyond this results in reduced performance for the default Cilk schemes and OpenMP tasks due to insufficient parallelism.

We observe that our dynamic coarsening optimization (labeled as "Dynamic" in the figure) performs competitively with the largest, static grain size shown, despite starting with the smallest grain size evaluated. In Figure 6.8, we present three histograms (sampling three different executions) per

benchmark that show the block size distribution resulting from our dynamic algorithm after convergence in five iterations. The histograms for each benchmark are similar, demonstrating that the scheduler converges to about the same dynamic grain sizes each time. For each set of bars, we observe that one set is much smaller than the rest. This indicates that small blocks are used to refine the schedule, while the large blocks provide an initial coarse-grained partitioning.

CHAPTER 7

Dynamic Splicing: Recursive Cache Locality Optimization

Maximizing application performance requires careful orchestration of the execution to best match the given architecture. Manually performing this task is expensive and error-prone due to non-obvious performance implications of source code transformations. The two most common automated approaches consist of compiler analysis and optimization, possibly coupled with automated exploration of the optimization space, or application composition from calls to carefully optimized libraries. Automated approaches can be applied to large programs, but are limited in the class of programs handled, types of transformations considered, and architectural features optimized for. Equally important, such optimizers often generate complicated code that adversely impacts other optimization phases (e.g. register allocation, vectorization). Tuned libraries provide a set of commonly used functions that have been carefully optimized for a given architecture. However, an application composed of library calls might not exploit optimization opportunities across function boundaries.

In general, applications are composed of multiple functions embedded in different libraries, motivating the need for inter-procedural locality analysis and optimization. We exploit data access (effect) annotations to identify the data reuse opportunities. Effect annotations specify the regions of data read and written by a function invocation. Prior efforts have employed such annotations to determine concurrency between sequentially ordered function invocations at the coarsest possible granularity to minimize runtime overheads.

Cache locality optimization requires us to go one step further: schedule the invocations such that the data remains in cache between consecutive uses of a data region across function boundaries.

We dynamically interleave (*splice*) their execution to increase the chances of cache reuse between access to the same data by the leading and trailing functions. The effect information is tracked in the call stack as the functions are executed. When the trailing function attempts an operation that would violate dependences inferred from the effect annotations, it is delayed and enqueued as waiting on the frame in the leading function it depends on. Depending on its effects, some sub-computations in the trailing function's invocation might be delayed while others continue to be interleaved. We attempt to maximally interleave the executions while satisfying the dependences.

Often, recursive programs are parallelized using nested fork/join programming systems. Fork/join programming systems divide a given work into sub-tasks that can be executed concurrently. These programming models underpin several popular approaches to multicore parallelization (e.g., the Cilk family [104, 35, 105], Thread Building Blocks [106], Task Parallel Library [107], OpenMP [2], and X10 [108]). Many common idioms, such as sequential and parallel loops, and data iterators, can be represented as using nested fork/join parallelism. Fork/join programs are typically load balanced using a work-stealing scheduler that balances the load across idle processes. We present an algorithm that further exploits data effects to automatically generate a fork/join program, combining depth-first execution of a nested fork-join program and dependence-driven task-graph scheduler to execute the delayed sub-computations. The entire scheduling strategy co-exists with a work-stealing scheduler.

We implement our approach in the MIT Cilk framework [109] and demonstrate that the effect system, concurrency and locality checks, and interleaved execution can be managed efficiently to accrue cache locality benefits. We also show that the space overheads of delayed execution are low. We evaluate the benefits of dynamic splicing as compared to the state-of-art approaches to optimizing loop programs. These approaches were chosen due to the ability of compiler optimizations to achieve the best performance. Specifically, we demonstrate that our approach can dynamically splice multiple function invocations to achieve performance comparable to optimized programs gener-

ated by Pluto, a polyhedral optimizer for affine loop programs, and Pochoir, a domain-specific language for stencil programs.

The primary contributions of this chapter are:

- Dynamic splicing as an approach to optimizing cache locality.
- Techniques to efficiently track effects and detect interference.
- An efficient scheduler that combines depth-first execution of function invocations, dependence-driven task scheduling for delayed subcomputations, and work-stealing-based load balancing.
- Experimental evaluation demonstrating that dynamic splicing can match performance achieved, in serial and in parallel, by complex compile-time transformations such as diamond tiling.

7.1 Problem Statement

Consider the example program in Figure 7.1a. It involves two operations each copying array A to a different array. This program can be implemented as two calls to the optimized memcpy() routine in the C library[1]. On the system described in Section 7.7, such an implementation involves four array transfers across the memory hierarchy—once for B and C and twice for A. A compile-time optimizer can fuse the two operations to avoid moving array A twice across the memory hierarchy.

While the compile-time techniques improve performance, applying them in the context of recursive programs involves overcoming several challenges. The data accesses in a recursive function invocation might be known only at runtime. Even if it can be approximated at compile-time, effecting such a compiler transformation across function boundaries can be non-trivial task. Finally, compiler transformation assumes that the source code is available for all functions of interest and in a specific form amenable to analysis transformation (e.g., non-linearized indices [110]). In this chapter, we focus on a runtime approach to achieving the benefits of compile-time fusion for recursive programs. For example, we consider a runtime approach to optimize locality for the recursive version of the example program shown in Figure 7.1b.

[1]For simplicity, we will restrict ourselves to the C language and library

```
char A[M], B[M], C[M]
for (i = 0; i < M; i++): B[i] = A[i]
for (i = 0; i < M; i++): C[i] = A[i]
```

(a) A sequence of copy operations

```
copy(A, B, M)
copy(A, C, M)

def copy(A, B, n):
  if n < threshold:
    for (i = 0; i < n; i++): B[i] = A[i]
  else:
    mid = n/2
    copy(A, B, mid)
    copy(A+mid, B+mid, n−mid)
```

(b) Recursive implementation of the copy operation

```
spawn copy(A, B, M)
spawn copy(A, C, M)

def copy(A, B, n):
  if n < threshold:
    for (i = 0; i < n; i++):
      B[i] = A[i]
  else:
    mid = n/2
    spawn copy(A, B, mid)
    spawn copy(A+mid, B+mid,
               n−mid)
```

(c) Fork/join implementation of the copy operation

```
def copy(A, B, n):
  if n < threshold:
    if /*dependences satisfied*/:
      for (i = 0; i < n; i++)
        B[i] = A[i]
    else:
      /*delay step*/
  else:
    mid = n/2
    /*yield to next thread*/
    copy(A, B, mid)
    /*yield to next thread*/
    copy(A+mid, B+mid, n−mid)
```

(d) Illustration of splicing for cache locality

Figure 7.1: Example copy program implemented sequentially (with for-loops), recursively, concurrently with fork/join, and transformed for splicing.

$$
\begin{aligned}
v &\in \mathbb{Z} && [\mathit{Values}]\\
b &\in \{\textbf{true}, \textbf{false}\} && [\mathit{Booleans}]\\
p &\in \{p_1, p_2, \ldots, p_k\} && [\mathit{Parameters}]\\
l &\in \{l_1, l_2, \ldots\} && [\mathit{Locals}]\\
g &\in \{g_1, g_2, \ldots\} && [\mathit{Globals}]
\end{aligned}
$$

$$
\begin{aligned}
e_b \in \mathit{BExprs} &::= f_b(e_1, e_2, \ldots)\\
pr_f \in \mathit{FPragmas} &::= \textbf{\#pragma splice func } \textbf{Reads}(e)\ \textbf{Writes}(e)\\
pr_c \in \mathit{CPragmas} &::= \textbf{\#pragma splice cont } \textbf{Reads}(e)\ \textbf{Writes}(e)\\
pr_s \in \mathit{SPragmas} &::= \textbf{\#pragma splice step } \textbf{Reads}(e)\ \textbf{Writes}(e)\\
e \in \mathit{Exprs} &::= v \mid l \mid g \mid p \mid e_b \mid f_v(e_1, e_2, \ldots)\\
s \in \mathit{Stmts} &::= \textbf{return} \mid s; s \mid l := e \mid g := e\\
&\quad \mid \textbf{if } e_b \textbf{ then } s \textbf{ else } s \mid \textbf{while } (e_b)\ s\\
&\quad \mid pr_c\ f(e_1, e_2, \ldots, e_k)\\
ss \in \mathit{StepStmts} &::= pr_s\ s\\
m \in \mathit{Method} &::= pr_f f(p_1, \ldots, p_k)\ ss\\
pgm \in \mathit{Program} &::= m\ pgm \mid ss
\end{aligned}
$$

Figure 7.2: Language for effect-annotated recursive programs.

Specifically, given a sequential recursive program, we try to answer the following questions:

- How can we efficiently capture dependence and reuse information across function invocations at a fine enough granularity?
- How can we adapt the runtime schedule to exploit the identified data reuse across function invocations?

7.2 Solution Approach

In this section, we present our runtime approach that dynamically *splices* or interleaves recursive function invocations in a sequential program to improve memory hierarchy data reuse. Our approach is based on the following observations:

- Inclusive effect annotations can help compactly capture and track data access and dependences in recursive programs.

```
RangeE:
  void *ptr, *ptr2;
RangeE RE(void* ptr,void* ptr2); /*constructor*/

#pragma splice cont Reads(RE(A,A+M),RE(C+1,C+M)) Writes(RE(C,C+M),RE(D,D+
    M-1))
copy(A, B, M)
#pragma splice cont Reads(RE(C+1,C+M)) Writes(RE(D,D+M-1))
copy(A, C, M)
#pragma splice cont Reads() Writes()
copy(C+1, D, M-1)

#pragma splice func Reads(RE(A,A+n)) Writes(RE(B,B+n))
def copy(A, B, n):
  if n < threshold:
    #pragma splice step Reads(RE(A,A+n)) Writes(RE(B,B+n))
    for (i = 0; i < n; i++)
      B[i] = A[i]
  else:
    mid = n/2
    #pragma splice cont Reads(RE(A+mid,A+n)) Writes(RE(B+mid,A+n))
    copy(A, B, mid)
    #pragma splice cont Reads() Writes()
    copy(A+mid, B+mid, n-mid)
```

Figure 7.3: Illustration of effect annotations. The RE function call returns an effect object.

- The order of execution of work within distinct function invocations can be controlled by interleaving user-level threads executing the function invocations on the *same* hardware thread, similar to coroutines [111].

7.2.1 Effect Annotations

Key to splicing is the tracking of execution in consecutive function invocations to ensure that no dependences are violated. In order to track these dependences, we consider functions written in a recursive form with effect annotations. The language for effect-annotated recursive programs is shown in Figure 7.2. The effect annotations associated with a function definition specify the data regions read or written by an invocation in terms of its formal parameters. A function invocation also specifies the effects of its *continuation*, the remaining computation in the body of the invoking function. An effect-annotated copy function is shown in Figure 7.3.

7.2.2 Spliced Execution using Lightweight Threads

To enable inter-invocation splicing, we must explore the enclosing scope to discover future function invocations that may be spliced. When normal execution encounters a function invocation marked to be spliced with its continuation, we begin spliced execution by creating a new user-level thread, referred to as the trailing thread, and execute the function invocation in one thread and its continuation in another, referred to as the leading and trailing threads. The continuation is explored to reach the next function invocation. Once we have two function invocations, each in its own stack, we execute them in an interleaved fashion.

7.2.3 Maintaining Multiple Call Stacks

During interleaved execution, we maintain the call trees for the two invocations in a symmetric fashion. That is, every function invocation (and return) in one stack is matched with a corresponding invocation (and return) in the other. The sequential yet interleaved execution of user-level threads in the

same hardware thread dynamically produces an effect similar to compile-time fusion of the functions.

