A Transformation Framework for Optimizing Task-Parallel Programs

V. KRISHNA NANDIVADA, IIT Madras
JUN SHIRAKO, JISHENG ZHAO, and VIVEK SARKAR, Rice University

Task parallelism has increasingly become a trend with programming models such as OpenMP 3.0, Cilk, Java Concurrency, X10, Chapel and Habanero-Java (HJ) to address the requirements of multicore programmers. While task parallelism increases productivity by allowing the programmer to express multiple levels of parallelism, it can also lead to performance degradation due to increased overheads. In this article, we introduce a transformation framework for optimizing task-parallel programs with a focus on task creation and task termination operations. These operations can appear explicitly in constructs such as `async, finish` in X10 and HJ, `task, taskwait` in OpenMP 3.0, and `spawn, sync` in Cilk, or implicitly in composite code statements such as `foreach` and `ateach` loops in X10, `forall` and `foreach` loops in HJ, and `parallel` loop in OpenMP.

Our framework includes a definition of data dependence in task-parallel programs, a happens-before analysis algorithm, and a range of program transformations for optimizing task parallelism. Broadly, our transformations cover three different but interrelated optimizations: (1) *finish-elimination*, (2) *forall-coarsening*, and (3) *loop-chunking*. Finish-elimination removes redundant task termination operations, forall-coarsening replaces expensive task creation and termination operations with more efficient synchronization operations, and loop-chunking extracts useful parallelism from ideal parallelism. All three optimizations are specified in an iterative transformation framework that applies a sequence of relevant transformations until a fixed point is reached. Further, we discuss the impact of exception semantics on the specified transformations, and extend them to handle task-parallel programs with precise exception semantics. Experimental results were obtained for a collection of task-parallel benchmarks on three multicore platforms: a dual-socket 128-thread (16-core) Niagara T2 system, a quad-socket 16-core Intel Xeon SMP, and a quad-socket 32-core Power7 SMP. We have observed that the proposed optimizations interact with each other in a synergistic way, and result in an overall geometric average performance improvement between 6.28× and 10.30×, measured across all three platforms for the benchmarks studied.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Optimization; Compilers*; *Parallelism*

General Terms: Algorithms, Performance, Experimentation

This article is an extended and reorganized version of the following two previous articles from conference proceedings [Shirako et al. 2009; Zhao et al. 2010].
The authors gratefully acknowledge support from the IBM Open Collaborative Faculty Awards in 2008 and 2009. This research is partially supported by the Center for Domain-Specific Computing (CDSC), funded by the NSF Expeditions in Computing Award CCF-0926127. The work is also partially supported by the New Faculty Seed Grant, funded by IIT Madras CSE/11-12/567/NFSC/NANV. The POWER7 system at Rice University was supported by a 2010 IBM Shared University Research (SUR) Award as part of IBM's Smarter Planet Initiatives in Life Science/Healthcare and in collaboration with the Texas Medical Center partners, with additional contributions from IBM, CISCO, Qlogic, and Adaptive Computing.
Authors' addresses: V. K. Nandivada, Department of Computer Science and Engineering, IIT Madras, Sardar Patel Road, Kanagam, Chennai, Tamil Nadu 600036 India; J. Shirako, J. Zhao, and V. Sarkar (corresponding author), Department of Computer Science, Rice University, 6100 Main St., Houston, TX 77005; email: vsarkar@rice.edu.

DOI:http://dx.doi.org/10.1145/2450136.2450138

ACM Reference Format:
Nandivada, V. K., Shirako, J., Zhao, J., and Sarkar, V. 2013. A transformation framework for optimizing task-parallel programs. *ACM Trans. Program. Lang. Syst.* 35, 1, Article 3 (April 2013), 48 pages.
DOI: http://dx.doi.org/10.1145/2450136.2450138

1. INTRODUCTION

Two complementary compiler-based approaches for multicore enablement of software are: (1) compilation and optimization of explicitly parallel programs and (2) automatic extraction of parallelism from sequential programs. This article follows the first approach with a focus on task parallelism in programming models such as OpenMP 3.0 [OpenMP 2008], Cilk [Blumofe et al. 1995], Java Concurrency [Peierls et al. 2005], X10 [Charles et al. 2005], Chapel [2005], and Habanero-Java (HJ) [Habanero 2009]. While task parallelism increases productivity by allowing the programmer to express multiple levels of parallelism that may be a natural fit with the underlying algorithm, it can also lead to performance degradation due to increased overheads. In this article, we introduce a transformation framework for optimizing task-parallel programs, with a focus on: (a) reasoning about dependency relations in task-parallel programs and (b) optimizing task creation, termination, and synchronization operations. Experimental results were obtained for a collection of task-parallel benchmark programs written in HJ on three platforms: a dual-socket 128-thread (16-core) Niagara T2 system, a quad-socket 16-core Intel Xeon SMP, and a quad-socket 32-core Power7 SMP. These results show geometric average performance improvements of 6.56×, 6.28×, and 9.77× on the three platforms, respectively, due to the optimizations introduced in this article. For certain benchmarks for which the original versions were highly inefficient, the maximum improvements on these three platforms ranged from 1103.90× to 3935.88×.

In addition to the performance benefits, we believe that this transformation framework can serve as an exemplar for optimizations for future explicitly parallel programs. Optimization of parallel programs is a challenging research area because the historical foundations of code optimization are deeply entrenched in the von Neumann model of sequential computing and have to be reworked for parallelism. As we will discuss, a number of new legality constraints and supporting transformations need to be incorporated in a unified transformation framework to optimize task-parallel programs. Our framework includes a definition of data dependence in task-parallel programs (called happens-before dependence), a static happens-before dependence analysis algorithm, and a host of whole program transformations that help to achieve performance benefits under three broad heads: (a) `finish`-*elimination* to optimize task termination operations, such as `finish` in X10 and HJ, `taskwait` in OpenMP 3.0, and `sync` in Cilk, (b) `forall`-*coarsening* to reduce the task creation and termination overheads incurred by parallel loops present within sequential loops, and (c) *loop-chunking* to derive useful parallel iterations from a given parallel loop specifying the ideal parallelism. These transformations pose interesting challenges in the presence of both data dependence and other synchronization operations. Another interesting challenge comes in the presence of programs that throw exceptions. The analysis and transformations presented in this article can handle all of these challenges. We also introduce a `seq` clause that simplifies writing and optimization of threshold conditions in task creation operations such as `async` spawning. To the best of our knowledge, this is the first such framework to include this set of analyses and transformations for optimizing task-parallel programs.

We now present the motivation behind each of the three categories of optimizations discussed in this article and note some of the underlying challenges.

```
  void sim_village_par(Village vil){
  // Traverse village hierarchy
1: finish {
2:  final Iterator it = vil.forward.iterator();
3:   while (it.hasNext()) {
4:    final Village v=(Village)it.next();
5:    async seq ((sim_level-vil.level) >= bots_cutoff_value)
6:      sim_village_par(v);
     } // while
7:   ... ...;
8:  } // finish:
9:  ... ...
  } // end function
```

Fig. 1. Original code for BOTS Health benchmark.

```
    void sim_village_par(Village vil){
     // Traverse village hierarchy
    1: if ((sim_level-vil.level) < bots_cutoff_value){
    2:  finish {
    3:    final Iterator it = vil.forward.iterator();
    4:    while (it.hasNext()) {
    5:      final Village v=(Village)it.next();
    6:        async sim_village_par(v);
          } /*while*/
          ... ...;} // finish
       } else {
    7:    final Iterator it = vil.forward.iterator();
    8:    while (it.hasNext()) {
    9:     final Village v = (Village)it.next();
    10:    sim_village_par(v);
          }
          ... ...;}
        ... ...
      } // end function
```

Fig. 2. Optimized version of Figure 1.

Finish-elimination. The `finish`-elimination optimization involves eliminating and/or reshaping the `finish` regions to reduce synchronization overhead and improve ideal parallelism. As an example, Figure 1 shows a code fragment from the BOTS Health benchmark [Duran et al. 2009] rewritten in HJ[1]. Each call to method `sim_village_par(v)` contains a `finish` construct spanning lines 1–9. The `async seq` construct in lines 5 and 7 executes the function `sim_village_par(v)` sequentially if condition `(sim_level - vil.level >= bots_cutoff_value)` is `true`, otherwise it creates a child task to invoke `sim_village_par(v)` (see Section 2 for details on HJ syntax). As a result, multiple child tasks created in multiple iterations can execute in parallel with the parent task. The parent task waits at the end of line 9 for all these child tasks to complete since the scope of the `finish` construct ends at line 9. The code fragment in Figure 2 shows the effect of applying `finish`-elimination optimization on the

[1]While HJ is the language used to describe the problem and our solution, the approach described in this article is applicable to any task-parallel language.

```
1:  delta = epsilon+1; iters = 0;
2:  while (delta > epsilon) {
3:    forall (j : [1:n]) {
4:      newA[j] = (oldA[j-1]+oldA[j+1])/2.0;
5:      diff[j] = Math.abs(newA[j]-oldA[j]);
      } // forall
      // sum and exchange
6:    delta = diff.sum(); iters++;
7:    temp=newA; newA=oldA; oldA=temp;
    } // while
```

(a)

```
1:  delta = epsilon+1; iters = 0;
2:  forall (j : [1:n]) {
3:    while (delta > epsilon) {
4:       newA[j] = (oldA[j-1]+oldA[j+1])/2.0;
5:       diff[j] = Math.abs(newA[j]-oldA[j]);
         // sum and exchange
6:       delta = diff.sum(); iters++;
7:       temp=newA; newA=oldA; oldA=temp;
      } // while
    } // forall
```

(b)

```
1:  delta = epsilon+1; iters = 0;
2:  forall (j : [1:n]) {
3:    while (delta > epsilon) {
4:       newA[j] = (oldA[j-1]+oldA[j+1])/2.0;
5:       diff[j] = Math.abs(newA[j]-oldA[j]);
         // sum and exchange
6:       next single {
7:         delta = diff.sum(); iters++;
8:         temp=newA; newA=oldA; oldA=temp;
         } // next single
      } // while
    } // forall
```

(c)

Fig. 3. (a) One-dimensional iterative averaging example; (b) naive `forall`-coarsening may be semantically incorrect; (c) correct coarsening.

example code shown in Figure 1. As can be seen, the number of dynamic `finish` constructs executed in Figure 2 are fewer than in Figure 1 since no `finish` constructs are executed in the `else` part of the code. The impact of this optimization depends on the relative overhead of task termination with underlying runtime scheduling policy such as work-sharing or work-stealing.

Forall-coarsening[2]. To illustrate the challenges in `forall`-coarsening, Figure 3(a) shows the pedagogical one-dimensional iterative averaging program [Chamberlain et al. 2004]. The `forall` loop has an implicit outer `finish` inside which `n` parallel tasks are created to execute the loop body. These `n` tasks terminate and *join* at the end of

[2]In a previous conference submission [Zhao et al. 2010], we referred to the `forall`-coarsening phase as `forall`-*distillation*.

the `forall` loop. These task creations and terminations are repeated in each iteration of the `while` loop, which can result in a large overhead. A naive attempt to move the `forall` header outside the serial loop (as shown in Figure 3(b)) would lead to an incorrect translation: in this example, the original computation outside the `forall` (`sum and exchange`) in Figure 3(a) should be executed only once per iteration of the `while` loop, and only after the termination of the `forall` loop. In the translated program shown in Figure 3(b), the `sum and exchange` code is executed for each iteration of the serial loop, which in turn is executed once for each parallel iteration of the `forall` loop, leading to incorrect semantics. A similar problem would arise if the input program could throw exceptions (see Section 5 for details). Further, the code shown in Figure 3(b) has a data race on `A` and `newA` among the parallel iterations of the `forall` loop and thus needs to be remedied by inserting additional synchronization operations (shown in Figure 3(c)). The `next` statement in this correct translation serves as a barrier with a *single* statement [Yelick et al. 2007] that is guaranteed to be executed by only one task[3]. We present a two-phased approach for `forall`-coarsening: (a) *simple* `forall`-*coarsening* to increase the granularity of synchronization-free parallelism, (b) `forall`-*coarsening with synchronization* to increase the granularity of parallelism that may involve the addition of new synchronization operations.

Loop-chunking. We start with the correctly transformed code after `forall`-coarsening shown in Figure 3(c). The code correctly captures the programmer's original intent. However, if `n` is larger than the number of available hardware threads, this code can incur significant overhead since the barrier synchronization performed by the phaser involves all `n` iterations. As indicated earlier, loop-chunking [Kennedy and Allen 2002] is a standard approach to improve the efficiency of a parallel loop. Figure 4(a) shows the result of performing a chunking transformation mechanically on the `forall` loop, with the goal of decomposing the `forall` loop into chunks of `S` iterations. (The `1:n:S` notation in the new `jj forall` loop is akin to the *low : high : stride* triple notation in Fortran 90 [Metcalfe and Reid 1990].) There has been considerable past work to address the problem of selecting an optimal value of S. The general problem of analytically determining the optimal chunk size of a parallel loop in the presence of overhead and variance was studied by Kruskal and Weiss [1985]. Their approach was extended by Flynn and Hummel [1990] to a sequence of multiple batches, each batch using a progressively smaller chunk size than the previous batch. The idea of using progressively smaller chunk sizes was also advocated by Polychronopoulos and Kuck [1987]. In models like OpenMP [2008], the programmer can guide the implementation by providing chunk policy and chunk size values that can be set dynamically for different platforms. Note that the code transformation in Figure 4(a) is independent of the value of the chunk size, S, and that S can in fact even be set at runtime. Thus, our transformation framework is orthogonal to the problem of selecting the optimal value of S, and we defer to the best-known solutions in practice to address that problem[4].

However, though this chunking transformation is legal for parallel loops that do not contain synchronization operations, it is not legal for the example in Figure 3(c) since it contains a `next` (barrier) operation. In particular, the transformed version Figure 4(a) will attempt to complete all iterations of the `while` loop for iteration `j` before starting iteration `j+1` from the same chunk, and this semantics is different from that of the original code in Figure 3(c). A semantically correct transformed version is shown in

[3]The detailed semantics of next with single statement is described in Section 2.1.1.

[4]If the chunk size is variable, the `1:n:S` triple will have to be replaced by a call to an appropriate runtime iterator.

```
1:   delta = epsilon+1; iters = 0;
2:   phaser ph = new phaser(single);
3:   forall ( point[jj] : [1:n:S] ) phased(single(ph)) {
4:     for (int j = jj ; j <= min(jj+S-1,n) ; j++) {
5:       while ( delta > epsilon ) {
6:         newA[j] = (oldA[j-1]+oldA[j+1])/2.0 ;
7:         diff[j] = Math.abs(newA[j]-oldA[j]);
8:         next single { // barrier with single statement
9:           delta = diff.sum(); iters++;
10:          temp = newA; newA = oldA; oldA = temp;
           } // next single
         } // while
       } // for
     } // finish
                          (a)

1:   delta = epsilon+1; iters = 0;
2:   phaser ph = new phaser(single);
3:   forall ( point[jj] : [1:n:S] ) phased(single(ph)) {
4:     while ( delta > epsilon ) {
5:       for (int j = jj ; j <= min(jj+S-1,n) ; j++) {
6:         newA[j] = (oldA[j-1]+oldA[j+1])/2.0 ;
7:         diff[j] = Math.abs(newA[j]-oldA[j]);
         } // for
8:       next single { // barrier with single statement
9:         delta = diff.sum(); iters++;
10:        temp = newA; newA = oldA; oldA = temp;
         } // next single
       } // while
     } // finish
                          (b)
```

Fig. 4. (a) Naive (incorrect) chunking of the program shown in Figure 3(c); (b) correct chunking.

Figure 4(b). A similar need for careful optimization would arise if the original `forall` loop contained *signal* and *wait* operations instead of *barrier* operations. In general, our optimization pass chunks `foreach` loops, whether one is tightly contained inside a `finish` barrier, such as in `forall` (note: a `forall` can be seen as syntactic sugar for `finish foreach`), or is present standalone.

Combined effect of the different optimizations. The transformations presented in this article can be used in conjunction with each other. To get an understanding of the scope of these transformations together, Figure 5(a) shows an HJ program that first computes elements of a table as an average over its neighbors from previous row. Then, based on a global option, it either processes each element in each row in parallel to compute the `sum` or aggregates the elements of each row in a serial code. After applying `finish`-elimination, `forall`-coarsening, and loop-chunking, the transformed code can be seen in Figure 5(b). Compared to the original code the transformed code has reduced number of barriers, avoids creating useless activities, and extracts useful parallelism from ideal parallelism, resulting in overall efficient code.

An interesting part of this transformation framework is that it is generic in nature and can be used in conjunction with other analyses and optimizations for task-parallel

```
 1: for (i:[1..n] {
 2:    forall (j:[1..1024]) {
 3:      A[i][j] = (A[i-1][j-1] + A[i-1][j] + A[i-1][j+1])/3;
 4:    }
 5:    finish {
 6:      if (!aggregate) {
 7:        foreach (j: [1..1024]) {
 8:          double tmp = processSingle(A[i][j]);
 9:          atomic sum+= tmp;
10:        }
11:      }else {
12:         sum += processAgg (A[i]); // has no escaping asyncs
13:      }
14:    } // finish
15: }
```

(a)

```
 1: forall (j1:[1..16]) { // forall lifted
 2:   for (i:[1..n] {
 3:     for (j2:[1..64]) { // chunked loop
 4:       j = (j1-1) * 64 + j2; // adjusting the index for chunking
 5:       A[i][j] = (A[i-1][j-1] + A[i-1][j] + A[i-1][j+1])/3;
 6:       next; // synchronization
 7:     } } } // forall
 8: if (!aggregate) {
 9:   forall (j1:[1..16]) { // foreach lifted
10:    for (j2:[1..64]) { // chunked loop
11:      j = (j1-1) * 64 + j2; // adjusting the index for chunking
12:      for (i: [1..n]) {
13:        double tmp = processSingle(A[i][j]);
14:        atomic sum+= tmp;
15:      } } } } // forall
16: else {
17:  for (i: [1..n]) {
18:    sum += processAgg (sum[i]); // has no escaping asyncs
19:    } } // else
```

(b)

Fig. 5. (a) HJ program shows the scope of our work; (b) transformed HJ program to show the complexity of the problem. Order of transformations: `finish`-elimination, `forall`-coarsening, loop-chunking.

programs. For instance, the approach introduced in this article could be used as a prepass to optimizations such as *synchronization optimization* [Nicolau et al. 2009].

Contributions. We make the following contributions.

