Optimizing Data Locality for Fork/Join Programs Using Constrained Work Stealing

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

Sriram Krishnamoorthy
Adv. Comp., Math & Data Division
Pacific Northwest National Lab
Email: sriram@pnnl.gov

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

***Abstract*—We present an approach to improving data locality across different phases of fork/join programs scheduled using work stealing. The approach consists of: (1) user-specified and automated approaches to constructing a *steal tree*, the schedule of steal operations, and (2) *constrained work-stealing* algorithms that constrain the actions of the scheduler to mirror a given steal tree. These are combined to construct work-stealing schedules that maximize data locality across computation phases while ensuring load balance within each phase. These algorithms are also used to demonstrate *dynamic coarsening*, an optimization to improve spatial locality and sequential overheads by combining many finer-grained tasks into coarser tasks while ensuring sufficient concurrency for locality-optimized load balance. Implementation and evaluation in Cilk demonstrate performance improvements of up to 2.5x on 80 cores. We also demonstrate that dynamic coarsening can combine the performance benefits of coarse task specification with the adaptability of finer tasks.**

***Keywords*—fork/join, cilk, data locality, task granularity**

I. Introduction

Fork/join parallelism implemented using work stealing is a popular choice to enable productive programming on multi-core systems and can be found in Java [1], Cilk [2], OpenMP [3], X10 [4], and extensions to C/C++ [5], [6]. Fork/join parallelism focuses on the control flow and involves dividing a task into concurrent sub-tasks and combining their results in a follow-on task. 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 paper, 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 redistribution. 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 work 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
- 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.

SC14, November 16-21, 2014, New Orleans
978-1-4799-5500-8/14/$31.00 ©2014 IEEE

II. Background

We present our approach in the context of Cilk, an exemplar fork/join model. In this section, we briefly describe the Cilk scheduler and its data access behavior. More detailed descriptions can be found elsewhere [7], [8].

Cilk is a parallel programming extension to the C language that introduces three additional keywords: `cilk`, `spawn`, and `sync`. The `cilk` keyword specifies that a function is capable of being executed in parallel. The `spawn` keyword specifies that the invoked function, referred to as the *spawned* task, is concurrent with the statements that follow in the spawning task. No statement following the `sync` statement in a task can be executed until all preceding tasks have completed.

Every function invocation is treated as a *task*. At any given point in time, the sequence of instructions remaining to be executed in a task is referred to as the task's *continuation*. A continuation often is marked by a goto label and referred to by an integer. A *closure* refers to 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
```

A worker with a valid closure executes the closure in a work-first (i.e. depth-first) fashion. A spawned task is immediately executed, allowing the continuation of the currently executing task to be stolen. Upon fully executing a task, a worker returns to execute the invoking task. When no local work is available, a thread becomes a thief and randomly attempts to steal the oldest continuation from another thread. The state of this continuation is encapsulated in a closure, which is then executed. This proceeds until the root task completes execution. A *schedule* is a specification of an ordered list of tasks executed by each thread.

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 IV-C). 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.

III. 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 IV-A) minimizes user effort but cannot immediately optimize for data locality and load balance. User-specified steal trees (discussed in Section IV-B) provide direct control to the user while requiring additional user effort.

We describe three constrained schedulers (discussed in Section IV-C) 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 paper 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);
```

IV. Detailed Design

A. 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. Previous work [9] has shown that work-stealing schedulers can be traced with low overhead. Instead of explicitly recording the execution order and thread for every task, the authors build a *steal tree* that encapsulates the schedule. The steal tree is a recursive structure that fully specifies the schedule for every task in a program by exploiting the hierarchical relationship between steals within the spawn structure of the application.

As described in [9], the steal tree is built by storing for each steal: the associated level (the nested spawn depth for a task), the position of that continuation within the victim's task, and the rank of the stealing thread. Due to the properties of work-first work stealing, a steal from a worker must occur at the highest level in the tree until all of the continuations at that level are either stolen or executed.

The steal tree described in [9] records the schedule for the entire program. We use the theory presented in that paper but employ a different design. Our goal beyond that work 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.