7.2.4 Dependence Management

Two consecutive function invocations with only shared read effects can be fully spliced. Opportunities for cache locality optimization also arise between dependent invocations. For example, the last two copy operations in Figure 7.3 can benefit from splicing, even though the array `C` produced by the second statement is used in the third. Naively splicing such invocations can violate dependences and lead to incorrect execution. We check the effects of statements in the trailing thread for interference with the pending continuations in the leading thread. Subtrees of the trailing thread's call tree that might interfere are delayed (placed in the heap) for execution at a later point in time. This allows the runtime to continue spliced execution. When a delayed subcomputation's dependences are satisfied, it is immediately executed to exploit any possible reuse.

Figure 7.1d illustrates the interleaved execution of the `copy` function. On encountering a step (line 4), we check that its dependences are satisfied. If true, we immediately execute the step.

In the rest of the chapter, we address several challenges in making this approach work in practice. The frequent switching between thread contexts can be expensive. The dependence tracking requires additional information on data accesses. Dependence management needs to be optimized to avoid checking for dependences between every step in one task with every step in another. In addition, dependences need to be tracked for any postponed step. Significantly improving cache locality might require spliced execution of several function, further increasing the runtime cost.

7.3 Data Effect Annotations for Recursive Programs

To correctly splice function invocations, the runtime system must be cognizant of the data accesses. The runtime obtains this information in the form of effect annotations on function invocations, their continuations, and steps (statement blocks between function invocations). An effect annotation spec-

ifies the data regions being read and written. An example effect-annotated program is shown in Figure 7.3.

A step's effect annotation specifies the data regions it reads and writes. The effect annotation associated with a function call or a continuation includes all transitive effects encountered during its execution. In other words, the read (write) effects of a function invocation are a subset of the read (write) effects of the invoking function as well as a subset of the effects of the continuations that enclose the call site. A step's effect is specified immediately preceding the statement block. The effect annotation associated with a function definition specifies the effects of a function invocation in terms of the function's formal parameters. Each function call site is annotated with the effects corresponding to its continuation. Continuations with empty effects need to be explicitly specified to be considered for splicing. A continuation with no effect specification will not be considered for splicing.

This inclusive, hierarchical structure of the effect specification enables the runtime to quickly determine whether locality benefits can be accrued and if a step's execution will lead to a conflict. The key requirement that enables efficient runtime check for interference is the specification of continuation effects. After each function invocation, the continuation in the current enclosing function scope must be annotated with the combined effects remaining in that scope. This enables the runtime to determine if a step in a trailing sub-computation will conflict with future accesses encapsulated by the continuation.

While our approach is applicable to a wide range of recursive programs, specifying and managing arbitrary effects can be expensive. Many recursive applications employ coarse base cases to minimize the cost of recursion. This also helps reduce the cost of managing the effects at runtime. The shared state modified by various function calls can often be compactly described—array sections, sub-trees, spatial regions, etc. In general, the most compact description of read/write effects depends greatly on the computation being performed. Therefore, we allow the user to define the effect types and the associated operators. For example, `RangeE` is a user-defined effect type in Figure 7.3. Side-effect-free functions (`RE()` in the example) are used to construct the effect objects.

Any effect type needs to support an interference operator is used at run-

time to ensure dependencies are met while accruing locality benefits[2]. The language specification for expressing an effect-annotated program is shown in Figure 7.2.

7.4 Splicing Scheduler

7.4.1 Initiating Spliced Execution

Splicing is initiated by the runtime through a user API that provides a hint to the runtime that splicing may be appropriate. The TrySplice(int n) API tells the runtime to examine the next n effect-annotated functions to determine if it can splice them and measure whether it may be beneficial for locality. Upon encountering TrySplice the runtime begins unraveling the sequential program in the current scope in a breadth-first manner, instead of following program order. It ensures this execution order is valid as it goes, by checking for interference between the inclusive effects of functions encountered in program order and any steps that may be interleaved. If it encounters interfering steps or unannotated steps or continuations, it cannot safely proceed and does not attempt to further unravel the scope.

Otherwise, it continues until it finds n annotated functions and saves function pointers and packs the corresponding evaluated parameters into structs for future execution. If the user supplies a effect size and intersection operator, the runtime uses a heuristic to determine if splicing may be beneficial by checking the size of the intersection between each function discovered. If the intersection is sufficiently large (a tunable parameter in the runtime), the system uses the stored function pointers and structs to launch user-level threads corresponding to each function that should be spliced. At this point, the splicing scheduler begins interleaved execution, relying on the recursive effects within the distinct function call stacks to ensure correctness. The following pseudo-code snippet shows a simplified sketch of how the top-level splicing scheduler is initiated for the example copy program, assuming that the user inserted a TrySplice(3) before the first copy.

```
ss.funcs = 3; ss.counter = 0 // from TrySplice(3)
```

[2]We shall treat these as polymorphic functions. In the implementation, these are handled in C using name mangling

```
if (ss.funcs != −1):
  f_eff = Reads(RE(A,A+M)) Writes(RE(B,B+M))
  c_eff = Reads(RE(A,A+M),RE(C+1,C+M)) Writes(RE(C,C+M),RE(D,D+M−1))
  ss.ults[ss.counter].{f_eff,c_eff} = {f_eff,c_eff}
  ss.ults[ss.counter++].handle = { copy, struct { A, B, M} }
else: copy(A, B, M)
if (ss.counter == ss.funcs) try_splice_local(ss)
if (ss.funcs != −1):
  f_eff = Reads(RE(A,A+M)) Writes(RE(C,C+M))
  c_eff = Reads(RE(C+1,C+M)) Writes(RE(D,D+M−1))
  ss.ults[ss.counter].{f_eff,c_eff} = {f_eff,c_eff}
  ss.ults[ss.counter++].handle = { copy, struct { A, C, M} }
else: copy(A, C, M)
if (ss.counter == ss.funcs) try_splice_local(ss)
if (ss.funcs != −1):
  f_eff = Reads(RE(C+1,C+1+M−1)) Writes(RE(D,D+M−1))
  c_eff = NULL
  ss.ults[ss.counter].{f_eff,c_eff} = {f_eff,c_eff}
  ss.ults[ss.counter++].handle = { copy, struct { C+1, D, M−1} }
else: copy(C+1, D, M−1)
if (ss.counter == ss.funcs) try_splice_local(ss)
```

A runtime structure ss is used to track the number of functions passed to TrySplice and the how many found so far (ss.counter). As it encounters annotated functions, it saves the required data to postpone function execution. When try_splice_local is called, it uses the heuristic to determine whether splicing will be fruitful, and if not executes the stored functions in program order instead of splicing them.

The following section focuses on the algorithms for maintaining correctness and efficiency while interleaving function call stacks that the scheduler has decided to splice.

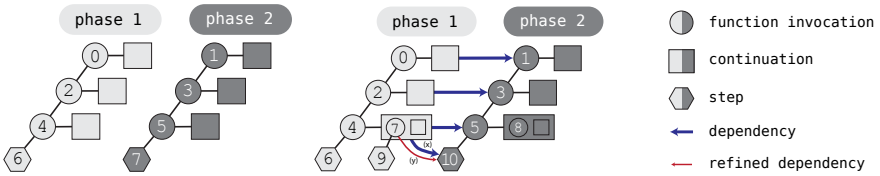
7.4.2 Interleaving Functions' Execution

At compile time, the splicing scheduler uses the effect annotations to embed calls to construct and manage the data effects. It inserts hooks to manage dependences and yield to other spliced functions. It also uses live variable analysis and a continuation-passing transform to enable delaying steps that cannot be executed immediately.

The following is pseudo-code of the copy program after the splicing compiler pass is applied:

```
def copy(A, B, n):
  if n < threshold:
    s_eff = Reads(RE(A,A+n)) Writes(RE(B,B+n))
    canExecute = splice_step(ss.cur_ult, s_eff)
    try_context_switch_ult((ss.cur_ult+1) % ss.funcs);
    if canExecute:
      for (i = 0; i < n; i++):
        B[i] = A[i]
    else:
      /* save stack frame and live vars to heap, and unwind */
  else:
    mid = n/2
    if (ss.inSplicedMode):
      f_eff = Reads(RE(A,A+mid)) Writes(RE(B,B+mid))
      c_eff = Reads(RE(A+mid,A+n)) Writes(RE(B+mid,B+n))
      splice_func_cont(ss.cur_ult, f_eff, c_eff)
      try_context_switch_ult((ss.cur_ult+1) % ss.funcs);
    copy(A, B, n)
    if (ss.inSplicedMode):
      f_eff = Reads(RE(A+mid,A+n)) Writes(RE(B+mid,B+n))
      c_eff = NULL
      splice_func_cont(ss.cur_ult, f_eff, c_eff)
      try_context_switch_ult((ss.cur_ult+1) % ss.funcs);
    copy(A+mid, B+mid, n−mid)
```

On encountering a step or function pragma, the compiler emits code to create the effect and pass it the appropriate runtime function: **splice_step** or **splice_func_cont**. These functions store the effect in the corresponding stack frame and perform the required interference checks. A subsequent yield call is inserted (try_context_switch_ult()) to pass control to the next spliced thread, effectively interleaving the user-level threads spliced together. The context switch function tracks the depth of each user-level thread to keep the stacks executing together as much as possible. A thread-local variable ss is used to track the current user-level thread ss.cur_ult and the number of functions spliced together ss.funcs. When a step is encountered, interference checks determine if the step can execute immediately or must be delayed for future execution until all dependencies are satisfied.



(a) Matching dependency (b) Offset dependency

Figure 7.4: Illustration of two phases (function invocations) spliced together, comparing (a) matching dependency with no delayed steps and (b) an offset dependency, which requires steps to be delayed.

7.4.2.1 Illustration

Figure 7.4 depicts a snapshot of two spliced phases of a recursive program. Each circle represents a recursive function invocation, a square represents the continuation following the invocation, and each hexagon represents a step, a block with no interleaved spliceable functions. The numbers on each shape denote the logical time at which the phases are executed when spliced together. For each function invocation in a trailing phase, we aggressively execute the function until a sequential block is encountered that will conflict. Figure 7.4a shows the resulting execution order when the phases have a matched dependence structure: i.e., every sequential block in the second/-trailing phase depends on the corresponding sequential block in the first/leading phase. In this case, as soon as a block finishes in the first invocation, we execute the corresponding block in the second invocation. In Figure 7.4b, the sequential block accesses are offset in the trailing phase, i.e., the i-th leaf in the trailing phase accesses the data produced by the $(i+1)$-th leaf in the leading phase. Here, each trailing function invocation depends on the continuation at the same level in the leading phase as indicated by the blue arrows in the figure. The spawn operations leading to the first leaf are perfectly interleaved between the two phases, as shown by the alternating logical time steps. However, the leaf in the trailing phase is delayed as it depends on the continuation of the invocation at time step 4, shown by the blue array labeled (x). When subsequent execution expands this continuation, the (x) dependence is refined to the invocation at time step 7, shown by the red arrow (y). After this dependency is satisfied at time step 9, the delayed step is immediately executed at time step 10 to reap locality benefits. Note that although the first leaf in the trailing phase is delayed, the call stacks are

expanded in an interleaved fashion in time steps 7 and 8.

7.4.2.2 Dependence Checks

Before providing the detailed algorithms, we describe the key notions of interference and checking dependences. A step, function invocation, or a continuation is said to interfere with effect if their effects share a data region (non-null intersection) and one of the shared effects is a write. Interfering operations need to be executed in the specified program order to preserve dependences. In particular, when splicing two phases, a step in the trailing phase can be executed only if it does not interfere with the remaining execution to be performed by the leading phase. In the case of any interference, this step needs to be delayed until any computation it interferes with in the leading phase has been executed. In a recursive program, the remaining execution in a phase is captured by the continuations in the call stack. Therefore, interference of a step is checked with this stack of continuations. A trailing phase step interfering with multiple leading phase continuations is marked as being dependent on the oldest such continuation. Such a continuation would be executed last and thus represents the "boundary" for that interference.