— an iterative algorithm to eliminate redundant `finish` operations that tries to increase ideal parallelism in the program;
— *simple* `forall`-*coarsening*, a transformation scheme that reduces the task creation/termination overhead without introducing any additional synchronization operations;
— `forall`-*coarsening with synchronization*, a more aggressive transformation scheme that replaces task creation/termination operations by lighter-weight barrier synchronizations;

— an iterative algorithm to realize useful parallelism from given specifications of ideal parallelism by chunking parallel loops;
— additional optimizations used to further improve performance as a post-optimization phase. These include Redundant *Next/next Single* Elimination (RNSE), an algorithm to eliminate and strengthen reduction of barrier operations, and loop readjustment that marks loop-exchange operations during prior transformation phases and reverses some of them to improve spatial data locality;
— preservation of exception semantics: the transformation framework presented in this article respects the exception semantics of the HJ language (derived from the `X10` v1.5 exception model [Charles et al. 2005]);
— experimental results. Our framework has been implemented within the HJ compilation system [Habanero 2009] and has been evaluated on three different platforms. The proposed optimizations interact with each other in a synergistic way and overall result in a geometric average performance improvement between 6.28× to 10.30×, measured across all three platforms.

Organization. The rest of this article is organized as follows. Section 2 introduces the HJ parallel programming language that is used in this article as the target of the optimizations. Section 3 presents the basic techniques used in the optimization framework, including the basic program analysis and transformation schemes. Section 4 presents the main optimization framework, and Section 5 gives the details of how to maintain the correct exception semantics during optimization. Section 6 discusses how all the proposed optimizations are integrated in our transformation framework. Section 7 describes how to implement this optimization framework with the HJ compilation system. In Section 8, we present the experimental results collected on three different hardware platforms. Section 9 discusses prior research related to the techniques introduced in this article and finally, we conclude in Section 10.

2. BACKGROUND

2.1. Habanero Java (HJ) Language

Our input programs are written in HJ [Habanero 2009], which extends the earlier Java-based version (v1.5) of the `X10` programming language [Charles et al. 2005] with phasers [Shirako et al. 2008] among other additions and modifications. The scope of this work is limited to the `async`, `finish`, and `isolated` parallel constructs in HJ, thereby making this work applicable to any task-parallel language with primitives for task creation, termination, and mutual exclusion. These constructs are summarized shortly. Following the basic principles of structured programming, these constructs can be arbitrarily nested with each other[5] and with other sequential control-flow constructs in Java.

async. Async is the HJ construct for creating or forking a new asynchronous task. The statement `async` ⟨*stmt*⟩ causes the parent task to create a new child task to execute ⟨*stmt*⟩ asynchronously (i.e., before, after, or in parallel) with the remainder of the parent task. ⟨*stmt*⟩ is permitted to read/write any data in the heap and to read any final local variable in the parent task's lexical environment.

In this article, we introduce an extension to `async` that simplifies programmer-controlled serialization of task creation. The extension takes the form of a `seq` clause with the following syntax and semantics.

```
async seq(cond) <stmt> ≡ cond ? <stmt>: async <stmt>
```

[5]The only exception is that `finish` and `async` are not permitted within an `isolated` statement.

A blocking operation (such as critical section, or barrier operation) inside an `async-seq` statement may lead to undesirable (and sometimes undefined) behavior. We employ a runtime mechanism to ensure that there are no blocking operations inside an `async-seq` statement; otherwise, a runtime exception is thrown.

The main benefit of the `seq` clause is that it removes the burden on the programmer to specify `<stmt>` twice with the accompanying software engineering hazard of ensuring that the two copies remain in sync. In the future, we plan to explore approaches in which the compiler and/or runtime system can select the serialization condition automatically for any `async` statement.

isolated. An isolated statement expresses a global critical section among all tasks. It supports *weak atomicity*, since no mutual exclusion guarantees are enforced between a statement within an isolated block and a statement outside an isolated block. We take inspiration from prior work [Larus and Rajwar 2006] and use the "isolated" keyword instead of "atomic" to make explicit the fact that the construct supports weak isolation rather than strong atomicity. Nesting of isolated statements is permitted but is redundant. HJ prohibits `async` and `finish` statements within an isolated statement. However, isolated blocks may contain loops, conditionals, and other forms of sequential control flow.

finish. The HJ statement `finish` ⟨*stmt*⟩ causes the parent task to execute ⟨*stmt*⟩ and then to wait until all subtasks created within ⟨*stmt*⟩ have terminated, including transitively spawned tasks. Operationally, each instruction executed in an HJ task has a unique *Immediately Enclosing Finish* (IEF) statement instance [Shirako et al. 2008].

An `async` in statement S is considered to be *escaping* [Guo et al. 2009] (also referred to as `e-async`) if it is not enclosed in a `finish` statement within S, that is, if its IEF is not contained within S.

Besides termination detection, the `finish` statement plays an important role with regard to exception semantics. As in `X10`, an HJ task may terminate normally or abruptly. A statement terminates abruptly when it throws an exception that is not handled within its scope; otherwise, it terminates normally. If any *async* task terminates abruptly by throwing an exception, then its IEF statement also terminates abruptly and throws a *MultiException* [Charles et al. 2005] formed from the collection of all exceptions thrown by all abruptly terminating tasks in the IEF. In contrast, in the Java model, an exception is simply propagated from a thread to the top-level console.

foreach. The statement `foreach (point p : R) S` supports parallel iteration over all the points in region `R` by launching each iteration as a separate `async`. A *point* is an element of an n-dimensional Cartesian space ($n \geq 1$) with integer-valued coordinates. A *region* is a set of points and can be used to specify an array allocation or an iteration construct as in the case of `foreach`. For instance, the region `[0:200,1:100]` specifies a collection of two-dimensional points `(i,j)` with `i` ranging from `0` to `200` and `j` ranging from `1` to `100`.

A `foreach` statement does not have an implicit `finish` (join) operation, but its termination can be ensured by enclosing it within a `finish` statement at an appropriate outer level. Further, any exceptions thrown by the spawned iterations are propagated to its IEF instance.

2.1.1. Phasers. In this section, we summarize the *phaser* construct [Shirako et al. 2008] as an extension to X10 *clocks* [Charles et al. 2005]. Phasers integrate collective and point-to-point synchronization by giving each activity (task) the option of registering with a phaser in *signal-only/wait-only* mode for producer/consumer

synchronization or in *signal-wait* mode for barrier synchronization. In addition, a *next* statement for phasers can optionally include a *single* statement (`next {S}`), which is guaranteed to be executed exactly once during a phase transition [Yelick et al. 2007].

These properties, along with the generality of *dynamic parallelism* and the *phase-ordering* and *deadlock-freedom* safety properties, distinguish phasers from synchronization constructs in previous studies including barriers [Gupta 1989; OpenMP 2008], counting semaphores [Sarkar 1988], and X10's clocks [Charles et al. 2005]. Though phasers as described in this article may seem X10-specific, they are a general unification of point-to-point and collective synchronizations that can be added to any programming model with dynamic parallelism such as OpenMP [2008], Intel's Thread Building Blocks, Microsoft's Task Parallel Library, and Java Concurrency Utilities [Peierls et al. 2005].

A *phaser* is a synchronization object that supports the following six operations by an activity A_i.

(1) *new.* When A_i performs a `new phaser(MODE)` operation, it results in the creation of a new phaser *ph* such that A_i is registered with *ph* according to `MODE`. The default mode is *signal-wait*; it includes `signal` and `wait` capabilities, and is used when `MODE` is omitted.
(2) *phased async.* When A_i performs "`async phased` $(ph_1\langle mode_1\rangle, ph_2\langle mode_2\rangle, \ldots)\, A_j$" statement, it creates a child activity A_j registered with a list of phasers with specified modes. If $\langle mode_k\rangle$ is omitted, the same mode as A_i is assumed by default.
(3) *drop.* A_i drops its registration on all phasers when it terminates. In addition, when A_i finishes executing a `finish` statement *F*, it completely deregisters from each phaser *ph* for which *F* is the IEF for *ph*'s creation. This constraint is necessary for the deadlock-freedom property for phasers [Shirako et al. 2008].
(4) *next.* The `next` operation has the effect of advancing each phaser on which A_i is registered to its next phase, thereby synchronizing all activities registered on the same phaser. The semantics of `next` is equivalent to a `signal` operation followed by a `wait` operation. The exception semantics for the single statement was unspecified [Shirako et al. 2008]. We define the exception semantics of the single statement as follows: an exception thrown in the single statement that causes all the tasks blocked on that `next` operation to terminate abruptly with a single instance of the exception thrown to the IEF task.[6]
(5) *signal.* A `signal` operation by A_i is shorthand for a `ph.signal()` operation performed on each phaser *ph* on which A_i is registered with `signal` capability. Note that *ph* will advance to its next phase when all activities registered on *ph* with `signal` capability perform `ph.signal()` operations.
(6) *wait.* A `wait` operation by A_i is a blocking operation to wait for all phasers on which A_i is registered with `wait` capability to advance to the next phase. Note that a `wait` operation is always performed as the latter part of a `next` operation and hence does not cause any deadlock.

forall. HJ introduces `forall` $\langle stmt\rangle$ as syntactic sugar for "`finish{ ph=new phaser(SIG_WAIT_NEXT); foreach phased(ph)` $\langle stmt\rangle$`}`". The scope of the phaser `ph` is limited to the implicit `finish` in the `forall`, and thus the parent task will drop its registration on `ph` after all the iterations of `forall` are created.

[6]Since the scope of a phaser is limited to its IEF, all tasks registered on a phaser must have the same IEF.

2.2. Classical Loop Transformations

This section briefly summarizes some classical loop restructuring techniques that have historically been used to improve parallelism and data locality, and expose other opportunities for compiler optimization [Kennedy and Allen 2002; Wolfe 1996].

— *Strip mining* is a loop transformation that replaces a single loop with two nested loops with smaller segments. This restructuring is an important preliminary step for vectorization, tiling, SIMDization, and other transformations for improving locality and parallelism.
— *Loop interchange* results in a permutation of the order of loops in a perfect loop nest and can be used to improve data locality, coarse-grained parallelism, and vectorization opportunities.
— *Loop distribution* divides the body of a loop and generates several loops for different parts of the loop body. This transformation can be used to convert loop-carried dependences to loop-independent dependences, thereby exposing more parallelism.
— *Loop unswitching* is akin to interchanging a loop and a conditional construct. If the condition value is loop invariant, it can be moved outside so that it is not evaluated in every iteration.
— *Loop fusion* is the inverse of loop distribution. It merges two loops to generate a loop with a single header. This transformation can also help improve data locality, coarse-grained parallelism, and vectorization opportunities.

The legality constraints for these transformations are well understood for cases in which the input program is sequential. In Sections 3.2 and 5, we show how these transformations can be extended in the context of task-parallel programs in the presence of synchronizations and exceptions.

2.3. Program Structure Tree

Agarwal et al. [2007] introduced a program representation called a Program Structure Tree (PST) which statically represents the parallelism structure of a single procedure. A PST for a procedure in a program is a rooted tree (N, E), where:

— the set N of nodes can have the following types: `root, statement, loop, async, finish`, and `isolated`. The `root` type corresponds to the start of the procedure, and the `statement` type corresponds to all other statements except `loop, async, finish`, and `isolated`;
— the set E contains edges resulting from reducing the abstract syntax tree of the procedure into the types listed before.

We present a Program Structure Graph (`PSG`) as an extension of `PST` to represent the whole program by incorporating call graph information. A program structure graph is given by a rooted graph (N, E), where a node in the set N may have the additional types: `function` and `call`, besides the types of the nodes of `PST`. Similarly, we extend the edge set by admitting optional labels `call` ⟨`context`⟩ and `return` ⟨`context`⟩; intuitively, `context` contains the calling context. For the sake of this presentation, we assume that each `finish` statement is represented as a pair of nodes in the `PSG`: begin-finish and end-finish.

3. BASIS OF OUR TRANSFORMATION FRAMEWORK

In this section, we present three fundamental instruments of our transformation framework: advances in program analysis techniques for task-parallel programs, extensions to traditional loop transformations in the context of task-parallel programs,

and a set of new transformations in task-parallel programs presented as variations of some of the traditional optimizations in the context of parallel constructs. We start the section by discussing two aspects of program analysis for task-parallel programs: data dependence and happens-before dependence analysis. We follow it up with two different sets of program transformation primitives that are inspired from many traditional program transformation techniques. To simplify the presentation, we first focus on the restricted case where the input code is known to be exception free. Later in Section 5, we discuss the more general case involving exceptions. We use HJ as the target language for describing the programs and the transformations there on. However, the specified transformations can be applied in other similar task-parallel languages (such as X10, OpenMP, Cilk, and so on).

3.1. Data Dependence in Task-Parallel Programs

Legal program transformation requires the preservation of the order of ordered interfering memory accesses in the input program. Data-dependence analysis has traditionally enforced this requirement and to maintain the legality of transformations of sequential programs. Modern optimizing compilers use data-dependence analysis for various program analysis and transformations, including loop transformations and automatic parallelization [Kennedy and Allen 2002; Wolfe and Banerjee 1987]. However, dependence analysis is more challenging in the context of task-parallel languages since parallel language constructs, such as `async`, impact which pairs of interfering data accesses should be treated (or not) as data dependences.

Another aspect of parallel language semantics that impacts the legality of program transformations is the *memory consistency model*. The data-dependence framework introduced in this article can be viewed from two perspectives. From the perspective of a strong memory model such as sequential consistency [Lamport 1979], this framework only specifies transformations that are legal for data-race-free programs. In this case, our framework would be applicable to memory models such as that proposed for C++ in which the behavior of programs with data races is undefined. From the perspective of a weak memory model, such as location consistency [Gao and Sarkar 2000], this framework specifies transformations that are legal for all programs whether or not they exhibit data races.

3.1.1. Dynamic Happens-Before Dependence. In this section, we extend the classical definition of data dependence in sequential programs to *happens-before dependence* in parallel programs. We begin by adapting the definition of a happens-before relation (`HB`) of Lamport [1978] to a dynamic execution of an HJ program. Specifically, the relation `HB` on instances I_A and I_B of statements A and B is the smallest relation satisfying the following conditions.

(1) *Sequential order*. If I_A and I_B belong to the same task, and I_B is sequentially control or data dependent on I_A, then $HB(I_A, I_B) = true$.
(2) *Async creation.* If I_A is an instance of an `async` statement, and I_B is the corresponding instance of the first statement in the body of the `async`, then $HB(I_A, I_B) = true$.
(3) *Finish termination.* If I_A is the last statement of an `async` task, and I_B is the end-finish statement instance of I_A's Immediately-Enclosing-Finish (IEF) instance, then $HB(I_A, I_B) = true$.
(4) *Isolated.* All instances of interfering isolated blocks in a dynamic execution of an HJ program can be assumed serialized in some total order. If I_A is the last statement in an isolated block instance, and I_B is the first statement of the next isolated block instance in the total order, then $HB(I_A, I_B) = true$.
(5) *Transitivity.* If $HB(I_A, I_B) = true$ and $HB(I_B, I_C) = true$ then $HB(I_A, I_C) = true$.

```
// Before loop distribution
for (int i = ...) {
  /* S1 */  X[f(i)] = ... ;
  async { /* S2 */  ...
= X[g(i)]; }
}
```

$\Longrightarrow$

```
// After loop distribution
for (int i = ...)
  /* S1 */  X[f(i)] = ... ;
for (int i = ...)
  async { /* S2 */  ... = X[g(i)]; }
```

Fig. 6. Loop distribution example.

Given the dynamic *HB* relation, we define a dynamic *happens-before dependence* relation HBD on statements A and B as follows. We say that HBD(A,B) = *true* if there is a possible execution of the program with instances I_A and I_B of statements A and B that satisfies all of the following conditions.

(1) $HB(I_A, I_B) = true$,
(2) I_A and I_B access the same location X and at least one of the accesses is a write, and
(3) there is no statement instance I_C in the same execution that writes X such that $HB(I_A, I_C) = true$ and $HB(I_C, I_B) = true$.

As with dependence analysis of sequential programs, we classify the dependence as *flow*, *anti*, and *output* when the accesses performed by I_A and I_B are read-after-write, write-after-read, and write-after-write, respectively. Further, the HBD relation can be qualified by restricting the sets of instances participating in the dependence akin to direction vectors and distance vectors in sequential programs. It should be easy to see that the HBD relation degenerates to sequential data dependences when the input program is sequential. Also, as with sequential data-dependence analysis, any HBD analysis performed by a compiler is necessarily conservative to guarantee soundness, that is, the analysis must err on the side of stating that HBD(A,B) = *true* when it is unsure of the dependence relation. Thus, HBD is a "may dependence" analysis.

We conclude this section with a discussion of HBD analysis on the example code fragment in Figure 6. We have a flow dependence from $S1$ to $S2$ on variable X with direction vector $(\leq)$ assuming that the subscript functions $f(i)$ and $g(i)$ are unanalyzable by the compiler. While a sequential compiler would also report a loop-carried antidependence from $S2$ to $S1$ with direction vector $(<)$, no such dependence occurs in the parallel case according to the definition of HBD since no execution of the code fragment can result in instances I_{S_1} and I_{S_2} of statements S_1 and S_2 such that $HB(I_{S_2}, I_{S_1}) = true$. Thus there is no dependence cycle that includes $S1$ and $S2$. As a result, loop distribution can be performed on $S1$ and $S2$ as shown in Figure 6 even though loop distribution would be illegal in the sequential case.

3.1.2. Computation of Happens-Before Dependence. We now present a scheme to compute the happens-before dependence information based on the static happens-before information. It involves a two-phase process. We first present a conservative constraint-based algorithm to compute may-happen-before information as a set *MHB* of pairs: if $(N_1, N_2) \in MHB$, then N_1 may happen before N_2. We use $N_1, N_2 \cdots$ (with numeric subscripts) to denote nodes in the PSG, corresponding to the static statements rather than the dynamic instances. In the second phase, we propagate the may-happens-before information introduced by the `isolated` statements.

Phase 1. We generate a set of constraints to compute static happens-before information in Figure 7. Note the following points pertinent to the constraints: (a) statically each `async` statement has a set of one or more IEFs in the PSG; and (b) unlike a dynamic instance of a statement, a static instance can have more than one possible last statement.

Phase 1
For each $N_1, N_2 \in Nodes$:

(1) if N_1 and N_2 are in the same activity, and N_1 is to the left of N_2 in G, then $(N_1, N_2) \in MHB$;
(2) if N_1 and N_2 are in the same activity, and both the nodes have a common loop node as one of their ancestors in G, then $\{(N_1, N_2), (N_2, N_1)\} \subseteq MHB$;
(3) if N_1 is an `async` statement and N_2 is the first statement in that `async`, then $(N_1, N_2) \in MHB$;
(4) if N_1 is one of the last statements of an `async` statement and N_2 is the end-finish statement of one of the IEF of the `async` statement, then $(N_1, N_2) \in MHB$;
(5) if $\exists N_3 \in Nodes$, $(N_1, N_3) \in MHB$ and $(N_3, N_2) \in MHB$, then (N_1, N_2) should also be added to MHB for transitivity i.e., $(N_1, N_2) \in MHB$.