B. 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 scheduler. 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 [9]. 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);
```

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.

C. 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 1, we depict how the three types of constrained schedulers can improve a default schedule:

- STOWS (strict ordered work stealing): the work-stealing scheduler exactly follows a template schedule t by guiding each worker to execute the tasks in the order prescribed by t.
- STUWS (strict unordered work stealing): the 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 (relaxed work stealing): the 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

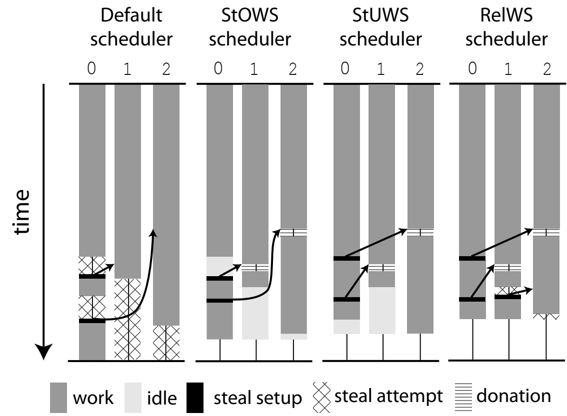


Fig. 1: Example computation scheduled with the default scheduler and then modified with three types of constrained work stealing. Default scheduler: all workers begin busy with work then attempt to steal when they finish. STOWS reproduces this schedule without having to search for work. STUWS is able to revise the order on thread 0, reducing the time. RELWS performs an additional steal, further balancing the workload.

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 4.1: 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 4.2: 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.

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 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 [8] 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);
```

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 designated 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);
```

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.

Experimental Setup. All of the experiments in this paper 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 [7] 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.

Overhead Evaluation. In Figure 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 stealing 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.

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.

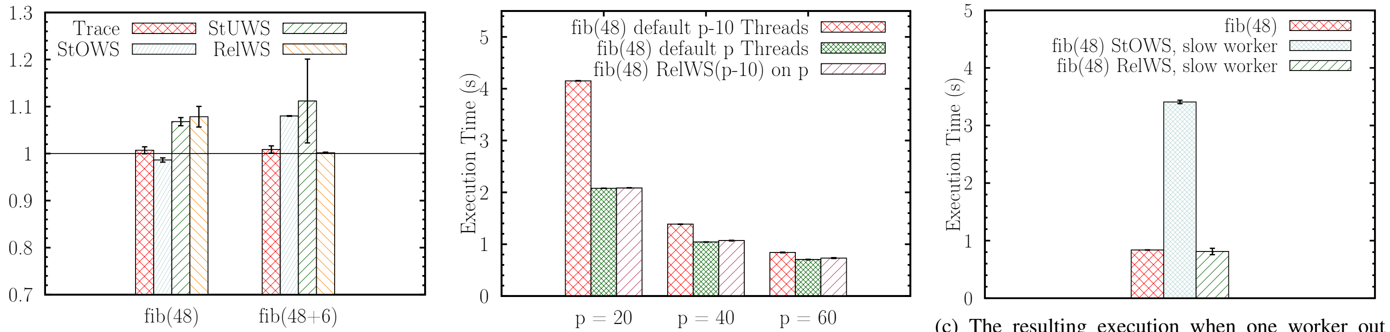
The strict unordered scheduler causes high average overhead but with a large standard deviation, indicating that the execution time is unpredictable, because the unordered scheduler has varying performance depending on the order of execution. However, RelWS recovers the lost performance entirely by adapting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

In Figure 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 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 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.

V. 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.



(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.

Fig. 2: The overhead of the fib microbenchmark with tracing and three scheduling schemes (2a) along with benchmarks to show how RELWS adapts to dynamic variations (2b, 2c).

```
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 I 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 [10]. 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 [11]. 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 [12], implemented in C and OpenMP [13]. 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 [12], implemented in C and OpenMP [13]. The parallel prefix benchmark performs a prefix sum on an array of doubles in parallel [14].

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:

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.

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.

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

Benchmark	Problem	Configuration	Tasks
heat	nx = ny = 32768	block = 64x8192	2k
floyd-war.	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 I: Benchmark configurations (for mg, the number of tasks depends on the level).

of the parallel prefix kernel, scheduling them with STUWS. For user-specified work partitioning, we programmatically construct a steal tree that is used for all of the phases.

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.

A. Empirical Evaluation

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 3 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.