We now expand this high-level description of dependence management and explain the runtime actions taken by the scheduler corresponding to the hooks introduced in the preceding discussion.

7.4.2.3 Encountering a Spliced Function Invocation.

Each continuation tracks the steps in the trailing phase that depend on it. The corresponding actions are taken by the **splice_func_cont** function and are shown in the Algorithm 3. When a non-null continuation is expanded, a function invocation may be discovered along with a follow-on continuation. To refine the dependencies, we inspect all the dependent steps (stored in p.deps) to check if each step in p.deps solely depends on the the newly discovered function. If they are confined to that function, we refine the dependency to eagerly release it when the function is finished executing. In Algorithm 3, p is the parent continuation that was expanded, and the refinement occurs on line 8.

Algorithm 3: Actions taken when encountering a spliced function invocation. $||$ and $\not||$ denote non-interference and interference, respectively.

```
   Input: t : current user-level thread running
1          s : effect specification for function and continuation
2splice_func_cont(t, s_func, s_cont);
3begin
4   | d = t.cur_stack_depth;
5   | p = t.stack[d − 1];
6   | t.stack[d].func ← s_func;
7   | t.stack[d].cont ← s_cont;
8   | s.deps ← { step | step ∈ p.deps, step || s_cont };
9   | p.deps ← { step | step ∈ p.deps, step ⊮ s_cont };
```

7.4.2.4 Encountering a Step

The actions taken on a spliced step are shown in Algorithm 4. When encountering a step, we check for interference with preceding threads by calling **find_add_dependencies** in Algorithm 4 on line 5. If the step has no dependencies, it is executed immediately. Otherwise, we delay the step's execution by storing the current state (live variables) in a heap-allocated object and wait for its dependencies to be released.

7.4.2.5 Managing Dependencies for a Step

Step dependencies are managed by checking for continuations in preceding spliced threads that interfere with a step (shown in Algorithm 5). The leading thread—or first thread being spliced—does not need to check for interference since it is executed in program order. However, all trailing threads must check their interference with preceding threads to ensure they can execute without violating a data dependency. The delayed steps in the heap are structured as a task graph, with each step tracking the number of incoming dependencies and the steps that depend on it (outgoing dependencies). A delayed step is considered ready when all its incoming dependences have been satisfied. When such a step is executed it notifies all its outgoing dependencies. These actions consist of three major non-interference checks required to ensure correctness.

First, the current step checks for non-interference with future work in any preceding thread. Because the set of continuations compactly captures fu-

ture work in a thread, we check all continuations in all preceding threads for interference. This check is shown in Algorithm 5 on line 8. Second, any delayed step must be checked for dependence on previously delayed steps in preceding threads (shown on line 11). In Section 7.6, we describe an optimization to find potentially interfering past delayed steps efficiently. Third, if the above two checks detect a dependence for the given step, then the step cannot be executed at this point in time. In this case, the step is delayed and stored in the heap, and steps in trailing threads that depend on it are marked to be notified. We perform this operation by incrementing the dependent steps' join counter and tracking then in this step's deps field, as shown in Algorithm 5 on line 17.

Algorithm 4: Actions taken when encountering a step during spliced execution.

```
   Input: t : current user-level thread running
1          s : the encountered step
2splice_step(t, s);
3begin
4   | d = t.cur_stack_depth;
5   | find_add_dependencies(s, t, d);
6   | if s.join_counter == 0 /*has no dependencies*/ then
7   |     | execute s;
8   | else
9   |     | delay step execution and place it in the heap;
```

7.4.2.6 Releasing Dependencies

When a stack frame s completes execution and is ready to be destroyed, all steps tracked as being dependent on it (s.deps) are released by decrementing their join counter. If any of these steps have no more dependences, they are immediately executed. This can cause more frames to be destroyed, enabling more steps. All such transitively enabled steps are executed immediately (shown in Algorithm 6).

Algorithm 5: Dependency checks for steps.

```
Input: s : step to be checked for interference
       t : current user-level thread running
       d : current stack depth
find_add_dependencies(s, t, d);
begin
    for u ← 1 to t − 1 do
        for l ← 0 to d do
            if s ⊬ u.stack[l].cont then
                u.stack[l].cont.deps+ = {s};
                atomic s.join_counter+ = 1;
                break;
        foreach { p | p ∈ u.delayed, p ⊬ s } do
            p.deps+ = s;
            s.join_counter + +;
    if s.join_counter ≠ 0 /*has dependency*/ then
        for u ← t + 1 to number of spliced threads do
            foreach { p | p ∈ u.delayed, p ⊬ s_step } do
                s.deps+ = {p};
                atomic p.join_counter+ = 1;
```

Algorithm 6: Actions taken when a stack frame is destroyed.

```
Input: t : current user-level thread running
       s : the stack frame being destroyed
@stackframe_destroy(t, s);
begin
    foreach { x | x ∈ s.deps } do
        decrement join counter for x;
        if x has no dependencies then
            execute x;
            recursively decrement join counter, and execute enabled steps that
            depend on x;
```

7.5 Splicing in Parallel

In this section, we describe how the data effect system and splicing approach can be utilized to generate a nested, parallel programs scheduled with work stealing.

Recursive programs can be translated into nested fork/join programs using a few simple keywords. The example copy program parallelized using Cilk primitives is shown in Figure 7.1c. The spawn keyword notes that the function invocation can be processed concurrent with the code that follows. The sync keyword signifies a dependence between one of more of the spawned functions that precede the sync and any statements that follows it. In other words, the sync statement acts as an ordering constraint, stating that the computation past the sync cannot begin until all computation preceding the sync in the task is complete. The two keywords—spawn and sync—or their variants constitute the key components of a large class of nested fork/join programs [35, 112, 113]. This specific spawn-sync model, together with the work-stealing scheduler, is used in the Cilk Plus C/C++ language extensions, now widely available in several mainstream compilers: Intel ICC, GNU GCC, and Clang. This scheduling policy provides provably-good space and time bounds [114] within a constant factor.

In the parallel context, a *step* is a sequence of instructions with no interleaving spawn or sync. Each step is executed by exactly one thread and can not be migrated once it begins execution. Each spawn has an associated level, also referred to as its stack depth. The initial task has a level of 0. A task's level is defined as one greater than the level of the task that spawned it. The program is executed using a work-stealing scheduler. One worker thread begins execution with one task and other worker threads begin execution in an idle state. An idle worker enters the *stealing phase* and attempts to steal work from a randomly chosen victim, repeating this process until it finds work. On finding work, a worker begins a *working phase*, which ends once its local double-ended queue (deque) of tasks is empty.

Note that these worker threads are often operating system threads, one per hardware core, and are meant to exploit all available hardware parallelism. We discuss the splicing of task invocations using lightweight (user-level) threads on top of the hardware threads employed by the work-stealing scheduler.

7.5.1 Transforming the Recursive Program to Cilk

The inclusive effect system described in the context of a recursive program lends itself naturally to automatically building a nested, fork/join program. Because the inclusive effect for a function and continuation are both present, we can determine at runtime if a sync is required between the function and following continuation. Thus, we eagerly insert a spawn for every function/-continuation pair that has a data effect present and conditionally insert a sync.

We perform the transformation by inserting a spawn for every function and continuation that is exposed with effect pragmas. We insert a spawn before the function call, and then immediately after the function call generate code to check if the function's effect interferes with the continuation's effect. If it does at runtime, we execute a sync, inhibiting parallel execution past this point. In this way, we ensure that the transformed concurrent program executes correctly in parallel. The following is the second half of the example copy program with automatically inserted spawns and syncs:

```
mid = n/2
{
  f_eff = Reads(RE(A,A+mid)) Writes(RE(B,B+mid))
  c_eff = Reads(RE(A+mid,A+n)) Writes(RE(B+mid,B+n))
  if (ss.inSplicedMode):
    splice_func_cont(ss.cur_ult, f_eff, c_eff)
    try_context_switch_ult((ss.cur_ult+1) % ss.funcs);
  spawn copy(A, B, n)
  if (effects_interfere(f_eff, c_eff)): sync
}
{
  f_eff = Reads(RE(A+mid,A+n)) Writes(RE(B+mid,B+n))
  c_eff = NULL
  if (ss.inSplicedMode):
    splice_func_cont(ss.cur_ult, f_eff, c_eff)
    try_context_switch_ult((ss.cur_ult+1) % ss.funcs);
  spawn copy(A+mid, B+mid, n−mid)
  if (effects_interfere(f_eff, c_eff)): sync
}
```

7.5.2 Tracking Global Dependencies

The dependency structure as discussed so far only considers local dependencies, which is sufficient when the program is interleaved sequentially. However, in the presence of steals, global dependencies must be maintained between working phases to ensure correct execution. In a Cilk program, work is moved between threads during a stealing phase when a thief successfully steals a continuation from a victim that has a task on the remote end of the deque. We have modified the Cilk stealing protocol during spliced execution to steal a set of symmetric continuations from all the spliced user-level threads which are stored as a n-tuple in the victim's deque.

A thief must determine which global conflicts may exist when a steal occurs so it does not execute interfering steps across user-level threads. The effects that may interfere are the effects at the spawn where the steal occurs for each spliced thread. Thus, the thief copies the effect set at that spawn for each user-level thread and checks for interference with the spawn's effect before executing any step. Because the stolen continuation is still present in the victim's stack, the victim will not execute any steps that may conflict with it (they will be dependent on a continuation that is never locally expanded). Thus, a steal during splice execution does not impose extra synchronization on the victim, beyond traditional work stealing. When future thieves successfully steal from the victim, they add to their global conflict list the set of previously stolen continuations from the victim's working phase to ensure non-interference between the previous thieves. Using this protocol, the global conflict set is propagated between distinct working phases as steals occur.

Algorithm 7: Actions taken on global tier when a steal occurs.

```
Input: v : the victim of the steal operation
       t : the thief stealing from v
       d : the depth at which the steal occurs
@steal(v, t, d) begin
    for i ← 1 to number of spliced threads do
        t.global_tier+ = i.stack[d].func v.global_tier+ = i.stack[d].cont
```

The actions described in Algorithm 7 are taken when a steal occurs, and the collection of effects in the global tier are copied to a local working phase when it starts. During this working phase, instead of only checking for local

dependencies (shown in Algorithm 5 on line 3), each step is checked with the global tier before the local. If it conflicts globally, the step will be delayed until the original synchronization point. Once the leading thread has finished execution up to the original synchronization point, any delayed steps and continuations due to global dependencies are released for each trailing thread in turn, starting with the first, until each trailing thread completely finishes execution up to the synchronization point.

7.6 Additional Optimizations

Thus far, we have presented the key aspects of effect-directed runtime scheduling and load balancing to optimize for locality. In this section, we discuss further optimizations employed and constraints imposed in our implementation to lower overheads and make the approach useful in practice.