Phase 2

(1) For each $N_1, N_2 \in Nodes$: if N_1 and N_2 are `isolated` statements and $(N_1, N_2) \notin MHB$ and $(N_2, N_1) \notin MHB$, then add both (N_1, N_2) and (N_2, N_1) to MHB i.e., $\{(N_1, N_2), (N_2, N_1)\} \subseteq MHB$;
(2) Generate and solve the following constraints: For each $N_1, N_2 \in Nodes$: if $\exists N_3 \in Nodes$, $(N_1, N_3) \in MHB$ and $(N_3, N_2) \in MHB$, then add (N_1, N_2) to MHB for transitivity i.e., $(N_1, N_2) \in MHB$.

Fig. 7. Constraints to compute static happens-before information. First, solve the constraints generated from Phase 1 to compute a first cut of *MHB* without taking into consideration the isolated statements ; then execute the Phase 2.

We solve these constraints to generate the set *MHB* (which contains only partial may-happen-before information, as this phase does not take into consideration the `isolated` statements). In general, the happens-before information may also contain a condition vector (akin to direction and distance vectors), giving the conditions under which the relation may hold. In such a case, each element of the set *MHB* will be a three tuple where the third element is the condition vector. A discussion on such precise happens-before information is left for future work.

Phase 2. After we have obtained the partial may-happen-before information in the first phase, we use a two-step process to update the set *MHB*, to include the happens-before relation introduced by the `isolated` statements. Step 1: For each pair of `isolated` statements, we introduce a commutative may-happen-before relation, if they are not already ordered. Step 2: We introduce constraints to address the transitive may-happen-before relation and solve them.

Now we summarize the algorithm to compute static happens-before dependence (which we call the may-happen-before dependence), based on the *MHB* information. For any two nodes N_1 and N_2, we say that N_2 has a may-happen-before dependence on N_1, denoted by $\texttt{MHBD}(N_1, N_2) = \mathit{true}$, if: (i) $(N_1, N_2) \in \mathit{MHB}$, (ii) N_1 and N_2 access the same variable or storage location and one of the access is a write, (iii) $\neg\exists N_3 \in \mathit{Nodes}$: $\texttt{MHBD}(N_3, N_1) = \mathit{true}$ and $\texttt{MHBD}(N_2, N_3) = \mathit{true}$. As an illustration, for the code snippet shown in Figure 8(a), a subset of the elements from the *MHB* set, and the complete `MHBD` set for each variable are shown in Figure 8(b) and Figure 8(c), respectively.

3.2. Extensions to Traditional Loop Transformations

In this section, we present some extensions to the traditional loop transformation techniques in the presence of task-parallel programs. These transformations will be used later to derive more complex program optimization techniques. Figure 9 presents some of our extensions to the traditional loop transformations in the context of task-parallel programs. The comments under each rule shown in Figure 9 act as the preconditions that need to be satisfied for the rule to be applied. An `e-async` is an escaping `async` as defined in Section 2. A statement is considered to be side-effect free if: (a) it does not update any variable whose value is visible after the execution of the statement, and

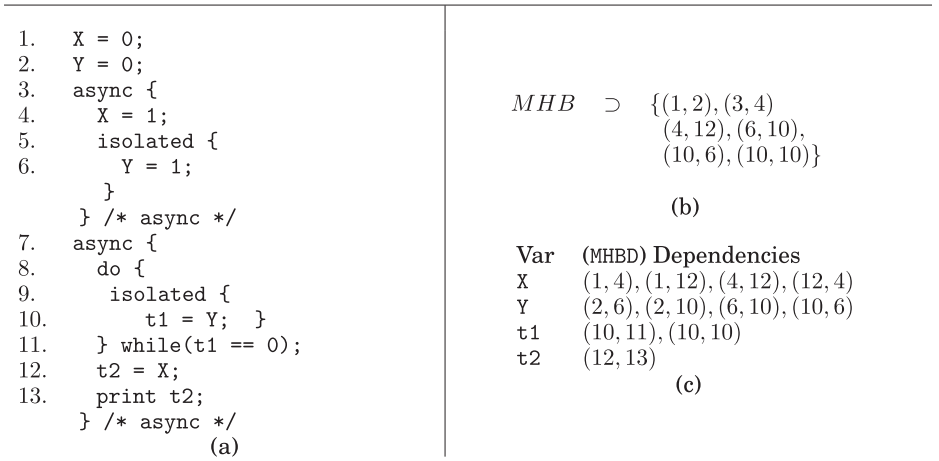

Fig. 8. Example to illustrate may-happen-before dependence. (a) input program with labels; (b) some illustrative elements of the MHB set; and (c) may-happen-before dependences.

(b) it does not have an `e-async`. The dependence relations mentioned in the preconditions refer to the may-happen-before dependence relations discussed in Section 3.1.2.

These preconditions are required for semantically correct translation. For instance, for the *serial loop distribution* (rule 1) to be correct, there should be no dependence cycle between `S1` and `S2`. While the rest of the rules are different extensions to traditional loop transformation techniques, the first rule (1) and the last rule (10) are the exact traditional loop distribution and loop unswitching rules [Muchnick 1997] reproduced in this article for completeness. It may be noted that even though we use the `for` loop to describe many of the rules, it is also applicable to other loops (such as `while` and `do-while`). We now discuss a few of the transformation rules.

Unlike serial loop distribution, *parallel loop distribution* (rule 2) does not require any dependence testing and thus has no preconditions. It builds on a well-known observation that a parallel loop can always be fully distributed [Kennedy and Allen 2002] since a loop-carried dependence is needed to create a distribution-preventing cycle. Hence the `forall` loops can be fully distributed. The implicit `finish` operations in `forall` ensure the correctness of the resulting transformation. As in classical serial loop distribution, it may be necessary in some cases to perform *scalar expansion* [Kennedy and Allen 2002] on any iteration-private scalar variables that may be accessed in both `S1` and `S2`.

Rule 3 (*loop/finish interchange*) increases the scope of a `finish` construct, and it can do so only if there are no dependencies between the escaping asyncs in `S3` and the body of the serial `for` loop.

The *serial-parallel loop interchange* (rule 4) has similarities to the traditional loop parallelization rule [Kennedy and Allen 2002]. Rule 5 (*parallel-serial loop interchange*) builds on a well-known observation from classical vectorization: "a loop that carries no dependences cannot carry any dependences that prevent interchange with other loops nested inside it" [Kennedy and Allen 2002]. Though this observation was developed for sequential loops that are parallelizable, it is just as applicable to parallel `forall` loops. Thus, the interchange in rule 5 can be performed without the need for checking any data dependences. For simplicity, we assume that the inner sequential loop's iteration space, `R2`, is independent of the outer `forall` loop's index variable. Extension of this rule to support interchange of trapezoidal loops should be straightforward as in past work on loop interchange in sequential programs [Kennedy and Allen 2002]. We also

assume that the loop body S does not contain any break or continue statements; support for these statements is more complicated but can be built on the exception support in Section 5.

Loop unpeeling (rule 6) expands the scope of a forall loop by adding the statement S2 to the body of the loop; S2 is executed as a next-single statement. This rule assumes that S2 does not have *break or continue* statements.

Loop fusion (rule 7) builds on the classical loop fusion transformation for sequential code [Kennedy and Allen 2002]. It merges two forall statements by fusing their bodies and inserting a next (barrier) statement. Both of these rules (unpeeling and fusion) use the implicit phaser associated with forall.

Loop switching (rule 8) is based on the inverse of classical loop unswitching transformation discussed in Section 4.2. It expands the scope of the forall loop by bringing an if statement inside the body of the loop.

Rule 9 (*parallel loop unswitching*) builds on the classical unswitching transformation for sequential code [Kennedy and Allen 2002] (also shown in rule 10). The main assumption here is that the condition e is independent of the forall loop's index variable.

3.3. Variations of Traditional Transformations with Parallel Constructs

In this section, we discuss some new transformations, presented as a variation to the traditional (non-loop) program transformation techniques in the presence of task-parallel programs. These transformations, along with the ones presented in Section 3.2, will be used later to derive complex program optimization techniques.

Figure 10 presents some of our extensions to the traditional program (non-loop) transformations in the context of task-parallel programs. The comments under each rule act as the preconditions that need to be satisfied for the rule to be applied. For instance, in the *finish distribution* (rule 1), if S1 contains an e-async, then the translation may be incorrect. We now present some details for the rest of the rules.

Redundant finish elimination removes the redundant finish around a forall statement that has no e-asyncs. The *tail finish elimination* (rule 5) applies to all the variants of *tail* finish statements, such as the finish statement occurring as the last statement of an e-async statement or as the last statement of a tail if-then block or else block. The *finish fusion* (rule 6) expands the scope of the finish block, provided there is no dependence between the e-asyncs of S1 and S2.

All these rules apply to both intraprocedural and interprocedural contexts. In an interprocedural context, we may have to do some code replication to maintain the program semantics. Figure 11 presents a sample rule in the interprocedural context.

3.4. Correctness Guarantees

In this section, we present an argument on the semantics-preserving nature of the transformations presented in the article. We state it in terms of a theorem on the semantics-preserving nature of any optimization phase that consists of applying one more instance of transformation rules presented in Figure 9 and Figure 10.

We first present a specialization of the may-happen-before dependency introduced in Section 3.1.

Definition 3.1. For a given variable (or storage location) v and any two nodes I_1 and I_2, we say that $\texttt{MHBD}_V(I_1, I_2, v) = \mathit{true}$ if: (i) $\texttt{MHBD}(I_1, I_2) = \mathit{true}$, (ii) I_1 and I_2 both access v and one of the access is a write, and (iii) $\neg\exists I_3 \in \mathit{Nodes}$: $\texttt{MHBD}_V(I_1, I_3, v) = \mathit{true}$ and $\texttt{MHBD}_V(I_3, I_2, v) = \mathit{true}$.

We now present a definition for semantics preservation for transformations that ensure that each source AST node can be found at one or more places in the target

1. Serial loop distribution:

```
for (...) { S1;S2; }
// no dependence cycle between S1 & S2
```
$\Longrightarrow$
```
for (...) {S1;}
for (...) {S2;}
```

2. Parallel loop distribution:

```
forall (point p : R1)
  { S1; S2; }
// S1 has no dependence on S2
```
$\Longrightarrow$
```
forall (point p : R1) S1;
forall (point p : R1) S2;
```

3. Loop/Finish interchange:

```
for (S1;cond;S2)
  finish S3;
// Say Es = set of e-asyncs in S3
// ¬∃e ∈ Es: cond has dependence on e
// ¬∃e ∈ Es:body of e has loop
//      carried dependence on S2, cond or S3
```
$\Longrightarrow$
```
S1;
finish
  for (;cond;S2)
    S3;
```

4. Serial-parallel loop interchange:

```
for (i: [1..n])
  forall (point p : R1)   S;
// iterations of the for loop are independent.
// R1 does not depend on i
```
$\Longrightarrow$
```
forall (point p : R1)
  for (i: [1..n])
    S;
```

5. Parallel-serial loop interchange:

```
forall (point p : R1)
  for (point q : R2)  S
// R2 is independent of p
// S contains no break/continue
```
$\Longrightarrow$
```
for (point q : R2)
  forall (point p : R1)
    S
```

6. Loop unpeeling:

```
forall (point p: R) S1;
S2;
// no break/continue in S2.
// Say Es = set of e-asyncs in S1
// ¬∃e ∈ Es: S2 has dependence on e
```
$\Longrightarrow$
```
forall (point p: R)
  {S1; next S2;}
```

7. Loop fusion:

```
forall (point p: R1) S1;
forall (point p: R2) S2;
// Say Es = set of e-asyncs in S1
// ¬∃e ∈ Es: S2 has dependence on e
```
$\Longrightarrow$
```
forall (point p: R1||R2)
 {if (R1.contains (p)) S1;
  next;
  if (R2.contains (p)) S2;}
```

8. Loop switching:

```
if (c)
  forall (point p: R)
    S;
```
$\Longrightarrow$
```
final boolean v = c;
forall (point p: R)
  if (v)  S;
```

9. Parallel loop unswitching:

```
forall (point p : R1)
  if (e) S
//e is a pure function and is independent of p
```
$\Longrightarrow$
```
if (e)
  forall (point p : R1)  S
```

10. Serial loop unswitching:

```
  for(S2;cond1;S3){
    if (cond2) S4;  else S5;
  }
// cond2 has no dependence
//      on S2,S3,S4 and S5,
// cond2 has no side effects
```
$\Longrightarrow$
```
  if (cond2) {
    for(S2;cond1;S3) S4;
  } else {
    for(S2;cond1;S3) S5;
  }
```

Fig. 9. Extending traditional loop transformations for task-parallel programs.

AST; extending the argument to the PSGs, for a given source PSG node I_1, we will assume that the set $T(I_1)$ gives the corresponding set of nodes in the target PSG.

Definition 3.2. A transformation of a parallel program is *semantics-preserving* if the set of happens-before dependencies of all the variables at all program points in the source program are conservatively preserved in the translated program; that is, in the

1. Finish distribution:

```
finish { S1; S2; }                    ⟹   { S1;
// S1 has no e-asyncs.                       finish { S2; }
```

2. Finish unswitching:

```
finish                                ⟹   { if (cond) finish S1;
  if(cond)S1; else S2;                       else finish S2;
// cond has no e-async
```

3. If expansion:

```
finish {                              ⟹   { finish {
  S1;                                        if (cond)
  if(cond) S2; else S3;                        {S1; S2; S4;}
  S4; }                                      else
// no dependence between cond and S1            {S1; S3; S4}
                                           }
```

4. Redundant finish elimination:

```
finish S;                             ⟹   { S;
// S has no e-async.
```

5. Tail finish elimination:

```
finish { S1;finish S2;}               ⟹   { finish {S1; S2; }
```

6. Finish fusion

```
finish S1;                            ⟹   { finish{
finish S2;                                   S1;
// Say E_s = set of e-asyncs in S1           S2;
// ¬∃e ∈ E_s: S2 has dependence on e      }
```

Fig. 10. Variations of traditional transformations for programs with parallel constructs.

Inter-proc Finish unswitching

```
finish {                                 if (cond)
  S0; foo(); S5 }                          finish{S0;foo1();S5;}
void foo() {                             else
  S1;                                      finish{S0;foo2();S5;}
  if(cond)S2;else S3;            ⟹      foo1() {
  S4 };                                    S1; S2; S4; }
// cond has no e-async                   foo2() {
// cond has no dependence on S0, S1         S1; S3; S4; }
```

Fig. 11. Interprocedural finish unswitching.

source program given a node I_1 in the PSG, a variable v, and a set S of nodes such that $\forall I_k \in S : \texttt{MHBD}_V(I_1, I_k, v) = true$, then in the target program, $\forall I_2 \in T(I_1), \forall I_j \in T(I_k) : \texttt{MHBD}_V(I_2, I_k, v) = true$.

LEMMA 3.3. *The preconditions for each rule shown in Figure 9 and Figure 10 ensure that the individual transformation resulting from each of the rules is semantics-preserving.*

PROOF. *(Sketch)*

We present a sketch for the proof of the transformations involving parallel constructs only; the proof for the traditional serial transformations (see the ones prefixed "Serial" in Figure 9) are skipped here. Before we proceed to the details, we bring to the notice of the reader that because of the chosen memory model for any given activity A, as seen by the other parallel activities, there is no assumed order among the instructions

of activity A. Thus, if a transformation does not introduce any new activities or modify the MHP information, then the transformation is semantics-preserving, provided the dependencies among the rest of the statements are preserved.

— (*Rule 2 in Figure 9, parallel loop distribution*). The transformation does not introduce any new dependence or any change in the MHP information in the program. The rule does introduce a new statement the second `forall` statement, but it does not modify the happens-before dependence relations.
— (*Rule 3 in Figure 9 loop/finish interchange*). Though this transformation increases the scope of activities created in `S2`, since the different asynchronous tasks created in `S2` have no dependence on different iterations of `S2`, the transformation does not affect the happens-before dependence relations of any source variable.
— (*Rule 4 in Figure 9, serial-parallel loop interchange*). The explanation for this rule is quite similar to the previous rule.
— (*Rule 5 in Figure 9, parallel-serial loop interchange*). Although the transformation reduces the scope of the activities created in the `forall` loop, it does not modify the happens-before dependence relation between any statements. While the order among the `forall` and `for` loops is indeed interchanged, there is no happens-before dependence relation between these statements.
— (*Rule 6 in Figure 9, loop unpeeling*). Because of the transformation, some of the e-asyncs present in `S1`, which in the source code terminate before `S2`, may run in parallel with `S2`. But the preconditions set ensure that there is no happens-before dependence between `S2` and these `e-asyncs`.
— (*Rule 7 in Figure 9, loop fusion*). The explanation for this rule is similar to the previous rule.
— (*Rule 8 in Figure 9, loop switching*). The explanation is trivial, considering that the evaluation of the predicate still happens before the `forall` statement.
— (*Rule 9 in Figure 9, parallel loop unswitching*). The precondition ensures that `e` is a pure expression with no side-effects, and has no dependence on `p`. Thus, unswitching the loop makes no difference to the happens-before dependence relations.
— (*Rule 1 in Figure 10, finish distribution*). The transformation does not change sequential program order. Since this rule is applied only if `S1` has no `e-asyncs`, there is no change in the MHP information, and the happens-before dependence does not change either.
— (*Rule 2 in Figure 10, finish unswitching*). The transformation does not change sequential program order. Since this rule is applied only if `cond` has no `e-asyncs`, there is no new MHP relation and the happens-before dependence does not change.
— (*Rule 3 in Figure 10, if expansion*). This is a trivially correct serial transformation involving code duplication.
— (*Rule 4 in Figure 10, redundant finish elimination*). `S` has no happens-in-parallel relation with any of the code after the `finish` closure, since it does not contain `e-asyncs`. Eliminating the `finish` does not violate any happens-before relation. Further, the elimination of the `finish` does not affect the order of execution of `S`.
— (*Rule 5 in Figure 10, tail finish elimination*). `S1` either happens-before or happens-in-parallel with `S2`, and eliminating the tail `finish` does not violate these relations. Thus there is no change in the happens-before dependence relations.
— (*Rule 6 in Figure 10, finish fusion*). `S1` and `S2` have no dependence, thus `finish S1` and `S2` can be exchanged without violating any dependence relations between statements in `S1` and `S2`. Since `S2` accesses no shared variables, moving `S2` before `finish S1` does not impact the happens-before dependence relations between the statements of `S2` and statements present in other parallel activities. Note that the

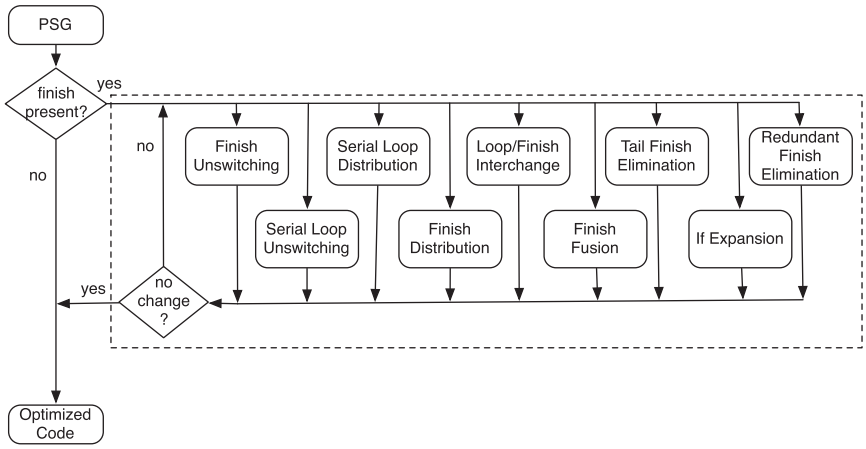

Fig. 12. Block diagram of the `finish`-elimination phase.

translation ensures that `S3` starts only after `S1` has terminated, as was the case in the input program. □

THEOREM 3.4. *Any optimization pass consisting of applying one or more instances of the rules shown in Figure 9 and Figure 10 is semantics-preserving.*

PROOF. Follows directly from the Lemma 3.3. □

4. NEW OPTIMIZATIONS FOR TASK-PARALLEL PROGRAMS

In this section, we discuss the details of our transformation framework to optimize task-parallel programs by presenting three new program optimizations. We use the basic infrastructure developed in Section 3 to develop new program optimization techniques for task-parallel programs written in HJ. It may be noted that these optimizations can be applied in other similar task-parallel languages as well (such as X10, OpenMP, and Cilk). These new optimizations are, namely, `finish`-*elimination*, `forall`-*coarsening*, and *loop-chunking*.