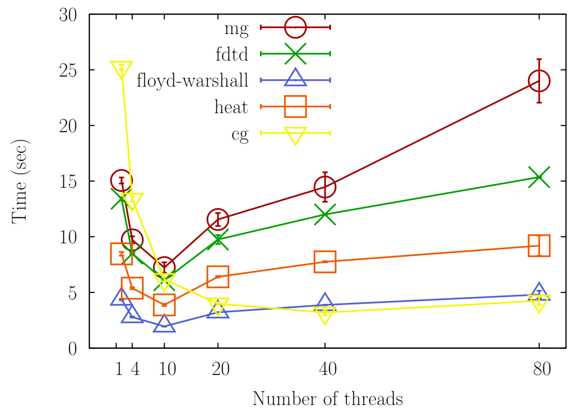


Fig. 6: Execution time for data redistribution with the fork/join initializer exploiting first-touch. Each point is the mean and standard deviation of five runs.

Figure 4 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-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 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.

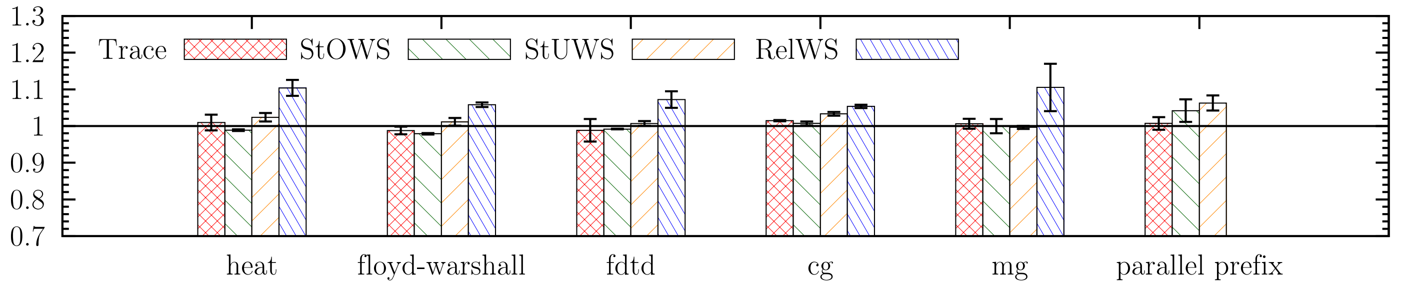


Fig. 3: Normalized execution time of four configurations (mean({Trace, StOWS, StUWS, RelWS})/mean(Baseline)) compared to the default Cilk scheduler. Error bars are relative standard deviation with a sample size of 5.