7.6.1 Splicing Multiple Threads

At the top-level when splicing is initiated by the runtime, often multiple function invocations will be spliced to maximize cache locality. When multiple function invocations are spliced using distinct user-level threads, the threads are ordered based on the function invocation order. Each thread's execution needs to check for interference with all preceding threads to preserve dependences. This can lead to quadratic dependence checking costs. Where possible, we reduce this overhead by exploiting transitive dependencies between preceding threads. For example, if an effect is equivalent or a subset of a preceding thread's effect, it is sufficient to wait until that preceding thread's effect is ready to execute, instead of searching and waiting on all dependencies in every preceding thread. We discover transitive dependencies by searching for equivalent or subset effects in preceding threads at the same level of the stack. If we find such an effect, we wait for that effect to be ready to execute instead of searching all preceding threads. For benchmarks that iteratively execute and often have a symmetric effect pattern, we find that this optimization significantly decreases the overhead of tracking dependencies. For this optimization to be available to the runtime, the user must supply an equivalence-subset operator for effects.

7.6.2 Fast Dependence Searches

Whenever we encounter a step in a trailing thread, we must check if that step interferes with each preceding thread. Transitive effects may reduce this cost to effectively checking a single preceding thread. However, performing this check naively requires checking that the step does not interfere with all the continuations on the stack. We reduce the cost of this check by incrementally checking for interference each time we encounter a function invocation during execution, instead of waiting until we encounter a step. When we encounter a function invocation in a trailing thread, we store the deepest level of each preceding thread's continuation for which it does not interfere. Due to the inclusiveness of the effect system, we can start at the deepest level stored with the parent function invocation when computing interference for the child. This incremental computation greatly reduces the number of interference checks required. This optimization is shown in Algorithm 8 and can be augmented to the basic Algorithm 3.

Algorithm 8: Fast dependency checks by incrementally checking for continuation interference.

```
1 for u ← 1 to t − 1 do
2 |   t.level[u] ← p.level[u];
3 |   for i ← p.level[u] + 1 to d do
4 |   |   if s_func || u.stack[i].cont then
5 |   |   |   t.level[u] ← i;
6 |   |   else
7 |   |   |   break;
```

The second dependence optimization we implement is for locating delayed work in preceding threads that may interfere. Instead of searching through a list of delayed steps for a given thread, we recursively walk the tree of live stack frames that lead to the steps; any live delayed step must be reachable from the root where splicing started. We search all paths from the root recursively, eliminating paths where the effect at that subtree does not interfere with the step being checked. This is valid due to the inclusive nature of the effect system. We find that this optimization increases performance substantially compared to the naive approach of searching through lists of delayed steps.

7.6.3 Step Pipelining

An executable step in the computation may encompass a large amount of sequential work that may benefit from being interleaved with other steps in other threads. Because we are not modifying or analyzing the sequential code in a step, we can not interleave it with other steps without help from the user. To enable step interleaving, we allow the user to write a slice operator for an effect, that constructs a pair of partial effects that can be executed distinctly. For this optimization to be applied, the user must also write a parametric variant of the step that takes as input an effect and performs the corresponding computation. When the runtime encounters a parametrically defined step with a sliceable effect, it invokes the slice operator on the effect, and applies the first partial effect. It then context switches to the next user-level thread and if there is a matching parametric step the runtime slices it and only executes the partial effect immediately if there is no interference. This pattern continues for all the threads, enabling a tight interleaving of matching steps across user-level threads. The system iteratively calls the slice operator until slicing is not possible or the effect is empty. By defining the slicing operator, the user controls the granularity of the resulting partial step that is executed by the system.

Algorithm 9: Pipelining of parametric steps with slicing (this operation is performed at line 6 of Algorithm 4).

```
1  if s is parametric ∧ s_step is sliceable then
2      s_next ← s_step;
3      repeat
4          (s_cur, s_next) ← slice(s_next);
5          if s_cur does not interfere then
6              parametric_fn(s_step)(s_cur);
7              context switch to next ult;
8          else
9              delay s_cur ∧ s_next to the heap;
10             context switch to next ult;
11             break;
12     until s_next = ∅;
```

7.7 Experimental Evaluation

All experiments were executed on an eight-socket system with 1 TB DRAM, comprised of eight 10-core 2.27 GHz Intel E7-8860 processors. Our splicing scheduler is implemented in MIT Cilk 5.4.6 and all generated codes compiled with GCC 5.2.0. To comparatively evaluate our splicing scheduler, we chose MIT Cilk and two compiler frameworks, Pochoir [50] and Pluto [115]. Pluto and Pochoir have been shown to generate highly-optimized implementations based on extensive prior research [116, 50]. Thus, we selected benchmarks that are statically analyzable, regular, and suitable for optimization by these compiler suites.

7.7.1 Benchmarks Evaluated

The benchmarks and configurations evaluated are shown in Table 7.3; these were taken from the Polybench [99] and Pochoir benchmark suites. Each benchmark has multiple phases that can benefit from data reuse. Four of the benchmarks are stencils (Jacobi-2D, FDTD-2D, Seidel-2D, APOP) that have significant data reuse potential between multiple iterations (often referred to as time tiling). The other benchmarks, MVT and BICG, have two distinct phases that can benefit from interleaved execution. We evaluated the four stencil benchmarks with both the Pochoir and Pluto compilers. The MVT and BICG benchmarks cannot be written in Pochoir, because it exclusively expresses stencil patterns. For the stencil benchmarks, we selected timestep counts that resulted in the best performance for each scheduler evaluated.

7.7.2 Evaluation with Pluto

Pluto is a polyhedral source-to-source compiler transformation framework for affine loop nests that optimizes locality and automatically parallelizes them. Pluto uses deep, static dependency analysis to time-tile and parallelize affine loops nests by generating C code annotated with OpenMP pragmas. For applicable loops, it has been shown to generate highly-optimized parallel implementations that rival other locality optimizing frameworks and the best production compilers. To evaluate our runtime approach, we compared against the latest version of Pluto [115] that uses *diamond* tiling (on

the pet branch). For the codes tested, the Pluto generated code naturally achieves good NUMA locality, due to the matched structure of the generated for loops. We compiled an analyzable for-loop version of all the codes in the Pluto transformation framework with `--parallelize`, `--lbtile`, and `--tile`. These options are mentioned as obtaining the best performance in a recent work by Pluto's authors [117]. For all the benchmarks, we tested several tile sizes (for L1 and L2 cache) to find the best performing configurations; the tiles we selected for Pluto are shown in Table 7.3.

7.7.3 Evaluation with Pochoir

Pochoir [50] is a compiler and runtime framework that transforms a stencil-specific DSL embedded in C++ code to a concurrent multicore execution using Cilk. The underlying Pochoir algorithm creates a cache-oblivious parallelization of time-stepped multidimensional grids using "hyperspace cuts", an asymptotic improvement over trapezoidal decompositions. Thus, Pochoir does not require any user input on tiling or NUMA placement. We evaluated a version of each benchmark written in Pochoir, backed with the Intel ICPC compiler, version 14.0.3, due to the limitations in compiling Pochoir-generated code using GCC[3]. We used the following flags for compiling the Pochoir codes: `-O3 -funroll-loops -fno-alias -fno-fnalias -fp-model precise`.

7.7.4 Evaluation with Splicing Scheduler: Sp

We implemented each benchmark in recursive form with the data effect annotations. For the stencils, we provided a hint to the splicing scheduler at the top-level timestepping loop to attempt splicing the subsequent T_s timesteps (an tunable parameter). For each stencil, we evaluated several block sizes and values for T_s and found the best performing block and time-tile sizes, presented in Table 7.3. To obtain the best performance for the stencil benchmarks, we implemented an parametric variant of the kernel that executes a partial effect, allowing the runtime to pipeline the kernel's execution. We

[3]We compiled Pochoir programs on a system with Intel compilers (but fewer cores) and executed the binary on this system.

found that without this optimization, the very small block sizes required to obtain cache reuse at high levels of the cache (L1/L2) were prohibitively expensive due to runtime effect management and Cilk overheads. For the non-stencil benchmarks (MVT and BICG), we provided a similar hint immediately preceding the two phases of each benchmark to the splicing scheduler.

7.7.5 Evaluation with Cilk: Cilk

For each benchmark, we evaluate the version equivalent to the Cilk code generated from the effect system. All effect-related actions are removed and the code is executed as a conventional Cilk program.

7.7.6 Evaluation with NUMA-Optimized Cilk: CilkN

Cilk, and the underlying random work-stealing scheduling algorithm, has been shown in past work to cause performance degradations in the NUMA contexts when the data accessed in a task is not executed where the page is mapped by the system. For instance, in the first-touch policy, if the data initialization, and thus page allocation, occurs in a different memory domain than use, the task time may be inflated. By using a random work-stealing scheduler, the locale of page allocation and subsequent use is often mismatched. Hence, our splicing algorithm when used in conjunction with random work stealing suffers of these effects. In a work by Lifflander et al. [118], the authors demonstrate a variant of Cilk that is augmented with NUMA optimizations that performs comparable to OpenMP for stencil-like benchmarks. We consider this version for evaluation as well.

7.7.7 NUMA-Optimized Splicing Scheduler: SpN

To reap the NUMA benefits from CilkN, we have built a version of our splicing scheduler on top of their scheduling algorithm, *relaxed work stealing*.

7.7.8 Experimental Speedup Comparison

In Figure 7.5, we compare the speedup achieved by the various scheduling and locality optimizers for the benchmarks evaluated. The speedup is calculated relative to serial execution time scaled by the number of cores. Each benchmark was written with standard for-loops and compiled with GCC 5.2.0 using `-O3`; serial timings are shown in Table 7.3. Except for Pochoir, all the generated codes were compiled with the same version of GCC. Each data point on the graphs is the arithmetic mean of five executions and the error bars show the standard deviation.

On the speedup graphs, the CILK bars represent the speedup achieved with the working-stealing Cilk scheduler. The scaling is limited by memory bandwidth and NUMA locality for these benchmarks. The CILKN bars show the speedup achieved by incorporating NUMA optimizations [118]. At sixteen cores, beyond a single memory domain, the scaling significantly improves with CILKN.

The SPN and SP bars show the speedup obtained using the spliced scheduler with and without NUMA optimization, respectively, by interleaving T_s timesteps (shown in Table 7.3). Across all benchmarks and scales, the splicing scheduler significantly outperforms native Cilk code. Compared to CILK, SP performs around 2×–3× better overall. To summarize the relative improvement, Table 7.1 presents the mean speedup over all the benchmarks, delineated by NUMA regime. By incorporating NUMA optimizations, performance is further improved beyond eight cores. Compared to CILK beyond eight cores, SPN executes 3.84× faster (arithmetic mean over the benchmarks), shown in Table 7.1. Compared to the NUMA-optimized Cilk, CILKN executes 2.42× faster beyond eight cores.

Across the benchmarks, our splicing scheduler is competitive with Pluto and Pochoir. Pluto and Pochoir both extract dependencies at compile-time either through static polyhedral analysis or expressed in the DSL. Although we pay the runtime cost of managing dependencies, context switching between user-level threads, and postponing tasks, we still improve performance of the native Cilk program significantly! In general, Pluto and Pochoir extract reuse at all levels of cache, including the L1 cache. Our scheduling mechanism is not able to optimize as well for L1 cache reuse, due to the overheads of executing a user-defined kernel. Without compile-time trans-

formations, the cost of a function call for each kernel limits the reuse that is possible without incurring prohibitive overheads. For the one-dimensional APOP benchmark in particular, Pluto performs better due to tight fusion of the kernel that is not limited by complex boundary conditions.

The MVT and BICG benchmarks involve a reduction-like operation from a matrix to a vector. Thus, Pluto is not able to generate an efficient parallel implementation in OpenMP because it does not detect and optimize for the reduction-like pattern. The effects of this limitation can be observed in Figure 7.5e and 7.5f: the performance decreases with scale. For the recursive versions we wrote, we used an accumulator to combine results into the vector. Thus, the two phases in MVT and BICG can be spliced without delaying work and decreasing concurrency. This implementation accrues locality benefits and scales much further than Pluto.