4.1. Finish-Elimination

In this section, we introduce a transformation technique to reduce the number of dynamic `finish` operations performed by an HJ program. The same framework should apply (with some adaptations) to optimizing termination operations in other languages such as OpenMP's `taskwait` and Cilk's `sync`.

The basic insight behind `finish`-elimination is that a `finish` statement is redundant if its body has no *escaping asyncs*. Our transformation technique is based on an iterative algorithm that incrementally optimizes the program.

4.1.1. Finish-Elimination Algorithm. We now present a new compiler optimization phase called *iterative finish-elimination* that depends on the happens-before dependence analysis discussed earlier (see Section 3.1).

Figure 12 shows the block diagram of our `finish`-elimination phase. Before the entry to this optimization pass, we first build the program structure graph (defined in Section 2.3). We then invoke the iterative finish-elimination algorithm on the root node of the graph; it performs a post-order traversal of the `PSG` and recursively invokes the rules shown in the block diagram (explained in Section 3). We repeatedly apply redundant finish elimination, tail finish elimination, finish fusion, loop/finish interchange,

Transformation	Input code	Profitability constraint
1. Finish distribution	`finish { S1; S2; }`	S2 *has* e-asyncs.
2. For loop distribution	`for (...) { S1;S2; }`	S1 or S2 *has* e-async
3. Finish unswitching	`finish` `  if(cond)S1; else S2;`	S1 or S2 *has* e-async

4. If expansion	`finish {`
	`  S1;`
	`  if(cond) S2; else S3;`
	`  S4; }`
		S1 *has* e-async.
		S2 *or* S3 has e-async

5. Loop unswitching	`finish {`
	`  S1`
	`  for(S2;cond1;S3){`
	`  if (cond2) S4;`
	`  else S5; }// for`
	`  S6 } // finish`
		S4 *or* S5 has e-async

6. Loop/Finish interchange	`for (S1;cond;S2)` `  finish S3;`	*The set of e-asyncs in* S3 *is not empty*
7. Tail finish elimination	`finish { S1;finish S2;}`	S1 and S2 *have* e-async.
8. Finish fusion	`finish S1;` `finish S2;`	S1, *and* S2 *have* e-async.

Fig. 13. Profitability constraints for iterative `finish`-elimination.

finish distribution, serial loop distribution, finish unswitching, if expansion, and serial loop unswitching. We continue the iterative process until either no further change is possible, or there is no parallel code left in the body of the `finish` node. These subtransformations are monotonic in nature and can be applied in any order. After each successful invocation of a rule on a node n, the program structure is changed and the PSG needs to be updated; it is sufficient to rebuild the subtree rooted at the parent node of n.

In addition to the correctness requirements of these transformation rules (shown as comments on the rules in Figure 9 and Figure 10), the rules are applied only if the profitability requirement is also satisfied. For each of the transformations, the profitability requirements are shown in Figure 13.

We now present the effect of invoking the `finish`-elimination algorithm on the running example shown in Figure 1 (reproduced in Figure 14(a)). There are some omitted shared heap accesses in the code in Figure 14(a) line 9. Thus, because of the possible concurrent data dependence, the `finish` node cannot be eliminated (Figure 14(a) line 1). Now the compiler expands `async seq` to an `if-then-else` statement and applies *if expansion* (rule 3, Figure 10). Next, it applies loop unswitching, if expansion, finish unswitching, and redundant finish-elimination to derive the optimized code. Before applying the redundant finish-elimination rule, the compiler checks that the body of the inner finish has no `e-async` within (precondition (1)); it does so by analyzing the body of finish, which involves analyzing the invoked function `sim_village_par`.

4.2. Forall-Coarsening

In this section, we present our transformation framework to reduce task creation and termination overhead. We introduce a new compiler optimization phase called `forall`-coarsening. In the HJ program snippet shown in Figure 15(a), the `forall` loop inside a `for` loop (with m number of iterations) results in creation of $m \times n$ number of tasks, with each of the n tasks waiting on a `finish`. The main goal of our translation is to generate coarse-grained `forall` statements that encompass the surrounding `for` loops and `while` loops. Depending on the actual program code, different translations are possible; Figure 15(b) and Figure 15(c) show two translations that coarsen the `forall` loop in Figure 15(a). We call the first translation *simple* `forall`-*coarsening* and the second one `forall`-*coarsening with synchronization*. While both translations are

```
// Input program.
void sim_village_par(final Village vil){
1:finish {
2: final Iterator it=vil.iterator();
3:  while (it.hasNext()) {
4:    final Village v=(Village)it.next();
5:    async seq ((sim_level - vil.level)
6:                >= bots_cutoff_value)
7:      sim_village_par(v);
8:  } // while
9:  ... ...;
10:} // finish:
11:... ...  }
```

(**a**)

```
// After if expansion
 void sim_village_par(final Village vil) {
1:finish {
2: final Iterator it=vil.iterator();
3:  while (it.hasNext()) {
4:    if ((sim_level - vil.level)
5:        < bots_cutoff_value)
6:      final Village v = (Village)it.next();
7:      async sim_village_par(v);
8:     else {
9:      final Village v = (Village)it.next();
10:      sim_village_par(v);
11:      } } // while
12:  ... ...;
13:} /*finish*/ ... ...  }
```

(**b**)

```
// After Loop Unswitching
void sim_village_par(final Village vil) {
1:finish {
2: final Iterator it=vil.iterator();
3:  if ((sim_level - vil.level)
        < bots_cutoff_value){
4:    while (it.hasNext())  {
5:     final Village v=(Village)it.next();
6:     async sim_village_par(v);} //while
7:  } else {
8:    while (it.hasNext()) {
9:    final Village v=(Village)it.next();
10:    sim_village_par(v);} }
11:   ... ...;} /*finish*/ ... ...; }
```

(**c**)

```
// After if expansion.
void sim_village_par(final Village vil) {
1:finish {
2: if((sim_level-vil.level)
      <bots_cutoff_value){
3:   final Iterator it=vil.iterator();
4:   while (it.hasNext()) {
5:    final Village v=(Village)it.next();
6:    async sim_village_par(v);}// while
7:        ... ...;
8:  }else {
9:    final Iterator it=vil.iterator();
10:   while (it.hasNext()) {
11:    final Village v=(Village)it.next();
12:    sim_village_par(v);}
13:    ... ...;} /*finish*/}... ...;}
```

(**d**)

```
// After finish unswitching
void sim_village_par(final Village vil) {
1: if ((sim_level - vil.level)
       < bots_cutoff_value){
2:   finish {
3:   final Iterator it=vil.iterator();
4:   while (it.hasNext()) {
5:    final Village v=(Village)it.next();
6:    async sim_village_par(v);} // while
7:    ... ...; } // finish
8: } else {
9:   finish {
10:  final Iterator it=vil.iterator();
11:  while (it.hasNext()) {
12:   final Village v=(Village)it.next();
13:   sim_village_par(v);} // while
14:   ... ...;} // finish
15: }  ... ...; }
```

(**e**)

```
// After redundant finish elimination
void sim_village_par(final Village vil) {
1:if((sim_level-vil.level)
     <bots_cutoff_value){
2: finish {
3:  final Iterator it=vil.iterator();
4:  while (it.hasNext()) {
5:   final Village v=(Village)it.next();
6:   async sim_village_par(v);} // while
7:   ... ...; } // finish
8:} else {
     // finish eliminated
9:   final Iterator it=vil.iterator();
10:  while (it.hasNext()) {
11:   final Village v=(Village)it.next();
12:   sim_village_par(v);} // while
13:   ... ...;
14: } ... ...; }
```

(**f**)

Fig. 14. Applying the iterative finish-elimination algorithm. (a) input program; (b) after if expansion; (c) after for unswitching; (d) after if expansion; (e) after finish unswitching; (f) after finish-elimination. Transformations are shown in **bold** face.

more efficient than the original code, the translation in Figure 15(b) is arguably more efficient than that in Figure 15(c). However, dependences in different parts of the code may (or may not) permit either of the translations.

We adopt a two-phase strategy for `forall`-coarsening, as shown in the overall block diagram in Figure 16: first we apply a set of transformations to attempt *simple* `forall`-*coarsening* (which needs no additional synchronization). After that, we address coarsening that may require synchronization. The different sets of transformations

```
for (int i=0;i<n;++i){
 S1;
 forall(point[j]:[1..m]){
  S2;
 }
 S3;
}
```
(a)

```
for (int i=0;i<n;++i){
 S1; }
forall(point[j]:[1..m]){
 for (int i=0;i<n;++i){
  S2; } }
for (int i=0;i<n;++i){
 S3; }
```
(b)

```
forall(point[j]:[1..m]){
 for (int i=0;i<n;++i){
  next S1;//next-single
  S2;
  next S3;//next-single
 }
}
```
(c)

Fig. 15. (a) Example program; (b) simple `forall`-coarsening: does not need any additional barriers (assuming that dependences permit); (c) `forall`-coarsening with synchronization: requires additional barriers (`next` statements), but is always legal.

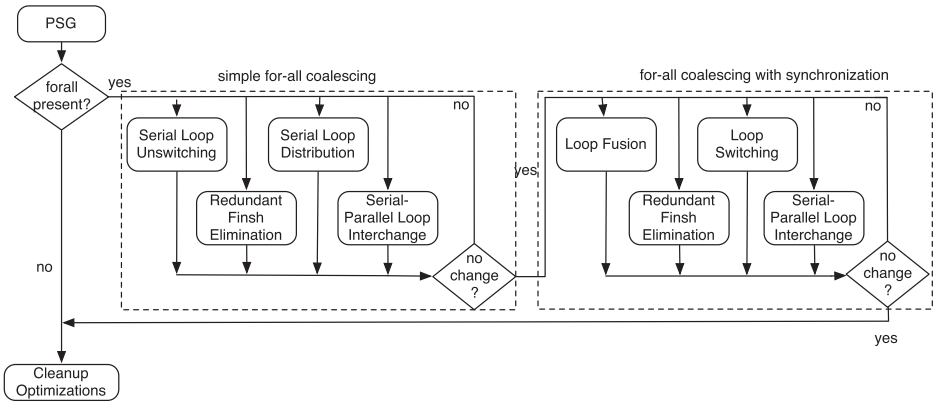

Fig. 16. Block diagram for `forall`-coarsening.

in each of these two phases satisfy a confluence; though they may be applied in any order, the resulting transformed code is guaranteed to be the same. Finally, we apply some cleanup optimizations to further optimize the generated code. We now present the details of each of these phases.

The rules for *simple* `forall`-*coarsening* and `forall`-*coarsening with synchronization* are derived from the transformation rules given in Figure 9 and Figure 10; a similar approach can also be applied to a limited set of while loops, as in Figure 3. We first start with the *simple* `forall`-*coarsening*: for any `for` loop, we repeatedly apply serial loop distribution, serial loop unswitching, redundant finish-elimination, and serial-parallel loop interchange until: (a) no `forall` statement occurs in the body of `for` loops, or (b) no further change is possible.

In contrast to *simple* `forall`-*coarsening*, intuitively, `forall`-*coarsening with synchronization* replaces fork-join synchronization by barrier synchronization, thereby further increasing the scope of `forall` iterations. For any `forall` loop, we repeatedly apply loop fusion, loop switching, redundant finish-elimination, and serial-parallel loop interchange until: (a) no `forall` statement occurs in the body of `for` loops, or (b) no further change is possible. The idea behind `forall`-*coarsening with synchronization* is to replace task creation/termination operations by lighterweight barrier synchronizations. This enables the programmer to express parallelism at a fine-grained task level and to leave it to the compiler and runtime to map the parallelism to a coarser level that can be implemented more efficiently.

Loop interchange is the key transformation to realize `forall`-*coarsening*. However, we do not stop the coarsening pass after a successful loop interchange. We keep iterating in search of further gains. The key loop-interchange rule discussed before requires

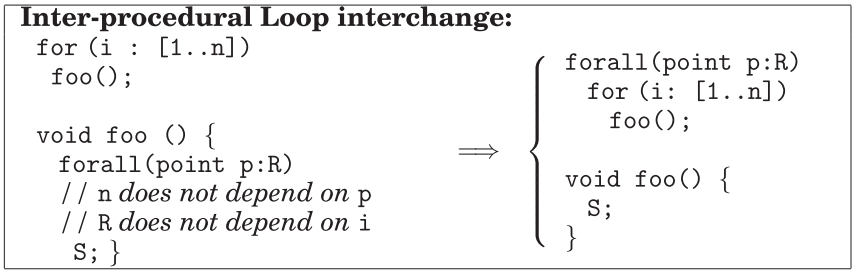

Fig. 17. Sample interprocedural translation rule.

that the body of the `for` loop should consist of only a `forall` loop. The other transformations used in both *simple* `forall`-*coarsening* and `forall`-*coarsening with synchronization* are used to fulfill that requirement.

4.2.1. Cleanup Optimizations and Discussion. The `forall`-coarsening techniques explained in the previous section may result in many `next` barriers inserted in the code. As part of our cleanup optimizations, we use an algorithm called Redundant Next/next-Single Elimination (RNSE). We use the following three heuristics:

— A `next` statement is considered redundant if the task drops the corresponding phaser without accessing any shared state (updated by another task in the same phase) after the barrier call.
— A `next`-single statement {`next S;`} can be replaced by {`next;S;`} if multiple parallel instances of the statement `S` can be executed independent of each other.
— A `next` statement is considered redundant if it always precedes another barrier, and the two sets of tasks registered on the phasers of these barriers are identical.

We invoke a post-pass of copy propagation, dead-code assignment elimination, and loop fusion (rule 7, Figure 9) that helps us further fine-tune our output.

We make a simple interprocedural extension to all the transformation rules described earlier. We present a sample interprocedural transformation for loop interchange in Figure 17. The remaining rules are similar in nature and effect.

While the two `forall`-coarsening phases explained in this section consist of multiple transformations, only two of them (*serial-parallel loop interchange* and *loop fusion*) actually contribute to any reduction in task creation and termination overhead. The rest of the transformations aid in increasing the scope and impact of these two transformations.

Traditional *loop interchange* transformation has a known history of impact on the cache behavior. For example, loop interchange transformation on the example given next can improve the cache performance of accessing `b[j][i]`, but it can degrade the reuse of `a[i]` and `c[i]`.

```
for (i: [1:10000])
   for (j : [1:10000])
       a[i] = a[i] + b[j][i] * c[i];
```

As a result, the overall performance may be degraded after loop interchange. Now say that the inner loop is a `forall` loop. Loop interchange interestingly can improve/-worsen the cache behavior of `a[i]`, `c[i]`, and `b[j,i]` (depending on the cache protocol). Studying the impact of cache on loop interchange would be an interesting problem in itself, and we leave it for future work. Increasing task granularity without any control

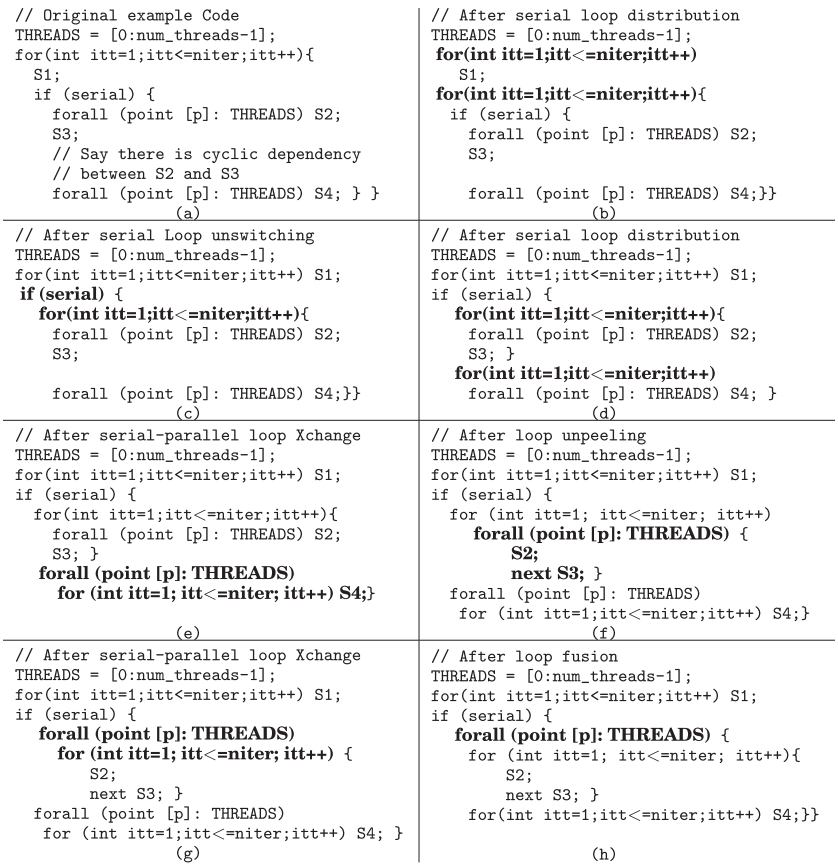

Fig. 18. Applying the `forall` coarsening described in Figure 16. (a) the input program; (b) *simple* `forall`-*coarsening*: serial loop distribution; (c) *simple* `forall`-*coarsening*: loop unswitching; (d) *simple* `forall`-*coarsening*: serial loop distribution; (e) *simple* `forall`-*coarsening*: serial-parallel loop interchange; (f) `forall`-*coarsening with synchronization*: loop unpeeling; (g) `forall`-*coarsening with synchronization*: serial-parallel loop interchange; (h) cleanup optimization: loop fusion. The changes are shown in **bold** font.

can also have a negative effect on load balancing (as the total parallelism is reduced). Identifying the optimal task size is a quite challenging problem in itself and is beyond the scope of this article. We assume that the compiler that invokes our `forall`-*coarsening* phase knows the maximum allowed task size and accordingly can control the coarsening phase to generate tasks with optimal size.