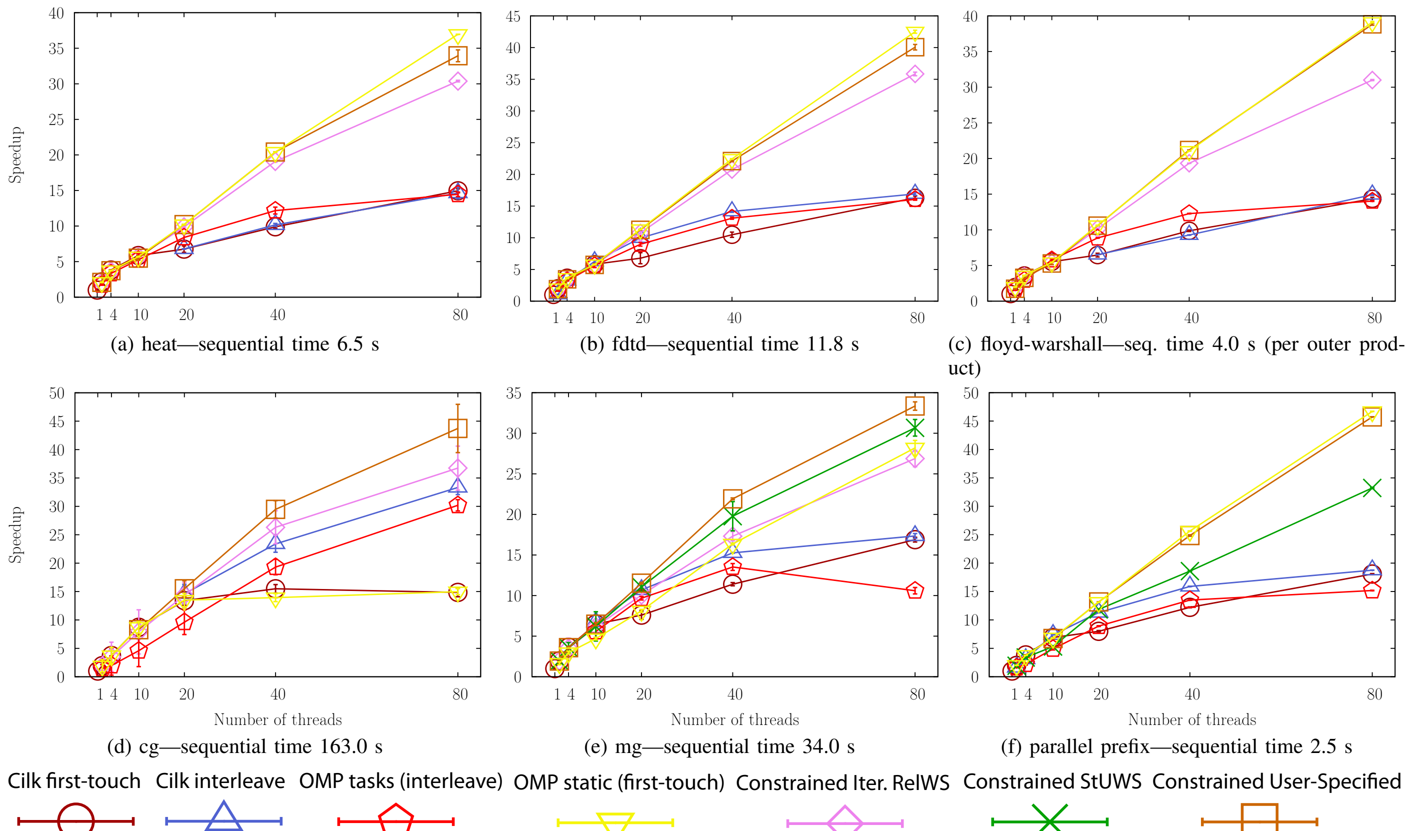


Fig. 4: Speedup achieved strong scaling to 80 cores with respect to a single thread (legend shown above). Cilk first-touch is the baseline using the default Cilk scheduler with no tracing overhead and first-touch. Compared to Cilk interleaved, OMP schedule(static), OMP tasks on all the loops. Speedup 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.

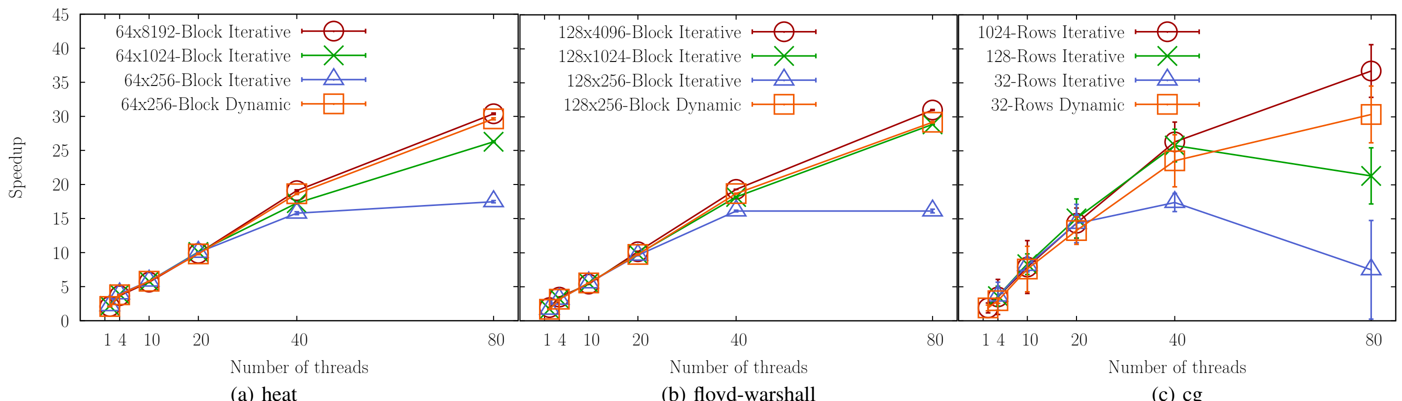


Fig. 5: Speedup achieved for three grain sizes using relaxed work stealing iteratively. These are compared with using a dynamic grain size, starting with a small user-specified grain size.