7.7.9 Hardware Performance Counters

To further explain the behavior of the splicing scheduler, we collected cache and TLB miss rates and instruction counts using hardware counters accessed through the Linux kernel API provided with `perf`. These results are presented in Table 7.5 for all the benchmarks. The instruction counts are higher for all locality optimized codes compared to the serial implementation. Pluto increases instruction count due to the conditionals and min/max operations it performs for tiling and parallelization. Pochoir's runtime performs complex hyperspace cuts, which significantly increases the instructions executed. Compared to Cilk and serial miss rates, the splicing scheduler reduces L2 and L3 misses, which correlate strongly with the performance improvement. The spliced version reduces the L2 miss rate by 3.7× compared to Cilk. Of the L3 accesses, the spliced version has 2× fewer misses compared to Cilk. The TLB accesses are reduced by 9× compared to Cilk, although the percent of TLB misses increases significantly. These factors demonstrate that the splicing performance improvements are correlated with better memory hierarchy reuse.

Domain	Arithmetic Mean				
	SpN	CilkN	Pluto	Pochoir	Cilk
Cores 1-8 (within NUMA)	1.01	2.00	0.93	1.17	1.99
Cores 16-64 (across NUMA)	0.70	1.69	0.66	0.58	2.89
Domain	**Geometric Mean**				
	SpN	CilkN	Pluto	Pochoir	Cilk
Cores 1-8 (within NUMA)	1.01	2.14	0.95	1.21	2.13
Cores 16-64 (across NUMA)	0.73	1.96	0.84	0.60	3.24

Table 7.1: Overview of trends across all benchmarks. Each number is the arithmetic or geometric mean of execution time ratios for all benchmarks between x, the column header, and the splicing (Sp) scheduler. For example, the first column is SpN/Sp.

7.7.10 Overheads

In Table 7.4 on the left, we present statistics on the splicing scheduler's operations to concurrently interleave while ensuring correctness on 64 cores. For Jacobi-2D, Seidel-2D, and FDTD-2D, the system performs tens of millions interference checks to splice the program and execute it concurrently. Because APOP is one-dimensional with simple boundary conditions, it requires fewer interference checks. The two phases spliced for MVT and BICG are effectively concurrent, and thus do not pause steps or have any dependencies. Overall, a large number of runtime operations are required to extract cache locality and parallelism. Thus, efficient effect description and runtime design are required to obtain high performance.

In Table 7.4 on the right, we present an upper bound on the amount of extra space required to store delayed steps during spliced execution. In general, the number of extra stack frames allocated will be proportional to the number of iterations spliced together and the number of conflicts. Beyond this, the size of each delayed step corresponds to the amount of state (live variables) on the stack that must be saved on the heap to be executed when ready. For the benchmark evaluated, we find that the space overheads due to delayed steps are low: the maximum extra space required is 16MB on 64 cores for the Jacobi-2D benchmark. MVT and BICG do not require extra space because the spliced phases are concurrent and thus never delayed.

Domain	Arithmetic Mean				
	Sp	**CilkN**	**Pluto**	**Pochoir**	**Cilk**
Cores 1-8 (within NUMA)	0.98	1.99	0.92	1.16	1.97
Cores 16-64 (across NUMA)	1.32	2.42	0.94	0.83	3.84
Domain	**Geometric Mean**				
	Sp	**CilkN**	**Pluto**	**Pochoir**	**Cilk**
Cores 1-8 (within NUMA)	0.99	2.12	0.94	1.20	2.11
Cores 16-64 (across NUMA)	1.38	2.69	1.15	0.85	4.46

Table 7.2: Overview of trends across all benchmarks with respect to SpN. Each number is the arithmetic or geometric mean of execution time ratios for all benchmarks between x, the column header, and the NUMA-optimized splicing (SpN) scheduler. For example, the first column is Sp/SpN.

Benchmark	**Problem**	**Cilk, CilkN, Sp, SpN**			**Pluto**	
		T_s	Block	Pipelining	Block	Timesteps
Jacobi-2D	32768^2	16	$1024 \cdot 4096$	Yes	L1=$128 \cdot 128 \cdot 128$, L2=$8 \cdot 8 \cdot 8$	128
Seidel-2D	32768^2	8	$512 \cdot 8192$	Yes	L1=$128 \cdot 128 \cdot 128$, L2=$8 \cdot 8 \cdot 8$	128
FDTD-2D	16384^2	8	$512 \cdot 4096$	Yes	L1=$128 \cdot 128 \cdot 128$, L2=$8 \cdot 8 \cdot 8$	128
APOP	268435456	16	1048576	Yes	L1=$512 \cdot 512$	16384
MVT	16384^2	N/A	$32 \cdot 1024$	No	L1=$8 \cdot 128 \cdot 8$, L2=$8 \cdot 2 \cdot 8$	N/A
BICG	32768^2	N/A	$32 \cdot 1024$	No	L1=$8 \cdot 128 \cdot 8$, L2=$8 \cdot 2 \cdot 8$	N/A

Benchmark	**Pochoir** Timesteps	**Serial Time** Per Iter. (s)
Jacobi-2D	256	4.7 ± 0.03
Seidel-2D	256	8.0 ± 0.20
FDTD-2D	256	3.8 ± 0.05
MVT	N/A	1.7 ± 0.02
BICG	N/A	7.0 ± 0.07
APOP	16384	2.6 ± 0.03

Table 7.3: Benchmark configurations evaluated. For each benchmark the best tile/block size and T_s (number of spliced timesteps) was selected.

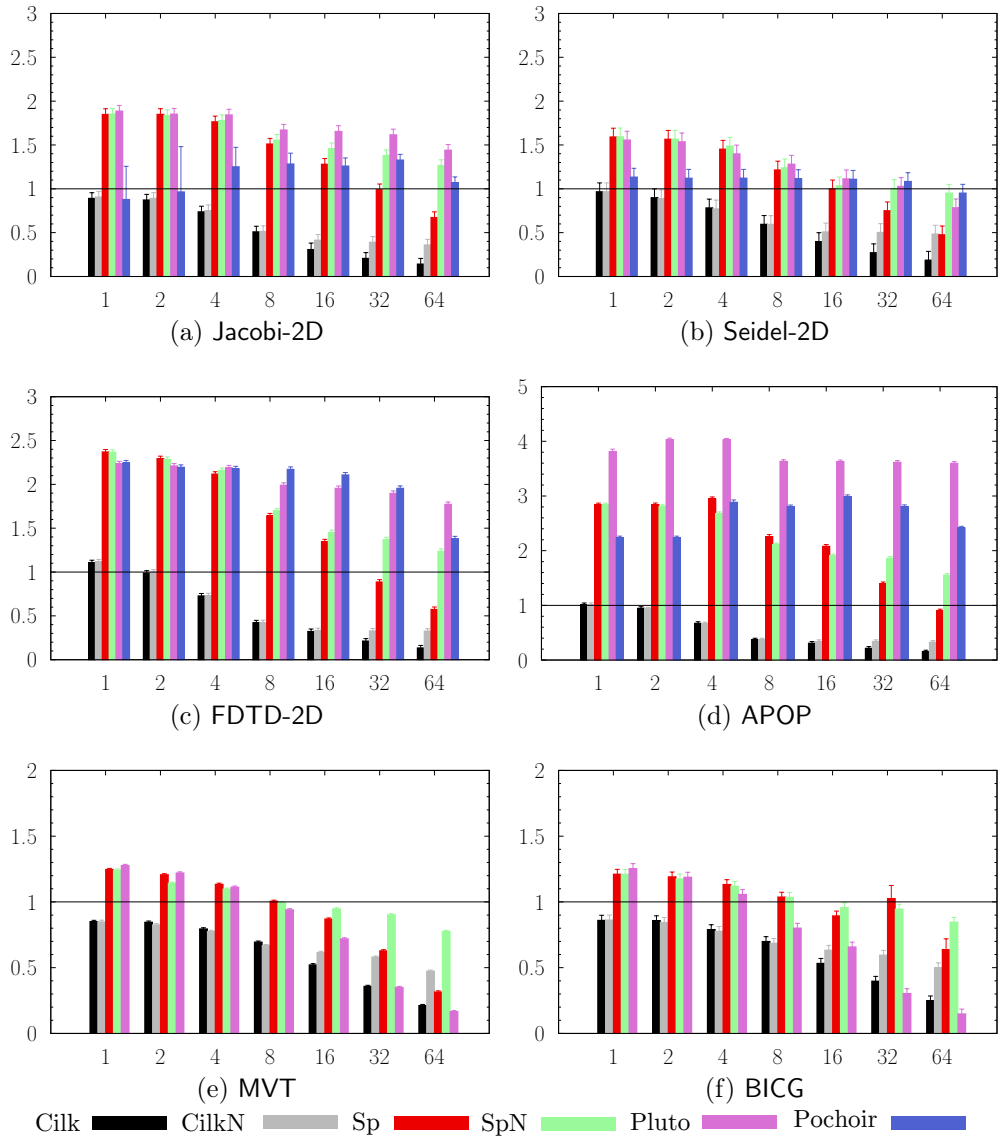


Figure 7.5: Mean speedup of five runs achieved over a hypothetical perfectly scaled serial implementation run on a single core ($\frac{\text{mean execution time}}{\text{serialtime}/c}$, where c is the number of cores). The Pluto compiler time tiles and parallelizes the codes with OpenMP. The CILK and CILKN vars correspond to using Cilk without and with NUMA optimizations. The SP and SPN bars correspond to using splicing scheduler without and with NUMA optimizations. x-axis—number of cores; y-axis—speedup; error bars—standard deviation.

Benchmark (64 cores)	Interference Checks	Context Switches	Delayed Steps	Total Deps.
Jacobi-2D	69,753,410	3,960,272	15,102	66,414
Seidel-2D	59,253,018	2,022,304	30,076	132,712
FDTD-2D	31,047,246	1,998,240	15,100	48,528
APOP	216,225	2,032,848	7,392	19,584
MVT	262,124	98,870	0	0
BICG	1,048,556	393,782	0	0

Benchmark	Space Overhead (1–64 cores, in MB)						
	1	2	4	8	16	32	64
Jacobi-2D	0.8	2	4	7	12	14	16
Seidel-2D	0.5	1	4	4	8	13	16
FDTD-2D	0.5	0.9	2	3	5	7	10
APOP	0.2	0.4	0.8	2	3	5	8
MVT	0	0	0	0	0	0	0
BICG	0	0	0	0	0	0	0

Table 7.4: Runtime statistics using our spliced scheduler.