Another key point to note is that though transformations such as *loop fusion* and *loop unpeeling* can decrease task creation and termination overheads, they may increase memory overheads due to the possible increase in the number of tasks live at a certain point in time. However, the loop-chunking phase that follows the `forall`-*coarsening* phase ameliorates this issue to a large extent.

We now present the effect of invoking our framework on an input program shown in Figure 18(a). Figure 18(b)–(h) show the results of applying our transformations on the input program. As described in Figure 16, *simple* `forall`-*coarsening* is applied first. There is no cyclic dependency between `S1` and the rest of the loop body, thus enabling loop distribution (shown in Figure 18(b)). Next, the *serial loop unswitching* rule is applied, and the conditional construct is moved out of the `for` loop (shown in

```
1. delta=epsilon+1; iters=0;
2. forall (point[j] : [1:n]) {
3.   while (delta > epsilon) {
4.     newA[j]=(oldA[j-1]+oldA[j+1])/2.0;
5.     diff[j]=Math.abs(newA[j]-oldA[j]);
6.     next {
7.      delta=diff.sum(); iters++;
8.      temp=newA; newA=oldA; oldA=temp; }}}
```

Fig. 19. Semantically equivalent translation of the code shown in Figure 3.

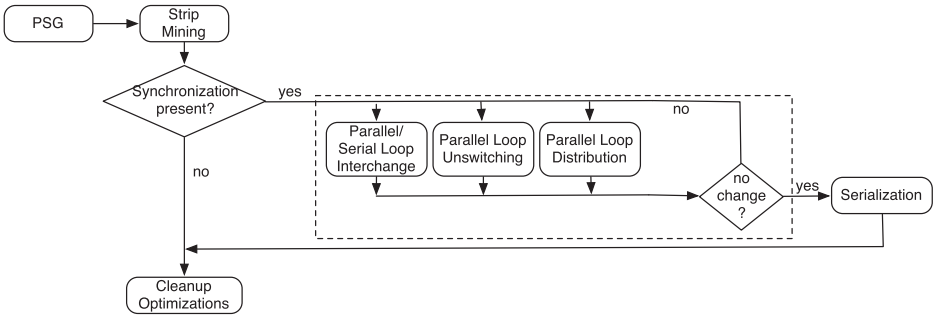

Fig. 20. Block diagram for the loop-chunking pass.

Figure 18(c)). Next, the *serial loop distribution* rule is applied (shown in Figure 18(d)). Note that, due to the cyclic dependency between S2 and S3, the loop cannot be further distributed. After the application of the *serial-parallel loop interchange* rule (shown in Figure 18(e)), there is no more scope for *simple* `forall`-*coarsening* and we proceed to apply `forall`-*coarsening with synchronization*.

First, the *loop unpeeling* rule is applied (shown in Figure 18(f)). After that, the *serial-parallel loop interchange* rule is applied again (shown in Figure 18(g)); at this point, no other `forall` loop occurs in the body of any `for` loop. To increase the granularity, the two `forall` loops can be merged by loop fusion (shown in Figure 18(h)); this is done in the context of cleanup optimizations. Comparing the original code (in Figure 18(a)) and the final code (in Figure 18(h)) clearly shows that `forall`-coarsening is not a straightforward transformation. Likewise, Figure 19 shows the correct transformation for the code snippet in Figure 3(a).

4.3. Loop-Chunking

In this section we present our chunking phase to enable chunking of `foreach` loops containing synchronization operations. The synchronization operation that we will focus on in this description is the `next` statement for clocks and phasers; as mentioned in Section 2, the phaser `next` statement can be used to support both barrier and point-to-point synchronizations.

Figure 20 shows a block diagram for our chunking phase. The general strategy to chunk parallel loops containing synchronization operations is as follows. The `foreach` loop is first strip mined into two nested parallel loops. If the loop body contains no `next` statements, then the inner loop can be serialized, and a chunked version can be obtained after performing some cleanup transformations (the "NO" case in the flowchart). If the loop body contains `next` statements, then a combination of three transformations—parallel loop distribution, parallel-serial loop interchange, and parallel loop unswitching (presented in Figure 9)—is applied repeatedly until: (a) no `next`

```
1. finish {
2.  ph = new phaser(); // SIG_WAIT mode by default
3.  foreach (point i: R) phased(ph) {
4.    for (int j = 0; j < m; j++) {
5.      S1;
6.      next;
7.      if (array[j] != 0) {
8.        for (int k = 0; k < l; k++) {
9.          S2;
10.         next; } } } } }
```

Fig. 21. Example `foreach` loop containing `next` statements.

statements occur inside any instance of an inner `foreach` loop, or (b) no further change is possible. In case (a), we can proceed to the serialization and cleanup transformations as before to obtain a chunked parallel loop. In case (b), the compiler is unable to chunk the parallel loop and the `foreach` statement is left unchanged. The motivation for selecting the preceding three transformations to iterate on is to attempt to isolate the `next` statements by moving the inner parallel loop as far inward as possible. The three transformations used in this framework are monotonic—though they may be applied in any order, the resulting transformed code is guaranteed to be deterministic. Of these three transformations, the *parallel loop distribution* is the basic transformation needed for chunking by isolating next operations. Interchange and unswitching increase the opportunities for isolation. Next contraction (described shortly) and choice of chunking policy are used to improve the efficiency of the chunked version.

Next Contraction:

```
i-forall (point p : R1)
  next                    ⟹   { next
// Region R1 is non-empty.
```

Next contraction is a new transformation that is specific to `X10` clocks and HJ phasers. If we have an `i-forall` loop that contains only a `next` statement, then we can replace it by a single `next` statement provided that its region is nonempty. This is because the only visible effect of an "`i-forall next`" statement is synchronization with other activities, which can be achieved just as well by a single `next` statement.

In this work, we assume that all programmer-specified conditions guarding a `next` statement are invariant in the initial `foreach` loop, that is, the conditions are *single-valued* [Yelick et al. 2007]. However, as we will see in Section 5.3, our transformation framework can handle cases in which a `next` statement is guarded by implicit exception conditions.

Figure 21 contains an example `foreach` loop with `next` statements. In this example, all iterations of the `foreach` loop are registered in *signal-wait* mode on phaser `ph`, which means that the `next` statements serve as barrier operations. However, the transformation framework is also applicable to other phaser registration modes for which a `next` statement may result in point-to-point synchronizations instead of a barrier operation. It is obvious that a standard chunking of the `foreach` loop in Figure 21 will not be legal. The following sections describe the transformations performed by a framework that can lead to a legal chunking.

4.3.1. Strip Mining. The classical strip mining transformation results in chunks of contiguous iterations. However, for generality, we will define strip mining of a region

Chunking Policy	Iteration Sets
Block	$\{0, 1, \cdots \frac{N}{P} - 1\}, \{\frac{N}{P}, \frac{N}{P} + 1, \cdots 2 \times \frac{N}{P} - 1\}, \cdots, \{(P-1) \times \frac{N}{P}, \cdots N - 1\}$
Cyclic	$\{0, P, \cdots,\}, \{1, P+1, \cdots,\}, \cdots, \{P-1, 2 \times P - 1, \cdots,\}$

Fig. 22. Iteration sets for block and cyclic chunking policies for region $R = [0 : N - 1]$ and P chunks.

```
foreach (point p: R) phased(⟨args⟩)          foreach (point g: Ig(R)) phased(⟨args⟩)
        S                                  ⟹      i-forall (point p: i.e.,Rg)
                                                       S
```

Fig. 23. `foreach` strip mining transformation rule.

(iteration space) R to be an ordered pair (`Ig`, `Ie`), where `Ig`(R) is an iterator over multiple chunks and for each chunk g, `Ie`(R, g) returns an iterator over the different indices in the chunk. In addition to the ability to specify chunks of noncontiguous iterations, this formulation allows us to specify chunking of multidimensional loops since regions can be multidimensional in HJ. Figure 22 shows the iteration spaces for block and cyclic chunking policies for region $R = [0 : N - 1]$ with P chunks.

Our rule for strip mining `foreach` loops is shown in Figure 23. The `i-forall` is a special "inner `forall`" construct that is defined only for our transformation framework. It is not available to the programmer, and it will not be present in the final output code. This new construct carries forward the dependence information and the exception semantics until we do the actual transformation. If chunking is successful, then all instances of `i-forall` are replaced by sequential `for` loops; otherwise the original `foreach` loop remains unchanged. This all-or-nothing approach is proposed for simplicity; extensions to support partial chunking is a topic for future work. Also, the real benefit of chunking in practice will only be realized when it is performed across all statements in the original `foreach`, since even a single unchunked statement will result in the creation of a large number of fine-grained activities.

The `i-forall` loop is very similar to the standard `forall` loop, except that it has no `phased` clause, thereby registering on all the parent's phasers with the same modes as the parent activity, that is, the outer `foreach`. Also, though transmission of clocks and phasers is not permitted through explicit `finish` operations in HJ, it is permitted through the implicit `finish` in an `i-forall` because we know that all `i-foralls` will eventually be replaced by sequential loops if a chunking transformation is performed. Considering the similarities between the `i-forall` and `forall` loop, all the transformations, listed in Figure 9 are applicable for `i-forall` as well.

The strip mining transformation (shown in Figure 23) is always legal, since the inner `i-forall` loop is still parallel. The fact that the inner `i-forall` has an implicit `finish` does not limit the parallelism in the original loop. Figure 24 shows the result of the strip mining transformation when applied to the code example in Figure 21 (the changes are shown in **bold**).

Our serialization mechanism (described in Section 4.3.2) requires that no `next` operations appear in any `i-forall` construct. In this section, we describe an iterative approach to either move all `next` operations out of the `i-forall` loops targeted for serialization or declare the original `foreach` loop to be non-chunkable. This approach is based on repeated applications of the transformations shown in Figure 20 and described in Figure 9 and Figure 10.

Figure 25(a)–(d) shows the results of applying our transformations on the strip mined code in Figure 24. First, Figure 25(a) shows the result of interchanging the `i-forall` loop with the sequential `for-j` loop. Next, Figure 25(b) shows the result of distributing the `i-forall` into three new `i-forall` loops. Then, Figure 25(c) shows the

```
finish {
  ph = new phaser();
  foreach (point i: R) phased(ph) {
    for (int j = 0; j < m; j++) {
      S1;
      next;
      if (array[j] != 0) {
        for (int k = 0; k < l; k++) {
          S2;
          next; } } } } }
```

(a)

```
finish {
  ph = new phaser();
  foreach (point g: Ig(R)) phased(ph) {
    i-forall (point i : Ie(R, g)) {
      for (int j = 0; j < m; j++) {
        S1;
        next;
        if (array[j] != 0) {
          for (int k = 0; k < l; k++) {
            S2;
            next; } } } } } }
```

(b)

Fig. 24. Strip mining of `foreach` loop: (a) original code; (b) transformed code.

```
// After parallel-serial loop Xchange
finish {
  ph = new phaser();
  foreach (point g: Ig(R)) phased(ph){
    for (int j = 0; j < m; j++) {
      i-forall (point i : Ie(R, g)) {
        S1;
        next;
        if (array[j] != 0) {
          for (int k = 0; k < l; k++){
            S2;
            next; } } } } }
```

(a)

```
// After parallel loop distribution
finish {
  ph = new phaser();
  foreach (point g: Ig(R)) phased(ph) {
    for (int j = 0; j < m; j++) {
      i-forall (point i : Ie(R, g)) {
        S1; }
      i-forall (point i : Ie(R, g)) {
        next; }
      i-forall (point i : Ie(R, g)) {
        if (array[j] != 0) {
          for (int k = 0; k < l; k++) {
            S2;
            next; } } } } }
```

(b)

```
// After next contraction and
// parallel loop unswitching
finish {
  ph = new phaser();
  foreach (point g: Ig(R)) phased(ph){
    for (int j = 0; j < m; j++){
      i-forall (point i : Ie(R, g)){
        S1; }
      next; // Contracted
      if (array[j] != 0) {
        i-forall (point i : Ie(R, g)) {
          for (int k = 0; k < l; k++){
            S2;
            next; } } } } }
```

(c)

```
// After parallel-serial loop Xchange,
// serial loop distribution,
// and next contraction
finish {
  ph = new phaser();
  foreach (point g: Ig(R)) phased(ph) {
    for (int j = 0; j < m; j++){
      i-forall (point i : Ie(R, g)){
        S1; }
      next;
      if (array[j] != 0) {
        for (int k = 0; k ¡ l; k++) {
          i-forall (point i : Ie(R, g)){
            S2; }
          next; // contracted
        } } } } }
```

(d)

Fig. 25. Applying our iterative transformation framework on the strip mined code in Figure 24. The changes in each step are shown in **bold**.

result of applying the rules *next contraction* and loop unswitching to move the third `i-forall` further inwards. Finally, Figure 25(d) shows the result of applying *loop interchange*, *loop distribution*, and *next contraction* transformations; it achieves our desired goal of isolating all `next` statements.

4.3.2. Serialization. The job of serialization is to confirm that no `i-forall` statement contains a `next` and (if so) to serialize all the `i-forall` constructs. If such an `i-forall` loop contains only a `for` loop nest and they are perfectly nested, we have the flexibility to apply additional parallel-serial loop interchanges. As a preprocessing of serialization, we readjust the position of th `ei-forall` loop so as to improve spatial data

```
1. finish {
2.  ph = new phaser(); // SIG_WAIT mode by default
3.  foreach (point g: Ig(R)) phased(ph) {
4.    for (int j = 0; j < m; j++) {
5.      for (point i : Ie(R, g)) {
6.        S1; }
7.      next;
8.      if (array[j] != 0) {
9.        for (int k = 0; k < l; k++) {
10.         for (point i : Ie(R, g)) {
11.           S2; }
12.         next;  } } } } }
```

Fig. 26. The chunked code for the running example shown in Figure 21.

locality. This loop readjustment pass brings performance improvements especially when the loop index of he `i-forall` loop is used in the innermost dimension of arrays, for example, `i-forall(i:[...]) { for(j:[...]) { A[j][i] ...}`. Figure 26 shows the generated code after the serialization pass is performed on the transformed code in Figure 25(d). The last `next` operation in Figure 26 is necessary because it is performed dynamically in each iteration of its immediately enclosing `for`–`k` loop. For example, if `S2` is chosen to be "`A[k][i+C] = A[k+1][i] + 1;`" and offset `C` is chosen to be larger than the chunk size, there can be a data race among the `foreach` iterations if the `next` statement is removed. A quick comparison with the original code in Figure 21 confirms that loop-chunking of parallel loops is not a straightforward transformation.

5. EXTENSIONS FOR EXCEPTIONS

In this section, we discuss the impact of exception semantics on the three optimization techniques discussed in Section 3 by extending the rules presented in Figure 9 and Figure 10. The rules in this section are presented in the context of the HJ and `X10` v1.5 exception model (which in turn builds on the Java exception model), but the overall approach should be relevant to other languages with exception semantics (such as C++).

As discussed in Section 2, an uncaught exception thrown inside an `async` statement terminates the `async` but not its parent activity. It is caught by the surrounding (explicit or implicit) `finish`. This `finish` bundles all the caught exceptions into a `MultiException` data structure and throws this collection instead of a single exception – which unless handled will in turn terminate the activity invoking the `finish`. Exceptions thrown in the iterations of a `foreach` loop are handled similarly (they do not impact the execution of other iterations), as each iteration of the `foreach` statement can be viewed as an independent `async` statement. Thus, an uncaught exception thrown inside the iterations of a `forall` are only caught by the surrounding implicit `finish`, after all the activities forked in the `forall` have terminated.

Thus, for the rules described in Figure 9 and Figure 10 that involve modifying the scope of any possible exception throwing statement, the semantics have to be maintained explicitly. Considering the complexity of these rules, we present separate discussions to explain the impact of exceptions on each of the three optimizations presented in this article.