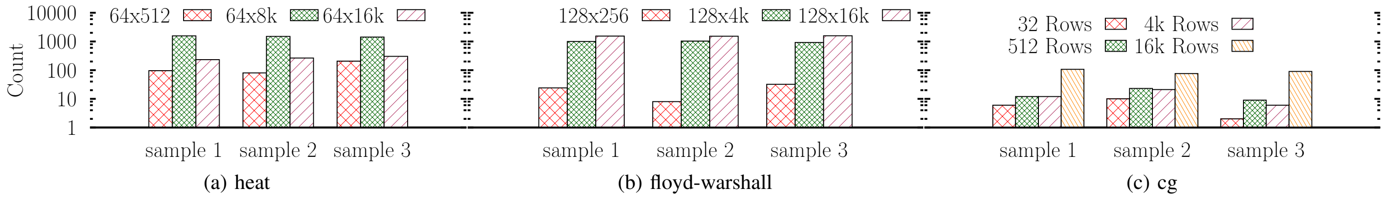


Fig. 7: Histograms that depict the dynamic grain size distribution after convergence for three samples per benchmark.

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 the following table, 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.

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

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 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). 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.

B. Productivity

To demonstrate the productivity of our approach, we measured the lines of code (using David Wheelers 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 include the entire source code for each benchmark:

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

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

VI. 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 paper. 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 sub-tasks and performing a coarser sequential computation. This enables the runtime to dynamically coarsen the tasks and improve performance without impacting load balance.

Figure 5 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 7, 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.

VII. Related Work

Cilk [7] employs random work stealing with a work-first execution strategy. Guo et al. [15] studied help-first scheduling policies to improve the load balance achieved in practice. Hierarchical place trees [16] and related approaches [17], [18] 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 [19] 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 [20] 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. We employ the steal tree design by Lifflander et al. [9] but employ enhanced replay algorithms compared to the strict versions used in their work. 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++ [21] explicitly associates computation with data objects and performs persistence-based load balancing [22] that is coupled with data migration. Our approach does not impose such tight binding of computation and the data it operates upon. Retentive work stealing [9], [23] incrementally improves load balance by reusing a prior schedule but does not consider data redistribution.

Nikolopolous et al. [24] studied reusing loop schedules to improve memory affinity for OpenMP looping constructs. Olivier et al. [25] 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 [26], [27], [28] and modifications to the random work stealing policy [29] also have been considered for OpenMP task programs.

The Pochoir compiler [30] 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 [31] have been employed. Charm++ has adaptive grain size control for state space search [32]. Other work has focused on only splitting grain sizes when a worker is in need of work, which is feasible with a managed runtime [33], [34], [35]. Lazy task creation has

been used to increase granularity [36], while other work has focused on increasing the steal granularity [37], [38], 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.

VIII. Conclusions

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. We also show that dynamic coarsening can effectively match the performance of a manually optimized grain size while retaining the scheduling flexibility of finer tasks.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under contract number 63823. The research was performed using PNNL Institutional Computing at Pacific Northwest National Laboratory.

References

[1] D. Lea, "Java specification request 166: Concurrency utilities," 2004.

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

[3] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of OpenMP tasks," *TPDS*, vol. 20, no. 3, pp. 404–418, 2009.

[4] V. A. Saraswat, V. Sarkar, and C. von Praun, "X10: concurrent programming for modern architectures," in *PPOPP*, 2007, p. 271.

[5] "Cilk plus," http://www.cilkplus.org.

[6] P. Haplern, 2012. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3409.pdf

[7] 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.

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

[9] J. Lifflander, S. Krishnamoorthy, and L. V. Kale, "Steal tree: low-overhead tracing of work stealing schedulers." in *PLDI'13*, 2013, pp. 507–518.

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

[11] 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.

[12] 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.

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

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

[15] 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.

[16] 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.

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

[18] 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.

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

[20] 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.

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

[22] G. Zheng, "Achieving high performance on extremely large parallel machines: performance prediction and load balancing," Ph.D. dissertation, UIUC, 2005.

[23] J. Lifflander, S. Krishnamoorthy, and L. V. Kale, "Work stealing and persistence-based load balancers for iterative overdecomposed applications," ser. HPDC '12, 2012, pp. 137–148.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] Y. Sun, G. Zheng, P. Jetley, and L. V. Kale, "An Adaptive Framework for Large-scale State Space Search," in *LSPP '11*, May 2011.

[33] 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.

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

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

[36] 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.

[37] 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.

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