Benchmark	Config	L1 A.	L1 M.	L2 M.	L3 M.	TLB A.	TLB M.	Inst
Jacobi-2D	Splice	1.6B	0.27B	0.018B	0.004B	0.10B	0.0020B	2.9B
Jacobi-2D	Cilk	1.4B	0.26B	0.068B	0.033B	0.91B	0.0016B	2.7B
Jacobi-2D	Serial	1.2B	0.26B	0.075B	0.036B	0.78B	0.0014B	2.3B
Jacobi-2D	Pluto	1.4B	0.18B	0.058B	0.00051B	0.87B	0.0067B	3.4B
Jacobi-2D	Pochoir	2.1B	0.21B	0.092B	0.00024B	1.3B	0.018B	4.4B
FDTD-2D	Splice	1.0B	0.11B	0.010B	0.0029B	0.63B	0.0013B	1.92B
FDTD-2D	Cilk	0.93B	0.11B	0.046B	0.023B	0.59B	0.0011B	1.8B
FDTD-2D	Serial	0.72B	0.12B	0.063B	0.031B	0.46B	0.0013B	1.55B
FDTD-2D	Pluto	1.4B	0.054B	0.029B	0.00035B	0.91B	0.0045B	3.5B
FDTD-2D	Pochoir	1.5B	0.23B	0.12B	0.00060B	0.95B	0.021B	3.0B
Seidel-2D	Splice	2.1B	0.33B	0.024B	0.0031B	1.36B	0.0018B	11B
Seidel-2D	Cilk	2.1B	0.32B	0.052B	0.024B	1.32B	0.0014B	9.1B
Seidel-2D	Serial	1.7B	0.31B	0.082B	0.037B	1.1B	0.0013B	4.54B
Seidel-2D	Pluto	2.8B	0.084B	0.050B	0.0017B	1.83B	0.015B	6.1B
Seidel-2D	Pochoir	2.1B	0.42B	0.18B	0.00048B	1.4B	0.034B	14B
APOP	Splice	0.38B	0.083B	0.010B	0.0020B	0.24B	0.00098B	1.1B
APOP	Cilk	0.36B	0.080B	0.032B	0.016B	0.23B	0.00088B	1.08B
APOP	Serial	0.36B	0.074B	0.032B	0.016B	0.23B	0.0034B	1.03B
APOP	Pluto	0.47B	0.0021B	0.00011B	0.000055B	0.30B	0.0039B	1.5B
APOP	Pochoir	0.70B	0.00039B	0.000010B	0.000001B	0.43B	0.0000020B	1.8B
MVT	Splice	0.63B	0.0275	0.0097B	0.0030B	0.48B	0.00050B	1.8B
MVT	Cilk	0.73B	0.0275	0.013B	0.0063B	0.48B	0.00040B	1.8B
MVT	Serial	0.36B	0.052B	0.020B	0.0091B	0.23B	0.00030B	1.0B
MVT	Pluto	0.42B	0.18408	0.10B	0.0090B	0.25B	0.084B	1.5B
BICG	Splice	2.9B	0.11B	0.033B	0.013B	1.92B	0.0019B	7.1B
BICG	Cilk	2.9B	0.11B	0.052B	0.025B	1.92B	0.0017B	7.1B
BICG	Serial	1.45B	0.21B	0.082B	0.038B	0.91B	0.0014B	4.13B
BICG	Pluto	1.7B	0.92B	0.41B	0.038B	1.0B	0.34B	5.8B

Table 7.5: Cache, TLB, and instruction count instrumentation on 1 core for all benchmarks.

7.8 Example Program Transformation Walkthrough

To elucidate the transformation process that the compiler undergoes to splice a code, we have written a sketch of the transformation sequence for a simple one-dimensional stencil. The following is the baseline stencil program. It contains a call to the splicing scheduler to initiate splicing every SPLICE_TIMESTEPS timesteps, which will not be effective because effect annotations are not currently present:

```
1  void kernel(A, B, lo, hi) {
2    for (int i = lo; i < hi; i++)
3      A[i] = 0.333 * (B[i]+B[i−1]+B[i+1]);
4  }
5
6  void stencil1D(A, B, lo, hi) {
7    if (hi−lo > THESHOLD1) {
8      mid = (hi−lo)/2+lo;
9      stencil1D(A, B, lo, mid);
10     stencil1D(A, B, mid, hi);
11   } else
12     kernel(A, B, lo, hi);
13 }
14
15 void main() {
16   double* M[2];
17   M[0] = malloc(sizeof(double)*N);
18   M[1] = malloc(sizeof(double)*N);
19   /* initalize M[1] */
20   for (t = 0; t < TIMESTEPS; t++) {
21     if (t % SPLICE_TIMESTEPS == 0)
22       TrySplice(MIN(SPLICE_TIMESTEPS, TIMESTEPS−t))
23     int t0 = t % 2 == 0
24     stencil1D(M[t0], M[!t0], 0, N)
25   }
26 }
```

In the next code snippet, we present an implementation of an example 1-D effect: declaration, construction, interference check, slicing function, subset-equal operator to optimize transitive effect checks, and a sizeof-intersection operator to allow the system to determine whether splicing is viable. These functions are sufficient to enable all the runtime optimizations described in Section 7.6.

```
typedef struct { void* ptr; int lo, hi; } 1deffect;
1deffect* 1D(void* ptr, int lo, int hi) {
  1deffect* e = malloc(sizeof(1deffect))
  e->ptr = ptr; e->lo = lo; e->hi = hi;
  return e;
}
int 1deffect_interfere(const 1deffect* a, const 1deffect* b) {
  return a.ptr == b.ptr && a->lo <= b->hi && b->lo <= a->hi;
}
int 1deffect_subset_equal(const 1deffect* a, const 1deffect* b) {
  return a.ptr == b.ptr && b->lo >= a->lo && b->hi <= a->hi;
}
int 1deffect_sizeof_intersection(const 1deffect* a, const 1deffect* b) {
  return MIN(a->hi,b->hi)-MAX(a->lo,b->lo);
}
void 1deffect_slice(const 1deffect* a, 1deffect** p1, 1deffect** p2) {
  if (a->hi-a->lo < 1024) { *p1 = a; *p2 = NULL; }
  else {
    int hi = MIN(a->lo+1024, a->hi);
    *p1 = 1D(a->ptr, a->lo, hi);
    *p2 = 1D(a->ptr, hi, a->hi);
  }
}
```

In the next code snippet, we present the final user code that is given to the compiler that is spliceable. It contains the added effects, highlighted in orange, that are supplied by the user to enable the system to interleave execution of multiple timesteps of this stencil. Additionally, it includes a parametric kernel to enable pipelining of the base case.

```
/* pipelined version of the kernel using partial effect */
void kernel_pipeline(1deffect* e, A, B, lo, hi) { kernel(A, B, e->lo, e->hi); }

void kernel(A, B, lo, hi) {
  for (int i = lo; i < hi; i++)
    A[i] = 0.333 * (B[i]+B[i-1]+B[i+1]);
}

#pragma splice func Writes(1D(A,lo,hi)), Reads(1D(B,lo-1,hi+1))
void stencil1D(A, B, lo, hi) {
  if (hi-lo > THESHOLD1) {
    mid = (hi-lo)/2+lo;
```

```
13      #pragma splice cont Writes(1D(A,mid,hi)), Reads(1D(B,mid−1,hi+1));
14      stencil1D(A, B, lo, mid);
15      #pragma splice cont Reads() Writes()
16      stencil1D(A, B, mid, hi);
17    } else {
18      #pragma splice cont Reads() Writes()
19      #pragma splice step(pipeline, kernel) Writes(1D(A,lo,hi)), Reads(1D(B,lo−1,hi+1))
20      kernel(A, B, lo, hi);
21    }
22  }
23  void main() {
24    double* M[2];
25    M[0] = malloc(sizeof(double)*N);
26    M[1] = malloc(sizeof(double)*N);
27    /* initalize M[1] */
28    for (t = 0; t < TIMESTEPS; t++) {
29      if (t % SPLICE_TIMESTEPS == 0)
30        TrySplice(MIN(SPLICE_TIMESTEPS, TIMESTEPS−t));
31      int t0 = t % 2 == 0;
32      #pragma splice cont Writes(1D(M[t0],0,N), 1D(M[!t0],0,N));
33      stencil1D(M[t0], M[!t0], 0, N);
34    }
35  }
```

The next snippet of code includes the generated code that saves the effect pragma declarations in the stack (e.g. f_eff), shown highlighted in orange. Highlighted in green is the code generated to transform the program using fork/join concurrency by inserting spawn and sync in the appropriate places, using the effects to test at runtime if a sync is required.

```
1   void stencil1D(double* A, double* B, int lo, int hi) {
2     if (hi−lo > THESHOLD1) {
3       mid = (hi−lo)/2+lo;
4       {f_eff = Writes(1D(A,lo,mid)), Reads(1D(B,lo−1,mid+1));
5        c_eff = Writes(1D(A,mid,hi)), Reads(1D(B,mid−1,hi+1));
6        spawn stencil1D(A, B, lo, mid);
7        if (effects_interfere(f_eff, c_eff)) sync;}
8       {f_eff = Writes(1D(A,mid,hi)), Reads(1D(B,mid−1,hi+1));
9        c_eff = NULL;
10       spawn stencil1D(A, B, mid, hi);
11       if (effects_interfere(f_eff, c_eff)) sync;}
12    } else {
13      {s_eff = Writes(1D(A,lo,hi)), Reads(1D(B,lo−1,hi+1));
```

```
14        c_eff = NULL;}
15      kernel(A, B, lo, hi);
16    }
17  }
18
19  void main() {
20    double* M[2];
21    M[0] = malloc(sizeof(double)*N);
22    M[1] = malloc(sizeof(double)*N);
23    /* initalize M[1] */
24    for (t = 0; t < TIMESTEPS; t++) {
25      if (t % SPLICE_TIMESTEPS == 0)
26        TrySplice(MIN(SPLICE_TIMESTEPS, TIMESTEPS-t));
27      int t0 = t % 2 == 0
28      {f_eff = Writes(1D(M[t0],0,N)), Reads(1D(M[!t0],0,N));
29        c_eff = Writes(1D(M[t0],0,N), 1D(M[!t0],0,N));
30        spawn stencil1D(M[t0], M[!t0], 0, N);
31        if (effects_interfere(f_eff, c_eff)) sync;}
32    }
33  }
```

The final snippet includes blue highlighted regions that are generated to initiate splicing (in the main function) and interleave execution if splicing is chosen at runtime in the stencil1D function. It includes a rough sketch of the runtime structures used to manage the splicing state and perform bookkeeping to track effects and required synchronization.

```
1   typedef struct {
2     /* 'handle' to store fnptr and packed parameters */
3     effect f_eff, c_eff;
4   } ult;
5   typedef struct {
6     bool hasSync = false, inSplicedMode = false;
7     int counter = 0, funcs = -1, cur_ult = 0;
8     ult ults[MAX_SPLICE_ULTS];
9   } splice_state;
10  pthread_local splice_state ss;
11
12  void try_splice_local(ss) {
13    ss.inSplicedMode = true;
14    /* use heuristic to determine of splicing is beneficial */
15    /* if so, create and interleave ss.funcs user-level threads on this pthread */
16    /* else, execute functions in program order */
```

```
17  }
18
19  void stencil1D(A, B, lo, hi) {
20    if (hi−lo > THESHOLD1) {
21      mid = (hi−lo)/2+lo;
22      {f_eff = Writes(1D(A,lo,mid)), Reads(1D(B,lo−1,mid+1));
23       c_eff = Writes(1D(A,mid,hi)), Reads(1D(B,mid−1,hi+1));
24       if (ss.inSplicedMode) {
25         splice_func_cont(ss.cur_ult, f_eff, c_eff);
26         try_context_switch_ult((ss.cur_ult+1) % ss.funcs);
27       }
28       spawn stencil1D(A, B, lo, mid)
29       if (effects_interfere(f_eff, c_eff)) sync;}
30      {f_eff = Writes(1D(A,mid,hi)), Reads(1D(B,mid−1,hi+1));
31       c_eff = NULL;
32       if (ss.inSplicedMode) {
33         splice_func_cont(ss.cur_ult, f_eff, c_eff);
34         try_context_switch_ult((ss.cur_ult+1) % ss.funcs);
35       }
36       spawn stencil1D(A, B, mid, hi)
37       if (effects_interfere(f_eff, c_eff)) sync;}
38    } else {
39      {s_eff = Writes(1D(A,lo,hi)), Reads(1D(B,lo−1,hi+1));
40       c_eff = NULL;
41       if (ss.inSplicedMode)
42         partial_effect = s_eff;
43         while(splice_step(ss.cur_ult, s_eff, &partial_effect)) {
44           kernel_pipeline(partial_effect, A, B, lo, hi)
45         }
46         if (partial_effect != NULL)
47           /* suspend this stack frame */
48       } else
49         kernel(A, B, lo, hi);
50    }
51  }
52
53  void main() {
54    for (t = 0; t < TIMESTEPS; t++) {
55      if (t % SPLICE_TIMESTEPS == 0)
56        ss.funcs = MIN(SPLICE_TIMESTEPS, TIMESTEPS−t)
57      t0 = t % 2 == 0
58      {f_eff = Writes(1D(M[t0],0,N)), Reads(1D(M[!t0],0,N))
59       c_eff = Writes(1D(M[t0],0,N), 1D(M[!t0],0,N))
```

```
60
61        if (ss.funcs != −1) {
62          ss.ults[ss.counter].{f_eff,c_eff} = {f_eff,c_eff}
63          ss.ults[ss.counter++].handle = { stencil1D, struct { M[t0], M[!t0], 0, N } }
64        } else
65          spawn stencil1D(M[t0], M[!t0], 0, N);
66
67        if (ss.counter == ss.funcs) try_splice_local(ss)
68
69        if (effects_interfere(f_eff, c_eff)) {
70          if (ss.counter != −1) ss.ults[ss.counter−1].hasSync = true;
71          else sync;
72        }}
73      }
74    }
```