5.1. Finish-Elimination in the Presence of Exceptions

In this section, we discuss the impact of exception semantics on the `finish`-elimination techniques discussed in Section 4.1.1 by extending the used transformation rules. We

Finish distribution:

```
finish { S1; S2; }
// (1) S1 has no e-asyncs.
// (a) S2 has e-asyncs.
```
$\Longrightarrow$
```
try {S1;}
catch(Exception e){
  MultiException me tt=new ···;
  me.pushEx(e1); throw me; }
finish { S2; }
```

Finish unswitching:

```
finish
  if(cond) S1; else S2;
// (1) cond has no e-async
// (2) cond is exception free.
// (a) S1 or S2 has e-asyncs
```
$\Longrightarrow$
```
if (cond) finish S1;
else finish S2;
```

Loop/Finish interchange:

```
for (S1;cond;S2)
  finish S3;
// Say E_s = set of e-asyncs in S3
// (1) ¬∃e ∈ E_s: cond has dependence on e
// (2) ¬∀e ∈ E_s:body of e has loop
   carried dependence on S2 or S3
// (3) cond is exception free.
// (4) S2 is exception free.
// (a) E_s ≠ null
```
$\Longrightarrow$
```
S1;
finish {
  for(;cond;S2){
    S3;
  }
}
```

Redundant finish elimination:

```
finish S;
// (1) S has no e-async.
```
$\Longrightarrow$
```
try {S;}
catch(Exception e){
   MultiException me=new ···;
    me.pushEx(e1); throw me; }
```

Tail finish elimination:

```
finish {
  S1;
  finish S2; }
// (a) S1 and S2 have e-asyncs
```
$\Longrightarrow$
```
finish {
 S1;
 try {S2;}
 catch(Exception e){
  MultiException me=new ···;
   me.pushEx(e1); throw me;}}
```

Finish fusion

```
finish S1;
finish S2;
//Say E_s = set of e-asyncs in S1
//(1) ¬∃e ∈ E_s: S2 has dependence on e
//(2) S1 throws no exceptions
//(a) S1 and S2 have e-async.
```
$\Longrightarrow$
```
finish{
   S1;
   S2;
}
```

Fig. 27. Transformation rules for `finish`-elimination in the presence of exceptions.

follow the same overall approach as shown in Figure 12 even in the presence of exceptions. Figure 27 presents the rules that need to be modified to handle exceptions, which are briefly discussed shortly. Similar to rules presented in earlier sections, each rule has preconditions presented as comments under each rule. The preconditions on each rule fall into two categories: (i) required for semantically correct translations (indexed by numerals), and (ii) profitability constraints that are employed for efficient compilation (indexed by letters). As can be seen, the rules have now become more complicated than the ones in Figure 9 and Figure 10, thereby underscoring the importance of compiler transformation.

When the scope of a `finish` statement is reduced by taking a statement outside the scope of the `finish` node, any exception that is thrown in the body of that statement has to be handled in accordance with the exception semantics. As shown in the rule for *finish distribution*, we catch any exception caught in the statement S_1, bundle

it in a MultiException, and throw it again. Similar translation can be seen in the rules given for *redundant finish elimination* and *tail finish elimination*. The rules for *finish unswitching*, *loop/finish interchange*, and *interprocedural finish unswitching* are applied only when the predicate cond does not throw any exception.

5.2. Forall-Coarsening in the Presence of Exceptions

In this section, we discuss the impact of exception semantics on the forall-coarsening phase discussed in Section 4.2. We follow the same overall approach as shown in Figure 16 even in the presence of exceptions. Figure 28 presents the rules to handle exceptions, and are briefly discussed shortly. Besides presenting a new rule (loop switching (try-catch)), we modify the existing rules for some of the transformations.

The *serial loop distribution* rule is applied only if S2 does not throw any exceptions. It first evaluates S1, and any exception thrown in a certain iteration (maxItr) is remembered and is thrown after maxItr−1 number of iterations of S2 have been executed.

The *serial-parallel loop interchange* rule generates code to check for any thrown exceptions after each evaluation of the statement S. In the generated code, each outer parallel iteration waits for other parallel iterations to finish executing one sequential iteration of S, then each parallel iteration checks if an exception was thrown in any of the iterations (by checking the flag excp) and breaks out of the inner for loop if the flag is set. If an exception is thrown by an iteration, then it is communicated to all the other threads, which in turn terminate their execution.

The *loop unpeeling* and *loop fusion* rules generate code to evaluate the statement S2 under the condition that no instance of S1 has thrown an exception. The *loop unpeeling* rule ensures that only one instance of S2 is executed. This execution happens in a try-catch block. We save any thrown exception in the variable ex, which is checked outside the forall loop; if ex is set, then it is thrown upward. The *loop fusion* rule does not evaluate S2 inside a *try-catch* block. Since in the original code S2 is inside the forall, the semantics are preserved.

The *loop unswitching (try-catch)* is a new rule that is relevant only in the presence of exceptions. It generates code to execute each iteration of S1 inside a try-catch block and saves the thrown exception in a MultiException data structure. The pushException method avoids data races by using appropriate synchronization mechanisms. After the forall loop has terminated, we check if any exception was thrown and invoke S2 accordingly.

5.3. Loop-Chunking in the Presence of Exceptions

In this section, we discuss rules to perform loop-chunking transformations in the presence of exceptions. We first discuss the exception semantics of the i-forall statement. Since the i-forall loop is generated from a foreach statement, we must execute each iteration of the i-forall regardless of exceptions thrown in other iterations. Thus, we define the exception semantics of the i-forall as follows: all the exceptions thrown by different iterations of the i-forall are thrown as independent asynchronous exceptions, that is, they are inserted into the MultiException collection gathered at the explicit IEF (Immediately Enclosing Finish) instance for the i-forall (ignoring implicit finish operations in i-forall statements).

We follow the same overall approach as shown in Figure 20, even in the presence of exceptions. However, we modify the rules for some of the transformations to handle exceptions and present the new rules in Figure 29; these are briefly discussed next.

Strip mining: We reuse the strip mining rule presented in Figure 23; the exception semantics of the i-forall statement guarantees correct translation, keeping in mind that the implicit finish in an i-forall does not collect exceptions like an explicit finish.

Serial loop distribution:

```
for (i: [1..n])
// No dependence cycle between
// S1 and S2.
// S2 does not throw exceptions
  { S1; S2; }
```

⟹

```
int maxItr = n+1;
Exception ex = null;
for (i: [1..n])
  try {S1;}
  catch (Exception e){
    ex=e; maxItr=i; break;}
for (i: [1..maxItr-1])   S2;
if (ex ≠ null) throw ex;
```

Serial-parallel loop interchange:

```
for (i: [1..n])
// Different iterations of the for loop
// are independent.
  forall (point p : R)
    // R does not depend on i
    S;
```

⟹

```
boolean excp = false;
forall (point p : R)
  for (i: [1..n]) {
    try {S;}
    catch (Exception e)
      {excp = true; throw e;}
    next;
    if (excp==true) break; }
```

Loop Unpeeling:

```
forall (point p: R)
  S1;
S2;
```

⟹

```
boolean excp = false;
Exception ex = null;
forall (point p: R) {
  try {S1;}
    catch (Exception e)
      {excp = true; throw e;}
  next;
  if (excp == false){
    next {try {S2;}
     catch(Exception e){ex=e;}}}}
if (ex ≠ null) throw ex;
```

Loop Fusion:

```
forall (point p: R1)
  S1;
forall (point p: R2)
  S2;
// Say E_s = set of e-asyncs in S1
// ¬∃e ∈ E_s: S2 has dependence on e
```

⟹

```
boolean excp = false;
forall (point p: R) {
   try {if (R1.contains(p)) S1;}
   catch (Exception e)
     {excp = true; throw e;}
   next;
   if (excp == false)
     if (R2.contains(p)) S2;}
```

Loop Switching (try-catch):

```
try {
  forall (point p: R)
   S1
}catch(MultiException e) {S2 }
```

⟹

```
MultiException e = new ...;
boolean excp = false;
forall (point p: R)  {
  try  { S1 }
  catch(Exception e1) {
   excp = true;
   e.pushException(e1);
  } }
if (excp) S2;
```

Fig. 28. Transformation rules for `forall`-coarsening in the presence of exceptions.

Loop interchange: Loop interchange (rule 1) requires special handling in the presence of exceptions since an exception thrown in the original inner `for` loop terminates the rest of the iterations of the `for` loop, but does not impact other iterations of the `i-forall` loop. Thus, in the transformed program, for any iteration of the outer sequential `for` loop, the inner `i-forall` should be invoked at program point Q only if no

Parallel-serial loop interchange:

```
i-forall (p: Ie(R, g)) phased
  for (s1;e;s2)
    S
// s1, e, s2 don't depend on p
```

⟹

```
boolean c; Exception EX = null;
try {s1; c = e;}
  catch (Exception ex) {EX = ex; c = false;}
if (EX≠null) foreach (p: Ie(R, g)) throw EX;
Region newR = new Region(Ie(R, g));
Exception[]exArr = new Exception[newR.size()];
for (;c;) {
 for (q: newR)
    if (exArr[q]≠null) newR.remove(q);
  i-forall (p: newR) phased{
     try {// body may need renaming
       S; s2; c = false; c = e;
      }catch (Exception e){exArr[p] = e;}
  } }
foreach (p: Ie(R, g))
  if (exArr[p] ≠null) throw exArr[p];
```

Parallel loop unswitching:

```
i-forall (p: Ie(R, g)) phased
  if (e)
    S
// e doesn't depend on p and
// is side effect free
```

⟹

```
boolean c; Exception EX = null;
try {c = e;}
  catch(Exception ex){EX = ex; c = false;}
if (EX≠null) foreach(p: Ie(R, g)) throw EX;
if (c) i-forall (p: Ie(R, g)) phased
     S
```

Loop unswitching (try-catch):

```
i-forall (p: Ie(R, g)) phased
  try {
    S1
  } catch (E e) S2
```

⟹

```
try {
  finish i-forall (p: Ie(R, g)) phased
     S1
} catch (MultiException e) {
    Region newR = new Region();
    for (p: Ie(R, g)) {
      ex = e.exceptions[p];
      if (ex ≠ null && ex instanceof E)
       newR.add(p); }
    i-forall (p: newR) phased {
       Exception e = e.exceptions[p];
       S2 }
    foreach (Exception ex: e.exceptions())
      if (ex ≠ null && !(ex instanceof E))
        {throw ex;}
}
```

Parallel loop distribution:

```
i-forall (p: Ie(R, g)) phased
{
  S1;
  S2
}
```

⟹

```
Exception exArr[] = new Exception [R.size()];
boolean exFlag[] = new boolean [R.size()];
i-forall (p: Ie(R, g)) phased
  try {S1} catch (Exception e)
    {exFlag[p] = true; throw e;}
Region newR = new Region();
for (p: Ie(R, g)) if (!exFlag[p]) newR.add(p);
i-forall (p: newR) phased S2;
```

Fig. 29. Transformation rules for loop-chunking in the presence of exceptions.

exception was thrown by any of the previous sequential iterations while executing the activity at point Q. We capture this behavior by maintaining a region of points (`newR`) for which no exception has been thrown. For any exception thrown, it is stored in an array and after the whole loop is executed, the contents of the array are individually thrown in an asynchronous manner.

Loop unswitching: If the predicate of the `if` statement is loop invariant and is side-effect free, then we can compute the predicate outside the loop as shown in rule 2.

Table I. Classfication of Transformations

Class	Optimizations	Dependencee analysis
High-level (group A)	`finish`-elimination Simple `forall`-coarsening	yes (execution order can be changed)
Low-level (Group B)	`forall`-coarsening with synchronization parallel loop-chunking	no (execution order is preserved)

Loop unswitching (try-catch): A try-block within a `foreach` statement can be lifted out of the loop, by treating the try-block and the catch-block as two computations in sequence (the catch-block is executed conditionally). We have to catch all the exceptions that might be thrown in the try-block. We do so by first unswitching and then enclosing the inner `i-forall` with a `finish` statement. Any exception thrown in `S1` is caught by the `finish` and is thrown as a MultiException. In the catch statement, we analyze the multiexception and execute `S2` inside a `i-forall` loop over all the points for which we have caught an exception while executing `S1` (`newR`). All the exceptions that are not caught by the catch-clause (exception not of type E) are thrown to the next level.

Parallel loop distribution: Given the body of a `foreach` loop to be {`S1; S2`}, after the loop distribution, `S2` is executed only by those iterations where `S1` did not throw any exception. We create a new region `newR` to represent the collection of points that executes `S1` normally (did not throw an exception outside) and use it to iterate over `S2`.

Serialization of `i-forall` statements must respect their exception semantics. We present next the rule for serialization in the presence of exceptions.

```
i-forall (p: Ie(R, g)) phased          ⟹  for(p:Ie(R, g))
  S                                        try {S}
                                           catch (Exception e)
                                            {async throw e;}
```

In each iteration, we catch any exception that is thrown and throw it asynchronously. This guarantees that we throw all the caught exceptions with the same semantics as the original `foreach` loop.

6. INTEGRATION OF INDIVIDUAL OPTIMIZATIONS

In this section, we describe how the optimizations introduced in Section 4 and Section 5 can be integrated and organized in a compiler framework. We classify the optimizations proposed in this article into two groups: (A) one that requires data-dependence analysis (as the listed optimizations may alter sequential execution order), and (B) one that does not require data-dependence analysis (although the listed optimization may replace pairs of task creation and termination by barrier operations, the original execution order is preserved). Another interesting facet of the first group of transformations is that they increase the sizes of parallel activities. A natural way of organizing all transformations is to first apply the optimizations of Group-A so as to increase the ideal parallelism in a given program and then apply optimizations listed in Group-B to help derive required useful parallelism, which may take into consideration different target machine-specific information. Such a division fits well into a compiler framework, where the machine-dependent (low-level) optimizations follow the machine-independent (high-level) ones.

6.1. High-Level Optimizer

The high-level optimizer includes `finish`-elimination and *simple* `forall`-*coarsening* optimizations. In Figure 30(a), we show the transformation resulting from the

```
// After high-level optimization
1:  for (i:[1..n]) {
2:   forall (j:[1..1024]) {
3:    A[i][j] = (A[i-1][j-1] + A[i-1][j]
4:               + A[i-1][j+1]) / 3;
5: }}
6: if (!aggregate) {
7:   finish {
8:    foreach (j: [1..1024]) {
9:      for (i:[1..n]) {
10:     double tmp=processSingle(A[i][j]);
11:     atomic sum+=tmp;
12: }}}
13:}else {
14: for (i:[1..n])
       sum += processAgg (A[i]);
15: }
```

(a)

```
// After low-level optimization
1:  Region R = [1..1024];
2:  forall (point g: Ig(R)) {
3:   for (i:[1..n]) {
4:     for (j: Ie(R,g)) {
5:      A[i][j] = (A[i-1][j-1] + A[i-1][j]
6:                 + A[i-1][j+1]) / 3;
7:    }
8:     next;
9: }}
10:if (!aggregate) {
11: finish {
12:   foreach (point g: Ig(R)) {
13:    for (i: [1..n]) {
14:      for (j: Ie(R,g)) {
15:      double tmp=processSingle(A[i][j]);
16:      atomic sum+= tmp;
17: }}}}
18:}else {
19: for (i:[1..n])
        sum += processAgg (A[i]);
20: }
```

(b)

Fig. 30. (a) Effect of invoking high-level optimizer; (b) effect of invoking low-level optimizer for the input code of Figure 5(a).

application of the high-level optimizations on the HJ example discussed in Figure 5. After the high-level optimization, the task termination overhead due to `finish` (line 5 of Figure 5) is reduced by the factor of `n`, which is the loop iteration count of `for-i` loop, and the granularity of `foreach` loop (line 7 of Figure 30) increases by a factor of `n`.

6.2. Low-Level Optimizer

The low-level optimization phase applies `forall`-*coarsening with synchronization* followed by loop-chunking. Note that the loop-chunking framework discussed in Section 4.3 can handle arbitrary parallel loops with barrier synchronizations created by the coarsening pass.

In Figure 30(b), we show the transformation resulting from the application of the low-level optimizations on the HJ code shown in Figure 30(a). The `forall`-*coarsening with synchronization* is applied to the first loop nest (lines 1–5 of Figure 30(a)), and the barrier in the inner `forall` loop is compensated with a lightweight `next` operation. Furthermore, parallel loop-chunking is applied to both `forall` loops to reduce excessive task creation. The details of the actual distribution used for the chunking are abstracted out by means of two symbolic iterators `Ig` and `Ie` that iterate on the groups and the elements of individual groups, respectively.

7. IMPLEMENTATION

The transformation framework discussed in this article was implemented in the context of the Habanero-Java Compiler framework (HJC) [Habanero 2009], which translates Habanero-Java (HJ) (see Section 2) source code to Java bytecode, along with calls to some relevant runtime APIs (RT APIs). Figure 31 presents the overall structure of the HJC compiler. The Polyglot [Nystrom et al. 2003]-based front-end for HJ was modified to emit a new Parallel Intermediate Representation (PIR) extension [Zhao and Sarkar 2011] to the Jimple intermediate representation used in the SOOT bytecode analysis and transformation framework [Vallée-Rai et al. 1999]. In addition to standard Java operators, the PIR includes explicit constructs for parallel operations, such as `async`, `finish`, and `isolated`.

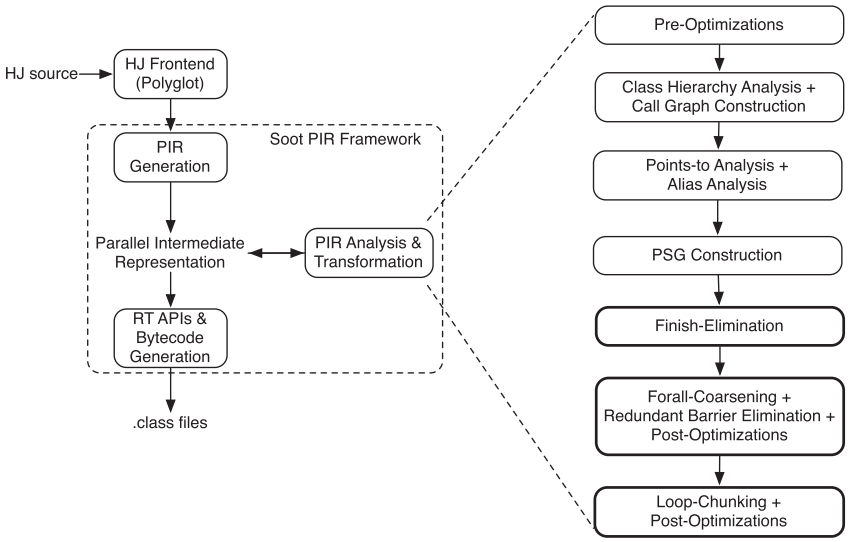

Fig. 31. Habanero-Java Compiler framework.

The analysis and transformations described in Section 4 and Section 5 are implemented in the HJC as additional optimization passes over the PIR. All the analyses and the transformations presented in this article are interprocedural in nature.

Some of the applied analysis and optimizations are shown on the right side of Figure 31. To help in the following phases of optimizations, we employed some preoptimization passes, such as constant propagation, loop-invariant code motion, copy propagation [Muchnick 1997], and method inlining within our compilation framework as the initial stage. After the preoptimization passes, we invoked several analysis passes to assist the following passes of alias analysis and dependence analysis, including class-hierarchy analysis [Dean et al. 1995], call graph construction [Muchnick 1997], and points-to analysis [Lhoták and Hendren 2003]. Our proposed data-dependence analysis uses some of the following analysis: region-level (e.g., `finish`, HJ method) escape analysis, interprocedural side-effect, and purity analysis, which is similar to the analysis presented in Salcianu and Rinard [2005]. For dependence analysis in loops, we used the GCD test [Muchnick 1997] adapted to Java with value numbering of array references [Sarkar and Fink 2001].

We start with building a Program Structure Graph (PSG), then proceed with our proposed three optimizations that may use the discussed dependence analysis: `finish`-elimination, `forall`-coarsening, and loop-chunking. After the coarsening phase, we apply redundant barrier elimination to remove the redundant lightweight barriers [Nicolau et al. 2009] and several postoptimization passes to clean up the code, including copy propagation and dead assignment elimination. Finally, the loop-chunking phase chunks fine-grained parallel loops into coarse-grained parallel tasks.

8. EMPIRICAL EVALUATION

In this section, we present experimental results for evaluating the transformation framework described in this article using the HJ compiler and runtime system [Habanero 2009; Zhao et al. 2010]. All transformations were performed using the rules in Section 5, which assume the possibility of exceptions. We discuss the details of the experimental setup in Section 8.1 and present the overall improvement of the optimization framework compared with unoptimized parallelism in Section 8.2.

Table II. Benchmarks

Bench. Suite	Prog. Name	code size	PSG nodes	Transformations			% incr in comp time
				finish elimination	forall coarsening	loop chunking	
Cilk	lud	1121	531	×	×		15.3
BOTS	fft	4480	290	×		×	17.4
	floorplan	327	110	×		×	9.35
	health	470	188	×		×	7.85
	strassen	655	117	×			8.22
NPB	cg	1160	821		×	×	15.5
	mg	1810	847		×	×	20.2
JGF	sor	175	72		×	×	18.7
	lufact	467	118		×	×	15.2
	moldyn	741	168		×	×	12.3

To understand the impact of each of the optimizations, we present a discussion on the incremental gains resulting from each of the three optimizations in the reverse order in which they are applied: effect of only loop-chunking (in Section 8.3), effect of `forall`-coarsening on top of loop-chunking (in Section 8.4), and the effect of `finish`-elimination on top of `forall`-coarsening and loop-chunking (in Section 8.5).

8.1. Experimental Setup

We used three multicore platforms for our experimental evaluation: (a) a 128-thread (dual-socket, 8 cores per socket, 8 threads per core) 1.2 GHz UltraSPARC T2 (Niagara 2) with 32GB main memory, running Solaris 10 and Sun JDK 1.5 (32-bit version); (b) a 16-core (quad-socket, quad-core per socket) Intel Xeon 2.4 GHz system with 30GB of memory, running Red Hat Linux (RHEL 5) and Sun JDK 1.6 (64-bit version); and (c) a 32-core (quad-socket, 8 cores per socket) 3.55 GHz Power7 with 256GB main memory, running Red Hat Linux (RHEL 5.4) with SMT=1 and IBM JDK 1.6 (64-bit version). This variation in platforms enables us to study the impact of different hardware on the performance improvements. For all the runs, the main program was extended with a 30-iteration loop within the same Java process, and the best of the 30 times was reported in each case so as to reduce the impact of JIT compilation overhead in the performance results, in accordance with the methodology reported by Georges et al. [2007]. The HJ runtime option, "-places 1:W", was used to set up an HJ execution for all runs with 1 place and W worker threads per place.

To evaluate our transformation framework, we use the following benchmarks ported to HJ by using the parallel constructs of HJ such as `finish`, `async`, `foreach`, `forall`, `isolated`, and phasers: four BOTS benchmarks[7] [Duran et al. 2009] (`health`, `floorplan`, `strassen` and `fft`); two NAS parallel benchmarks [Bailey et al. 1991] (`cg`, `mg`); one Cilk benchmark [Feng and Leiserson 1997] (`lud`); and three Java Grande Benchmarks [JGF 2000] (`lufact`, `sor`, and `moldyn`). We chose those benchmarks that can get benefit from more than one of the three transformations discussed in this article. Table II gives some details of the benchmarks, including the source-code size, number of PSG nodes, and the transformations applied. The last column depicts the incurred overhead in terms of the percent increase in the compilation time due to our optimization passes.

[7]The HJ versions of the BOTS benchmarks were obtained by porting the OpenMP versions to HJ. The OpenMP 3.0 `task`, `taskwait`, and `critical` directives were replaced by `async`, `finish`, and `isolated` statements in HJ, respectively.

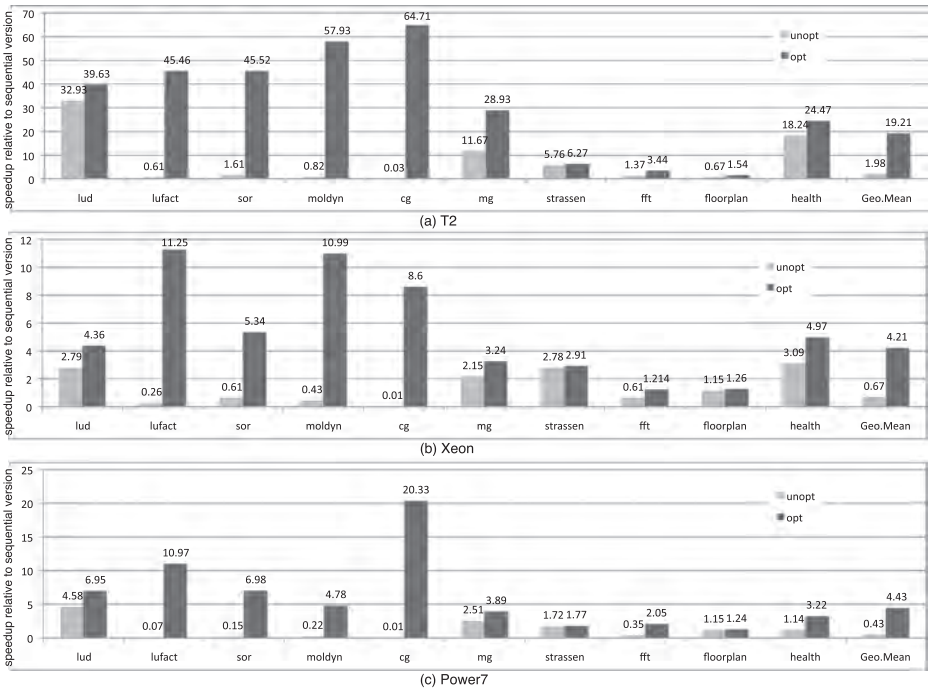

Fig. 32. Performance improvement by overall transformations. "unopt": Compilation with the base HJ compiler, with none of the optimizations discussed in this article; "opt": Compilation with the base HJ compiler, extended with all the three optimizations passes discussed in the work.

8.2. Overall Improvement

Figure 32 shows the comparison of the speedups between the unoptimized parallel benchmarks and the optimized versions that were generated by the transformation framework discussed in this article. This shows the overall improvement by applying the three stages of optimizations. The last column shows the geometric mean average improvement of the optimized and the unoptimized versions, each compared to the Java serial version. In the charts, we show the comparison with respect to the Java serial version to show the reader that the ported benchmarks do not perform worse than the serial benchmarks (an indication to the goodness of the ported benchmarks). Compared to the unoptimized version, the minimum, maximum, and geometric mean average improvements of our optimized version are as follows: on T2 Niagara 1.09×, 2049.04×, and 9.68×, respectively; on Xeon 1.05×, 1103.90×, and 6.28×, respectively; on Power7 1.03×, 3935.88×, and 10.3×, respectively.

We now present the impact of the individual optimizations introduced in this article. We follow a practice similar to that of optimizing compiler evaluations where back-end optimizations are used to establish a baseline for evaluating the impact of higher-level optimizations. In our evaluation, the results for the lower-level optimizations are presented first, and higher-level optimizations are then added incrementally to study their impact.

8.3. Impact Due to Foreach Loop-Chunking

Chunking the fine-grain parallel loops into coarse-grained parallel tasks eliminates the significant overhead for task spawning and scheduling. This section presents the effect of loop-chunking on nine benchmarks; `lud` and `strassen` were not included in

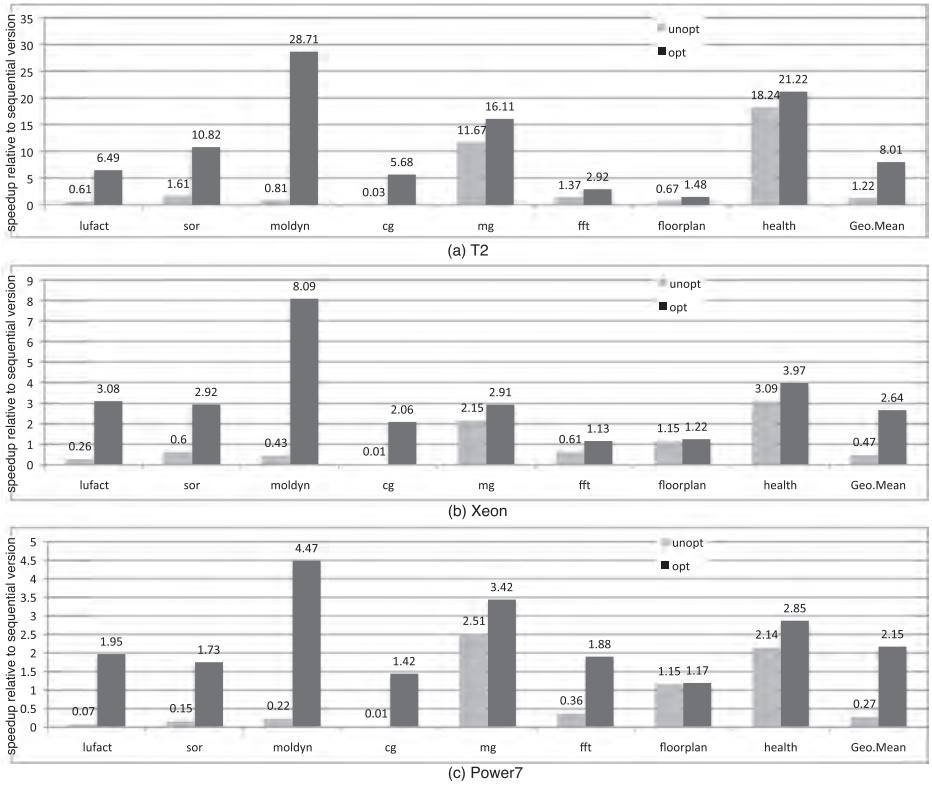

Fig. 33. Performance improvement by loop-chunking. "unopt": Compilation with the base HJ compiler, with none of the optimizations discussed in this article; "opt": Compilation with the base HJ compiler, extended with loop-chunking.

this discussion since there are no parallel loops in these two benchmarks to show any gains from chunking. Figure 33 shows the speedups on the three SMP platforms. The bar charts show the comparison of the speedups (HJ parallel program versus Java sequential program and chunked HJ parallel program versus Java sequential program). Compared to the unoptimized version, the geometric mean average improvements of the version optimized using loop-chunking are 6.56×, 6.28×, and 9.77× on T2, Xeon, and Power7, respectively.

8.4. Impact Due to Forall-Coarsening

The benefits of `forall`-coarsening can be categorized into two heads: (a) direct improvements: reduced task creation, termination, synchronization, and scheduling overheads; and (b) indirect improvements: transformations like loop interchange and loop fusion may improve locality. Regarding the scope of impact, not all of the benchmarks can benefit from this optimization; only those that contain SPMDizable `forall` loops can be transformed by coarsening. Figure 34 gives the performance comparison between the programs optimized with `forall`-coarsening + loop-chunking (tagged as "opt") and programs optimized with only loop-chunking (tagged as "unopt"). An shown in these charts, `forall`-coarsening leads to significant improvements. The amount of gains in the coarsened version (compared to the non-coarsened version) depends on the granularity of the parallel tasks in the input programs. Benchmarks with finer-grain tasks (i.e., CG, SOR and LUFact) report higher gains. Compared to the "unopt" version

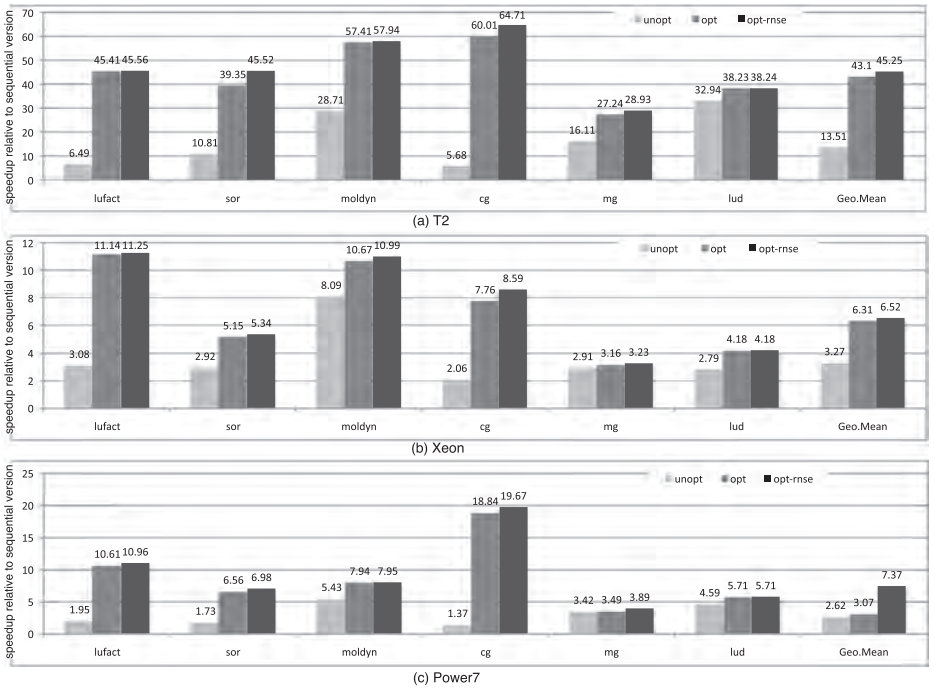

Fig. 34. Performance improvement by `forall`-coarsening. "unopt": Compilation with the base HJ compiler, extended with loop-chunking; "opt": Compilation with the base HJ compiler, extended with loop-chunking and `forall`-coarsening; "opt-rnse": opt + redundant next/next-single elimination.

the geometric mean average improvements of the "opt" version are 3.19×, 1.93×, and 1.17× on T2, Xeon, and Power7, respectively.

In these charts we have an additional evaluation point, namely that of the "opt" version further optimized with the Redundant Next/Next-Single Elimination (RNSE) phase (see Section 4.2). Compared to the "unopt" version the geometric mean average improvements of the "opt+RNSE" version are 3.35×, 1.99×, and 2.81× on T2, Xeon, and Power7, respectively.

One interesting aspect of this study was that the behavior of these benchmarks varied between Xeon, Niagara, and Power7 systems. For instance, RNSE is effective on MG and SOR on Niagara, on CG on Xeon, and MG and SOR on Power7. We attribute this to the significantly varying system architecture (Niagara and Power7 are multithreaded, Xeon is not; in Niagara all cores on a chip share the same L2 cache, Xeon contains two L2 caches each shared by two cores, and in POWER7 each core has 32KB L1 and 256KB L2 cache, and 32MB L3 cache is shared by 8 cores on a chip).

8.4.1. Coarsening and Data Locality. The improvements shown in Figure 34 result from two factors, and the data locality plays an important role for performance improvement, especially for those loop parallelism benchmarks. We now present a study to understand the contribution of these factors in the improvements cited in Section 8.4. To understand the impact of these two underlying factors, we conducted a simple experiment: for each of the benchmarks presented in Figure 34, we compared the following three versions:

— "unopt": parallel version of the benchmark with no coarsening;
— "opt": manually apply the `forall`-coarsening;

Table III. Execution Time (in seconds) Numbers to Identify the Impact of Locality

Benchmark	8 hardware threads			64 hardware threads		
	unopt	locality	opt	unopt	locality	opt
cg	16.40	10.87	9.37	11.67	12.07	1.40
mg	19.03	12.28	12.07	4.11	4.00	2.81
sor	11.37	6.89	6.56	2.72	2.79	1.01
lufact	32.34	19.53	18.39	13.28	14.28	3.19
moldyn	65.51	33.19	32.69	10.45	7.97	5.58

— "locality": we counted the reduction in the number of activities and barriers in the "opt" version, and manually inserted code to create an equal number of dummy activities and the corresponding barriers to achieve comparable task overheads to the "unopt" version while preserving the locality of the "opt" version.

The locality version gives a rough estimate of the impact due to improvements in data locality only (by comparing it to the "opt" version). For instance, the locality version for the code shown in Figure 15(b) is generated by adding the following compensation code.

```
for(i=0;i<n-1;++i){forall(j:[1..m]){/* empty */}}
```

Table III presents the execution time numbers for each of the three versions of the benchmarks shown in Table II. We only present the numbers on Niagara T2 system, by setting the number of parallel threads to 8 (when all the 8 threads are scheduled on one socket and share L2 cache), and 64 (the 64 threads could be scheduled on both two sockets). In the numbers shown for 8 threads, we see that most of the gains are coming mainly from the improvements to locality (similar behavior was observed for 1, 2, and 4 number of threads), reduction in activities further improves the code. For the case with 64 threads, it can be seen that the locality version may improve the performance depending on the underlying computation (for instance, in MG, and MolDyn). The gains in the "opt" version here are significant enough to show improvements, irrespective of the impact due to locality. For benchmarks like CG, SOR, and LUFact most of the benefits are coming mainly from reduction in the number of tasks and barriers. We have observed similar behavior for 16, 32, and 128 number of hardware threads, thus emphasizing the importance of reducing task creation overhead in the context of systems with higher number of cores/hardware threads.

We conclude that the direct impact from the reduction in activities and barriers is significant, and the `forall`-coarsening may also aid in improving the data locality (may be significant when all the threads share the L1 cache).