7.9 Discussion

Compile-time optimization across function boundaries are inherently limited by the inter-procedural analysis and the availability of enough information—source code, effects, or higher-level semantics—to enable such transformations. Achieving the best performance with any strategy—traditional compiler analysis, compilation from domain-specific languages, or runtime optimization—requires a careful tuning of a large parameter space. Our results cannot be used to claim any strategy as being definitively superior across all the benchmarks and configurations considered. The alternative strategies considered can reason in a richer dependence space and perform a greater number of transformations—permutation, reversal, tiling, duplicated execution, etc.—that we do not consider in this work. While these strategies can often handle more general dependence patterns, our approach works best with local dependences that enable continued splicing with a limited number of delayed steps. We consider our approach a complement to the compile-time optimizers, to be used when they cannot be applied.

CHAPTER 8

Conclusion

The widespread use of work stealing necessitates the development of mechanisms to study the behavior of individual executions. The algorithms presented in Chapter 4 allow the efficient construction of steal trees that enable low overhead tracing and replay of async-finish programs scheduled using work-first or help-first work stealing schedulers. We also 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 or constrained stealing.

In Chapter 5, we presented scalable algorithms for persistence-based load balancing and work stealing. The hierarchical persistence-based load balancing algorithm presented locally rebalances the load in a greedy fashion. The work stealing algorithm is optimized for distributed memory machines, by coalescing remote operations, reducing the duration of locked operations, and enabling concurrent steal operations that allow overlapped task migration. We presented retentive stealing to further adapt work stealing for iterative applications.

Work stealing is traditionally considered not to scale beyond small core counts, due to its perceived limitations: randomization, the need to repeatedly rebalance, the cost of termination detection, and the potential interference with application execution. While not universally applicable, we demonstrate that work stealing scales better than commonly believed. Retentive stealing is also shown to improve execution efficiency by incrementally improving the load balance and reducing the overheads associated with steal-

ing.

In Chapter 6, we present an approach to optimize fork/join programs for data locality and grain size selection. We describe two different methodologies: (1) user-specified steal tree construction that requires additional programmer effort and is not adaptive and (2) an automatic iterative optimization scheme that nearly converges to the same performance. The evaluation demonstrates that we can obtain up to 2.5x performance improvement using our iterative scheme. We show that high performance still can be obtained without the application-specific knowledge user specification requires while maintaining the efficiency and automatic load balancing that work stealing provides.

In Chapter 7, we demonstrate a approach to cache locality optimization across function invocations of effect-annotated recursive functions. We demonstrated significant performance improvements using our strategy, competitive with the best alternative optimization strategies. This included efficiently tracking effects, preventing dependence violations, and lightweight-thread-based scheduling to interleave execution of function invocations. In particular, we demonstrated splicing across multiple time iterations in stencil computations to mimic the complex diamond-tiling transformation in Pluto and time tiling in Pochoir.

Building on the techniques presented in Chapter 7, structuring libraries as collections of effect-annotated recursive functions can enable automatic runtime optimization of applications composed of such functions.

REFERENCES

[1] 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*, ser. PLDI '12, 2012, pp. 531–542.

[2] L. Dagum and R. Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming," *IEEE Computational Science & Engineering*, vol. 5, no. 1, January-March 1998.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55–69, 1996.

[4] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 2005, pp. 519–538.

[5] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-java: the new adventures of old x10," in *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*. ACM, 2011, pp. 51–61.

[6] G. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha, "Implementation of a portable nested data-parallel language," *Journal of Parallel and Distributed Computing*, vol. 21, pp. 102–111, 1994.

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

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

[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*, ser. IPDPS '09. IEEE, 2009, pp. 1–12.

[10] 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*, ser. SPAA '04, 2004, pp. 133–144.

[11] T. Karunaratna, "Nondeterminator-3: a provably good data-race detector that runs in parallel," Ph.D. dissertation, Massachusetts Institute of Technology, 2005.

[12] 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*, ser. PPoPP '08, 2008, pp. 163–174.

[13] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The data locality of work stealing," *Theory of Computing Systems*, vol. 35, no. 3, pp. 321–347, 2002.

[14] N. R. Tallent and J. M. Mellor-Crummey, "Identifying performance bottlenecks in work-stealing computations," *IEEE Computer*, vol. 42, no. 12, pp. 44–50, 2009.

[15] A. Darte, J. Mellor-Crummey, R. Fowler, and D. C. Miranda, "Generalized multipartitioning of multi-dimensional arrays for parallelizing line-sweep computations," *JPDC*, vol. 63, no. 9, pp. 887–911, 2003.

[16] Ü. V. Çatalyürek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 7, pp. 673–693, 1999.

[17] R. D. Williams, "Performance of dynamic load balancing algorithms for unstructured mesh calculations," *Concurrency: Pract. Exper.*, vol. 3, pp. 457–481, October 1991.

[18] R. Das, Y.-S. Hwang, M. Uysal, J. Saltz, and A. Sussman, "Applying the CHAOS/PARTI library to irregular problems in computational chemistry and computational aerodynamics," in *Scalable Parallel Libraries Conference*, oct 1993, pp. 45 –56.

[19] B. Hendrickson and R. Leland, "An improved spectral graph partitioning algorithm for mapping parallel computations," *SIAM J. Sci. Comput.*, vol. 16, pp. 452–469, March 1995.

[20] Ü. V. Çatalyürek, E. G. Boman, K. D. Devine, D. Bozdag, R. T. Heaphy, and L. A. Riesen, "A repartitioning hypergraph model for dynamic load balancing," *JPDC*, vol. 69, no. 8, pp. 711–724, 2009.

[21] G. Karypis, K. Schloegel, and V. Kumar, "Parmetis: Parallel graph partitioning and sparse matrix ordering library," *Version 1.0, Dept. of Computer Science, University of Minnesota*, 1997.

[22] 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 *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95*, Santa Barbara, California, July 1995, mIT. pp. 207–216.

[23] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proc. of 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, 2004.

[24] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of OOPSLA'93*, A. Paepcke, Ed. ACM Press, September 1993, pp. 91–108.

[25] A. Chandramowlishwaran, K. Knobe, and R. Vuduc, "Performance evaluation of concurrent collections on high-performance multicore computing systems," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

[26] G. Gao, T. Sterling, R. Stevens, M. Hereld, and W. Zhu, "ParalleX: A study of a new parallel computation model," in *IEEE International Parallel and Distributed Processing Symposium, 2007*, march 2007, pp. 1 –6.

[27] N. H. Darach Golden and S. McGrath, "Parallel adaptive mesh refinement for large eddy simulation using the finite element methods," in *PARA*, 1998, pp. 172–181.

[28] G. Zheng, A. Bhatele, E. Meneses, and L. V. Kale, "Periodic Hierarchical Load Balancing for Large Supercomputers," *International Journal of High Performance Computing Applications (IJHPCA)*, March 2011.

[29] J. C. Phillips et al., "Scalable molecular dynamics with namd," *Journal of Computational Chemistry*, vol. 26, no. 16, pp. 1781–1802, 2005.

[30] R. D. Blumofe and P. A. Lisiecki, "Adaptive and reliable parallel computing on networks of workstations," in *USENIX*, 1997. [Online]. Available: http://dl.acm.org/citation.cfm?id=1268680.1268690 pp. 10–10.

[31] R. V. van Nieuwpoort, T. Kielmann, and H. E. Bal, "Efficient load balancing for wide-area divide-and-conquer applications," in *PPoPP*, 2001. [Online]. Available: http://doi.acm.org/10.1145/379539.379563 pp. 34–43.

[32] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. K. Panda, "High performance remote memory access communication: The ARMCI approach," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 233–253, 2006.

[33] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *SC*, 2009.

[34] V. A. Saraswat, P. Kambadur, S. B. Kodali, D. Grove, and S. Krishnamoorthy, "Lifeline-based global load balancing," in *PPoPP*, 2011, pp. 201–212.

[35] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," *ACM Sigplan Notices*, vol. 33, no. 5, pp. 212–223, 1998.

[36] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," in *IPDPS'09*, 2009, pp. 1–12.

[37] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar, "Hierarchical place trees: A portable abstraction for task parallelism and data movement," in *LCPC '10*. Springer, 2010, pp. 172–187.

[38] S.-J. Min, C. Iancu, and K. Yelick, "Hierarchical work stealing on many-core clusters," in *PGAS '11*, 2011.

[39] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The data locality of work stealing," *TOCS*, vol. 35, no. 3, pp. 321–347, 2002.

[40] G. E. Blelloch and P. B. Gibbons, "Effectively sharing a cache among threads," in *SPAA '04*, 2004, pp. 235–244.

[41] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," *Acm Sigplan Notices*, vol. 40, no. 10, pp. 519–538, 2005.

[42] L. V. Kale and S. Krishnan, *CHARM++: a portable concurrent object oriented system based on C++*, 1993.

[43] G. Zheng, "Achieving high performance on extremely large parallel machines: performance prediction and load balancing," Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[44] D. S. Nikolopoulos, E. Artiaga, E. Ayguadé, and J. Labarta, "Exploiting memory affinity in OpenMP through schedule reuse," *ACM SIGARCH Computer Architecture News*, vol. 29, no. 5, pp. 49–55, 2001.

[45] S. L. Olivier, B. R. de Supinski, M. Schulz, and J. F. Prins, "Characterizing and mitigating work time inflation in task parallel programs," in *SC '12*, 2012, pp. 1–12.

[46] L. Huang, H. Jin, L. Yi, and B. Chapman, "Enabling locality-aware computations in OpenMP," *Sci. Program.*, vol. 18, no. 3-4, pp. 169–181, Aug. 2010.

[47] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C. A. Nelson, and C. D. Offner, "Extending OpenMP for NUMA machines," in *SC '00*, 2000.

[48] F. Broquedis, N. Furmento, B. Goglin, R. Namyst, and P.-A. Wacrenier, "Dynamic task and data placement over NUMA architectures: An OpenMP runtime perspective," in *IWOMP '09*, 2009, pp. 79–92.

[49] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, M. Spiegel, and J. F. Prins, "OpenMP task scheduling strategies for multicore NUMA systems," *Int. J. High Perform. Comput. Appl.*, vol. 26, no. 2, May 2012.

[50] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The pochoir stencil compiler," in *SPAA '11*. ACM, 2011, pp. 117–128.

[51] A. Chien, W. Feng, V. Karamcheti, and J. Plevyak, "Techniques for efficient execution of fine-grained concurrent programs," in *LCPC '93*. Springer, 1993, pp. 160–174.

[52] Y. Sun, G. Zheng, P. Jetley, and L. V. Kale, "An Adaptive Framework for Large-scale State Space Search," in *Proceedings of Workshop on Large-Scale Parallel Processing (LSPP) in IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2011*, Anchorage, Alaska, May 2011.

[53] V. Kumar, D. Frampton, S. M. Blackburn, D. Grove, and O. Tardieu, "Work-stealing without the baggage," *OOPSLA '12*, vol. 47, no. 10, pp. 297–314, 2012.

[54] A. Tzannes, G. C. Caragea, R. Barua, and U. Vishkin, "Lazy binary-splitting: A run-time adaptive work-stealing scheduler," in *PPoPP '10*, 2010.

[55] M. A. Rainey, "Effective scheduling techniques for high-level parallel programming languages," Ph.D. dissertation, Chicago, IL, USA, 2010.

[56] E. Mohr, D. A. Kranz, and R. H. Halstead Jr, "Lazy task creation: A technique for increasing the granularity of parallel programs," *TPDS*, vol. 2, no. 3, pp. 264–280, 1991.

[57] S. L. Olivier and J. F. Prins, "Evaluating OpenMP 3.0 run time systems on unbalanced task graphs," in *IWOMP '09*. Springer, 2009, pp. 63–78.

[58] D. Hendler and N. Shavit, "Non-blocking steal-half work queues," in *PODC '02*. ACM, 2002, pp. 280–289.

[59] R. M. Burstall and J. Darlington, "A transformation system for developing recursive programs," *J. ACM*, vol. 24, no. 1, pp. 44–67, Jan. 1977.

[60] V. K. Nandivada, J. Shirako, J. Zhao, and V. Sarkar, "A transformation framework for optimizing task-parallel programs," *ACM Trans. Program. Lang. Syst.*, vol. 35, no. 1, p. 3, 2013.

[61] R. Rugina and M. C. Rinard, "Automatic parallelization of divide and conquer algorithms," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1999. [Online]. Available: http://doi.acm.org/10.1145/301104.301111 pp. 72–83.

[62] R. Rugina and M. C. Rinard, "Symbolic bounds analysis of pointers, array indices, and accessed memory regions," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 2, pp. 185–235, 2005. [Online]. Available: http://doi.acm.org/10.1145/1057387.1057388

[63] R. Rugina and M. C. Rinard, "Pointer analysis for structured parallel programs," *ACM Trans. Program. Lang. Syst.*, vol. 25, no. 1, pp. 70–116, 2003. [Online]. Available: http://doi.acm.org/10.1145/596980.596982

[64] A. K. Sujeeth, T. Rompf, K. J. Brown, H. Lee, H. Chafi, V. Popic, M. Wu, A. Prokopec, V. Jovanovic, M. Odersky, and K. Olukotun, "Composition and reuse with compiled domain-specific languages," in *European Conference on Object-Oriented Programming*, 2013. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-39038-8_3 pp. 52–78.

[65] R. Chandra, A. Gupta, and J. L. Hennessy, "Data locality and load balancing in COOL," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993. [Online]. Available: http://doi.acm.org/10.1145/155332.155358 pp. 249–259.

[66] T. L. Veldhuizen, "Active libraries and universal languages," Ph.D. dissertation, Indiana University, 2004.

[67] K. Kennedy, B. Broom, K. Cooper, J. Dongarra, R. Fowler, D. Gannon, L. Johnsson, J. Mellor-Crummey, and L. Torczon, "Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries," *Journal of Parallel and Distributed Computing*, vol. 61, no. 12, pp. 1803–1826, 2001.

[68] S. Z. Guyer and C. Lin, "An annotation language for optimizing software libraries," in *Conference on Domain-specific Languages*, 1999. [Online]. Available: http://doi.acm.org/10.1145/331960.331970 pp. 39–52.

[69] D. Quinlan, M. Schordan, Q. Yi, and A. Saebjornsen, "Classification and utilization of abstractions for optimization," in *Leveraging Applications of Formal Methods*. Springer Berlin Heidelberg, 2006, vol. 4313, pp. 57–73. [Online]. Available: http://dx.doi.org/10.1007/11925040_5

[70] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson, "Scheduling threads for constructive cache sharing on cmps," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, 2007. [Online]. Available: http://doi.acm.org/10.1145/1248377.1248396 pp. 105–115.

[71] Y. Guo, J. Zhao, V. Cavé, and V. Sarkar, "SLAW: A scalable locality-aware adaptive work-stealing scheduler," in *IEEE International Symposium on Parallel and Distributed Processing*, 2010. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2010.5470425 pp. 1–12.

[72] S. Heumann, V. S. Adve, and S. Wang, "The tasks with effects model for safe concurrency," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2013. [Online]. Available: http://doi.acm.org/10.1145/2442516.2442540 pp. 239–250.

[73] S. Treichler, M. Bauer, and A. Aiken, "Language support for dynamic, hierarchical data partitioning," in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2013. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509545 pp. 495–514.

[74] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: expressing locality and independence with logical regions," in *SC*, 2012, p. 66.

[75] E. M. Westbrook, J. Zhao, Z. Budimlic, and V. Sarkar, "Permission regions for race-free parallelism," in *Runtime Verification Conference*, 2011, pp. 94–109.

[76] A. Pan and V. Pai, "Runtime-driven shared last-level cache management for task-parallel programs," Department of Electrical and Computer Engineering, Purdue University, Tech. Rep. 466, 2015.

[77] Y. Jo and M. Kulkarni, "Enhancing locality for recursive traversals of recursive structures," in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2011. [Online]. Available: http://doi.acm.org/10.1145/2048066.2048104 pp. 463–482.

[78] J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas, and K. Li, "Thread scheduling for cache locality," in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996. [Online]. Available: http://doi.acm.org/10.1145/237090.237151 pp. 60–71.

[79] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos, "First-class user-level threads," in *ACM Symposium on Operating Systems Principles*, 1991. [Online]. Available: http://doi.acm.org/10.1145/121132.344329 pp. 110–121.

[80] K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: An API for programming with millions of lightweight threads," in *IEEE International Symposium on Parallel and Distributed Processing*, 2008, pp. 1–8.

[81] "Boost Context," http://www.boost.org/doc/libs/1_56_0/libs/context/doc/html/index.html.

[82] K. Agrawal, C. E. Leiserson, and J. Sukha, "Executing task graphs using work-stealing," in *IEEE International Symposium on Parallel and Distributed Processing*, 2010, pp. 1–12.

[83] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Tasirlar, "Concurrent collections," *Scientific Programming*, vol. 18, no. 3, pp. 203–217, 2010.

[84] E. Chan, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn, "Supermatrix out-of-order scheduling of matrix operations for smp and multi-core architectures," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, 2007. [Online]. Available: http://doi.acm.org/10.1145/1248377.1248397 pp. 116–125.

[85] O. Tardieu, H. Wang, and H. Lin, "A work-stealing scheduler for x10's task parallelism with suspension," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2012, pp. 267–276.

[86] 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*, ser. PPoPP '07, 2007, pp. 271–271.

[87] B. Fulgham, "Computer language benchmarks game," August 2012. [Online]. Available: http://shootout.alioth.debian.org/

[88] J. F. Box, "Guinness, Gosset, Fisher, and Small Samples," *Statistical Science*, vol. 2, no. 1, pp. 45–52, Feb. 1987.

[89] 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*, vol. 23, no. 10, pp. 1923–1933, 2012.

[90] C. Joerg and B. C. Kuszmaul, "Massively parallel chess," in *Proceedings of the Third DIMACS Parallel Implementation Challenge, Rutgers*, 1994.

[91] P. Charles, C. Grothoff, V. Saraswat et al., "X10: an object-oriented approach to non-uniform cluster computing," in *OOPSLA*, 2005, pp. 519–538.

[92] N. Francez, "Distributed termination," *ACM Trans. Program. Lang. Syst.*, vol. 2, pp. 42–55, January 1980.

[93] "NERSC Hopper," http://www.nersc.gov/users/computational-systems/hopper.

[94] "ALCF Intrepid," http://www.alcf.anl.gov/intrepid.

[95] "OLCF Titan," http://www.olcf.ornl.gov/computing-resources/titan.

[96] G. Baumgartner et al., "Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models," *Proc. of IEEE*, vol. 93, no. 2, pp. 276–292, 2005.

[97] A. Szabo and N. S. Ostlund, *Modern Quantum Chemistry*. New York: McGraw-Hill Inc., 1996.

[98] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin, "Reducers and other Cilk++ hyperobjects," in *SPAA '09*. ACM, 2009, pp. 79–90.

[99] L.-N. Pouchet, "Polybench: The polyhedral benchmark suite," 2012.

[100] J.-S. Park, M. Penner, and V. K. Prasanna, "Optimizing graph algorithms for improved cache performance," *TPDS*, vol. 15, no. 9, pp. 769–782, 2004.

[101] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber et al., "The NAS parallel benchmarks," *Int. J. High Perform. Comput. Appl.*, vol. 5, no. 3, pp. 63–73, 1991.

[102] S. Seo, G. Jo, and J. Lee, "Performance characterization of the NAS parallel benchmarks in OpenCL," in *IISWC '11*, Nov 2011.

[103] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *Journal of the ACM (JACM)*, vol. 27, no. 4, pp. 831–838, 1980.

[104] 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 *PPOPP '95*, 1995, pp. 207–216.

[105] J. S. Danaher, I.-T. A. Lee, and C. E. Leiserson, "Programming with Exceptions in JCilk," *Sci. Comput. Program.*, vol. 63, no. 2, pp. 147–171, Dec. 2006.

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

[107] "The Task Parallel Library," http://msdn.microsoft.com/en-us/magazine/cc163340.aspx, Oct. 2007.

[108] "The X10 Programming Language," www.research.ibm.com/x10/, Mar. 2006.

[109] "MIT Cilk 5.4.6," http://supertech.lcs.mit.edu/cilk.

[110] V. Maslov, "Delinearization: An efficient way to break multiloop dependence equations," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1992. [Online]. Available: http://doi.acm.org/10.1145/143095.143130 pp. 152–161.

[111] M. E. Conway, "Design of a separable transition-diagram compiler," *Commun. ACM*, vol. 6, no. 7, pp. 396–408, July 1963. [Online]. Available: http://doi.acm.org/10.1145/366663.366704

[112] D. Lea, "A java fork/join framework," in *Proceedings of the ACM conference on Java Grande*, 2000, pp. 36–43.

[113] A. D. Robison, "Composable Parallel Patterns with Intel Cilk Plus," *Computing in Science and Engineering*, vol. 15, no. 2, pp. 66–71, 2013.

[114] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, *Cilk: An efficient multithreaded runtime system*, 1995, vol. 30, no. 8.

[115] "Pluto – an automatic parallelizer and locality optimizer for affine loop nests," http://pluto-compiler.sourceforge.net.

[116] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, 2008. [Online]. Available: http://doi.acm.org/10.1145/1375581.1375595 pp. 101–113.

[117] U. Bondhugula, V. Bandishti, and I. Pananilath, "Diamond Tiling: Tiling Techniques to Maximize Parallelism for Stencil Computations," Indian Institute of Science, Tech. Rep., October 2015. [Online]. Available: http://drona.csa.iisc.ernet.in/~uday//publications/diamond-tiling-submission-copy.pdf

[118] J. Lifflander, S. Krishnamoorthy, and L. V. Kalé, "Optimizing data locality for fork/join programs using constrained work stealing," in *SC*, 2014, pp. 857–868.