8.5. Impact Due to Finish Elimination

This section demonstrates the impact of our `finish`-*elimination* by discussing both the static and dynamic results. We present two metrics: the number of eliminated `finish` operations (static and dynamic) and the performance improvement. Similar to the behavior of the previous two optimizations, even the `finish`-*elimination* optimization may not be universally effective; `finish`-*elimination* has been typically found effective in programs where parallel tasks are spawned conditionally, such as some system-specific threshold. Such instances are common in programs written using the divide-conquer pattern; the recursive nature of the parallel program makes it otherwise tricky to optimize using an automated tool like a compiler. Table IV shows the dynamic count of the finish operations before and after finish-elimination among the

Table IV. Dynamic Counts of Finish Operations for Unoptimized and Optimized Code

	lud	floorplan	health	strassen	fft
no `finish`-elim	17,082	3,613,785	2,124,789	400	6,959
with `finish`-elim	13,176	622	6,588	8	4,634

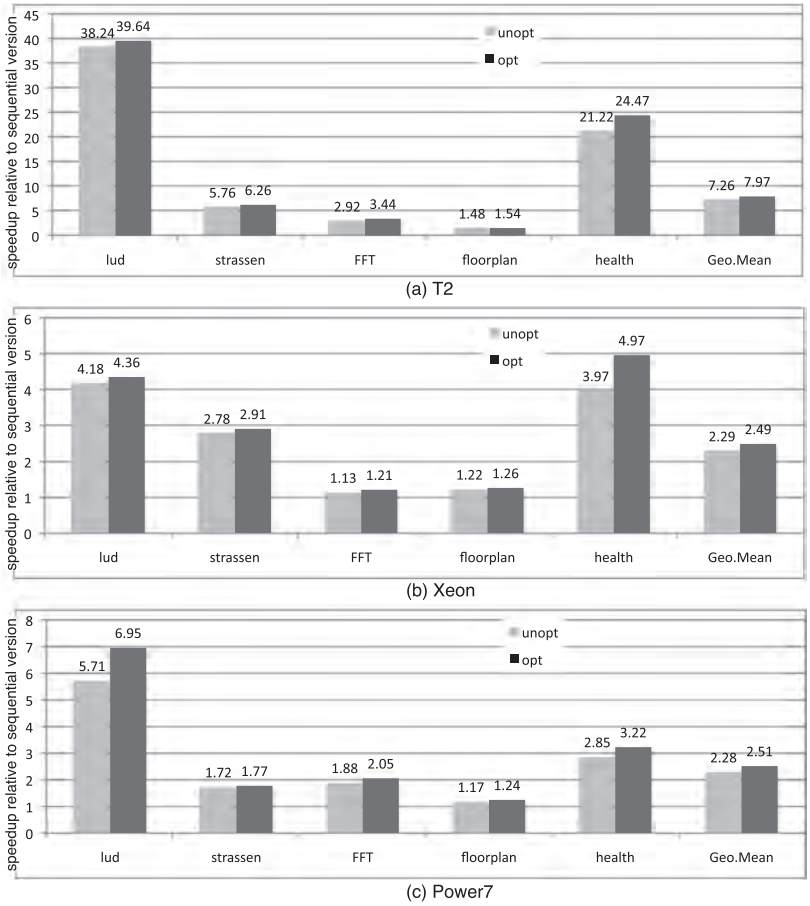

Fig. 35. Performance improvement by finish-elimination. "unopt": enables `forall`-coarsening and loop-chunking but disables finish-elimination, "opt": enables all three optimizations.

selected benchmarks. As the table shows, the `finish`-*elimination* pass can significantly reduce the dynamic number of finish statements in many cases.

Figure 35 gives the performance comparison between the programs optimized with `finish`-elimination + `forall`-coarsening + loop-chunking (tagged as "opt") and programs optimized with only `forall`-coarsening + loop-chunking (tagged as "unopt"). The actual performance improvement depends on: (a) the number of dynamic finish operations eliminated and the cost of finish operation on that architecture for the underlying runtime system[8] and (b) the number of tasks spawned within a finish

[8]We measured the overhead of an empty finish statement as 6 microseconds, 35 microseconds, and 6 microseconds on the Xeon, Niagara, and Power7 platforms, respectively, for the same runtime systems as those used to obtain the experimental results shown in Figure 35.

region. Compared to the "unopt" version the geometric mean average improvements of the "opt" version are 1.10×, 1.09×, and 1.10× on T2, Xeon, and Power7, respectively. Item (a) impacted all benchmarks, especially `health`. Item (b) impacted `lud`, `strassen`, `health`, and `fft`; elimination of redundant `finish` operations enlarged the parallel tasks and resulted in better load balance for these benchmarks. The `floorplan` benchmark showed the smallest speedups and the least improvement due to optimization because it contains `isolated` constructs that limit the available parallelism.

Across all the benchmarks, our proposed transformations have not resulted in any performance degradation on any of the platforms. Further, it can be seen that all the three proposed optimizations work in a synergistic way to derive significant performance benefits.

9. RELATED WORK

We divide the related work into four different subsections, one for each of the main contributions of this article.

9.1. Analysis of Task-Parallel Programs

Happens-Before Analysis. The happens-before relationship was first studied by Lamport [1978] in the context of distributed systems. It has been widely used for parallel computing, especially in the areas related to concurrency analysis and data race detection. Duesterwald and Soffa [1991] applied happens-before/happens-after information to the context of dataflow analysis for concurrent programs. Their framework expresses a partial execution ordering for program regions that have happens-before/happens-after relation. In our work, we generalized the happens-before relation to define happens-before dependency, which is used to build legal program transformations.

Data-Dependence Analysis. Data-dependence analysis for sequential programs has been extensively studied [Kennedy and Allen 2002; Wolfe and Banerjee 1987], and those techniques have been widely applied. We extend the traditional notions of data dependence to happens-before dependence by taking into account features in task-parallel programs. We present a set of constraints (which depend on not only the happens-before dependence information, but also the program structure) to ensure the legality of our proposed compiler transformation.

9.2. Barrier Synchronization and Task Creation optimization

Many previous studies have optimized parallel loops to reduce task spawning and synchronization overheads [Heinz and Philippsen 1993; Tseng 1995; Yonezawa et al. 2006]. Compared to these works that optimize only parallel loops, we have built a general compiler framework that focuses on eliminating arbitrary redundant `finish` barrier operations by applying sophisticated analysis and transformations. The use of global split barriers [Bikshandi et al. 2009] as an efficient translation of outermost finish operations can be used to further improve the performance.

Nicolau et al. [2009] presented an approach to optimize point-to-point synchronization by eliminating redundant `wait` operations and relocating `post`/`wait` operations to minimize barrier overhead. Compared to their approach, we present an interprocedural transformation that optimizes arbitrary `finish` barriers in task-parallel languages. Further, we present a transformation scheme that preserves exception semantics.

Bikshandi et al. [2009] present the notion of *inlinable* `async` statements for which they avoid the activity creation overhead. They make a static compile-time decision to serialize an `async` based on the structure of the body of `async`. Compared to that, our

proposed `seq` clause helps make dynamic decisions on serializing an `async` based on programmer-decided constraint or runtime resources.

9.3. Forall-Coarsening

There has been a lot of past work on reducing thread creation and synchronization overheads. These include SPMDization [Amarasinghe and Lam 1993; Bikshandi et al. 2009; Cytron et al. 1990; Tseng 1995], synchronization optimizations [Diniz and Rinard 1997], and barrier elimination [Tseng 1995]. Cytron et al. [1990] present an approach for transforming code written in fork-join style to SPMD code. Tseng [1995] furthers the work of Cytron et al. by translating fork-join parallel loops into (merged) SPMD regions. Once SPMD regions have been formed, the barrier communications among them are targeted for optimization using communication analysis. Our `forall`-coarsening has similarities to the traditional SPMDization techniques. Some of the rules like parallel loop fusion and serial-parallel loop interchange used in Section 4.2 are similar to the translation scheme suggested by Tseng [1995]. However, there are three main differences: (a) While their target is to reduce the number of synchronization operations, our main goal is to reduce the number of dynamic activities created, thus our rules are more aggressive; (b) the result of our transformation is a task-parallel program that can contain fork (`async`) and join (`finish`) operations, and is not necessarily an SPMD program; (c) we handle programs with exceptions and perform further cleanup optimizations to gain performance.

Recently, Bikshandi et al. applied SPMDization to task-parallel languages [Bikshandi et al. 2009], where they identify a subset of `X10` (called `Flat X10`) and use it to derive output programs in SPMD form. In our work, we preserve the task-parallelism language features and perform the translation implicitly in the compiler back-end. Further, we handle programs with arbitrary `async` operations, `forall` loops, and exceptions.

Nicolau et al. [2009] present an approach to optimize point-to-point synchronization by eliminating redundant `wait` operations. Their approach has similarities only to our postoptimization pass, where we eliminate some redundant barriers.

Ferrer et al. [2009] present techniques to unroll sequential loops that contain parallel loops. They aggregate the multiple generated loops in the body of the sequential unrolled loop to reduce the number of activity creation tasks. Our `forall`-coarsening phase can be invoked as a postpass to their phase to further increase the gains.

9.4. Chunking of Parallel Loops

There has been a lot of past work on reducing synchronization and thread creation overheads. These include SPMDization [Bikshandi et al. 2009], synchronization optimizations [Diniz and Rinard 1997], and barrier elimination [Tseng 1995]. Researchers have studied the impact of loop chunking on different parameters of interest. Hari et al. [Narayanan et al. 2005] use loop chunking as a means of efficient scheduling of temperature-aware code. OpenMP 3.0 [OpenMP 2008] supports different loop scheduling policies, as specified by the programmer, in parallel loops. However, the OpenMP language framework is restrictive in its support for synchronization operations inside parallel loops.

There has also been significant interest in loop scheduling [Kennedy and Allen 2002]. Akin to chunking, loop scheduling has been directed at reducing the number of overall barriers and thread creation overheads. The loop scheduling techniques also use different loop transformation techniques (for example, loop interchange and loop coarsening) to identify chunks of iterations that can be scheduled together. Loop-chunking can be seen as a special version of loop scheduling where all the iterations scheduled to be executed on the same processor are executed sequentially.

We are not aware of any past work that supports chunking of parallel loops in the presence of synchronization, as in this article, for languages that support dynamic parallelism with fine-grain synchronization.

10. CONCLUSION

In this article, we present a transformation framework for optimizing task-parallel programs. Our framework includes: (a) `finish`-elimination: an iterative algorithm to eliminate the redundant termination operations and increase parallelism, (b) `forall`-coarsening: a scheme to replace task creation/termination optimizations by lighterweight barrier synchronizations, and (c) loop-chunking: an iterative algorithm to realize useful parallelism from given specifications of ideal parallelism by chunking parallel loops. All of these transformations preserve the exception semantics. To ensure the legality of transformation, we presented a definition of data dependence in task-parallel programs and a happens-before dependence analysis algorithm.

Experimental results were obtained for a collection of task-parallel benchmarks on three platforms: a dual-socket 128-thread (16-core) Niagara T2 system, a quad-socket 16-core Intel Xeon SMP, and a quad-socket 32-core Power7 SMP. These results show geometric average performance improvements of 6.56×, 6.28×, and 9.77× on the three platforms, respectively, due to the optimizations introduced in this article. For certain benchmarks for which the original versions were highly inefficient, the maximum improvements on these three platforms ranged from 1103.90× to 3935.88×. Though these results were obtained in the context of HJ, we are confident of deriving similar improvements in other task-parallel programming languages such as X10, Chapel, and OpenMP, among others.

ACKNOWLEDGMENTS

We would like to thank the members of the Habanero group at Rice University and the X10 team at IBM for valuable discussions related to this work. We would like to thank the anonymous reviewers for their comments and suggestions on past conference publications and submissions related to this work. In particular, the example in Figure 8(a) was provided by one of the reviewers. Finally, we would like to thank Keith Cooper for providing access to the Xeon system, Doug Lea for providing access to the UltraSPARC T2 system, and the Research Computing Support Group at Rice University for providing access to the POWER7 system used to obtain the performance results reported in this article.

REFERENCES

Agarwal, S., Barik, R., Sarkar, V., and Shyamasundar, R. K. 2007. May-Happen-In-Parallel analysis of x10 programs. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'07)*. ACM Press, New York, 183–193.

Amarasinghe, S. P. and Lam, M. S. 1993. Communication optimization and code generation for distributed memory machines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'93)*. ACM Press, New York, 126–138.

Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., Simon, H. D., Venkatakrishnan, V., and Weeratunga, S. K. 1991. The nas parallel benchmarks - Summary and preliminary results. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. 158–165.

Bikshandi, G., Castanos, J. G., Kodali, S. B., Nandivada, V. K., Peshansky, I., Saraswat, V. A., Sur, S., Varma, P., and Wen, T. 2009. Efficient, portable implementation of asynchronous multi-place programs. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'09)*. ACM Press, New York, 271–282.

Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. 1995. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not. 30*, 8, 207–216.

Chamberlain, B. L., Eun Choi, S., Deitz, S. J., And Snyder, L. 2004. The high-level parallel language ZPL improves productivity and performance. In *Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing*. IEEE, 66–75.

Chapel. 2005. Chapel. 2005. The chapel language specification version 0.4. http://chapel.cray.com/spec/spec-0.4.pdf.

Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., Von Praun, C., and Sarkar, V. 2005. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*. ACM Press, New York, 519–538.

Cytron, R., Lipkis, J., and Schonberg, E. 1990. A compiler-assisted approach to SPMD execution. In *Proceedings of the ACM/IEEE Conference on Supercomputing (Supercomputing'90)*. IEEE Computer Society Press, Los Alamitos, CA, 398–406.

Dean, J., Grove, D., and Chambers, C. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*. Springer, 77–101.

Diniz, P. and Rinard, M. 1997. Synchronization transformations for parallel computing. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*. ACM Press, New York, 187–200.

Duesterwald, E. and Soffa, M. L. 1991. Concurrency analysis in the presence of procedures using a data-flow framework. In *Proceedings of the Symposium on Testing, Analysis, and Verification (TAV4)*. ACM Press, New York, 36–48.

Duran, A., Teruel, X., Ferrer, R., Martorell, X., and Ayguade, E. 2009. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *Proceedings of the International Conference on Parallel Processing (ICPP'09)*. IEEE Computer Society, Los, Alamitos, CA, 124–131.

Feng, M. and Leiserson, C. E. 1997. Efficient detection of determinacy races in cilk programs. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'97)*. ACM Press, New York, 1–11.

Ferrer, R., Duran, A., Martorell, X., and Ayguade, E. 2009. Unrolling loops containing task parallelism. In *Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science, vol. 5898, Springer, 416–423.

Flynn, L. E. and Hummel, S. F. 1990. Scheduling variable-length parallel subtasks. Tech. rep. RC 15492, IBM.

Gao, G. R. and Sarkar, V. 2000. Location consistency-A new memory model and cache consistency protocol. *IEEE Trans. Comput. 49*, 798–813.

Georges, A., Buytaert, D., and Eeckhout, L. 2007. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (OOPSLA'07)*. ACM Press, New York, 57–76.

Guo, Y., Barik, R., Raman, R., and Sarkar, V. 2009. Work-First and help-first scheduling policies for async-finish task parallelism. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*. IEEE Computer Society, Los Alamitos, CA, 1–12.

Gupta, R. 1989. The fuzzy barrier: A mechanism for high speed synchronization of processors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM Press, New York, 54–63.

Habanero. 2009. Habanero Java. http://habanero.rice.edu/hj.

Heinz, E. A. and Philippsen, M. 1993. Synchronization barrier elimination in synchronous foralls. Tech. rep. 13/93, Department of Informatics, University of Karlruhe.

JGF. 2000. The java grande forum benchmark suite. http://www.epcc.ed.ac.uk/javagrande/javag.html.

Kennedy, K. and Allen, J. R. 2002. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann, San Francisco, CA.

Kruskal, C. and Weiss, A. 1985. Allocating independent subtasks on parallel processors. *IEEE Trans. Softw. Engin. SE-11*, 10.

Lamport, L. 1978. Time clocks, and the ordering of events in a distributed system. *Comm. ACM 21*, 558–565.

Lamport, L. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput. 28*, 690–691.

Larus, J. R. and Rajwar, R. 2006. *Transactional Memory*. Morgan and Claypool.

Lhotak, O. and Hendren, L. 2003. Scaling java points-to analysis using spark. In *Proceedings of the 12th International Conference on Compiler Construction (CC'03)*. Springer, 153–169.

Metcalfe, M. and Reid, J. 1990. *Fortran 90 Explained*. Oxford Science Publishers.

Muchnick, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA.

Narayanan, S. H. K., Chen, G., Mahmut Kandemir, M. X., and Xie, Y. 2005. Temperature-Sensitive loop parallelization for chip multiprocessors. In *Proceedings of the International Conference on Computer Design (ICCD'05)*. IEEE Computer Society, Los Alamitos, CA, 677–682.

Nicolau, A., Li, G., Veidenbaum, A. V., and Kejariwal, A. 2009. Synchronization optimizations for efficient execution on multi-cores. In *Proceedings of the 23rd International Conference on Supercomputing (ICS'09)*. ACM Press, New York, 169–180.

Nystrom, N., Clarkson, M. R., and Myers, A. C. 2003. Polyglot: An extensible compiler framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction (CC'03)*. Springer, 138–152.

OpenMP. 2008. OpenMP application program interface, version 3.0. http://www.openmp.org/mpdocuments/spec30.pdf

Peierls, T., Goetz, B., Bloch, J., Bowbeer, J., Lea, D., and Holmes, D. 2005. *Java Concurrency in Practice*. Addison-Wesley Professional.

Polychronopoulos, C. D. and Kuck, D. J. 1987. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Trans. Comput. C-36*, 12.

Salcianu, R. D. and Rinard, M. C. 2005. Purity and side effect analysis for Java programs. In *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*.199–215.

Sarkar, V. 1988. Synchronization using counting semaphores. In *Proceedings of the 2nd International Conference on Supercomputing (ICS'88)*. ACM Press, New York, 627–637.

Sarkar, V. and Fink, S. J. 2001. Efficient dependence analysis for Java arrays. In *Proceedings of the 7th International Euro-Par Conference on Parallel Processing (Euro-Par'01)*. Springer, 273–277.

Shirako, J., Peixotto, D. M., Sarkar, V., and Scherer, W. N. 2008. Phasers: A unified deadlock-free construct for collective and point-to-point synchronization. In *Proceedings of the 22nd Annual International Conference on Supercomputing (ICS'08)*. ACM Press, New York, 277–288.

Shirako, J., Zhao, J. M., Nandivada, V. K., and Sarkar, V. N. 2009. Chunking parallel loops in the presence of synchronization. In *Proceedings of the 23rd International Conference on Supercomputing (ICS'09)*. ACM Press, New York, 181–192.

Tseng, C.-W. 1995. Compiler optimizations for eliminating barrier synchronization. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'95)*. ACM Press, New York, 144–155.

Vallee-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. 1999. Soot - A java bytecode optimization framework. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'99)*. IBM Press, 125–135.

Wolfe, M. 1996. *High Performance Compilers for Parallel Computing*. Addison-Wesley.

Wolfe, M. and Banerjee, U. 1987. Data dependence and its application to parallel processing. *Int. J. Parallel Program. 16,* 137–178.

Yelick, K., Bonachea, D., Chen, W.-Y., Colella, P., Datta, K., Duell, J., Graham, S. L., Hargrove, P., Hilfinger, P., Husbands, P., Iancu, C., Kamil, A., Nishtala, R., Su, J., Michael, W., and Wen, T. 2007. Productivity and performance using partitioned global address space languages. In *Proceedings of the International Workshop on Parallel Symbolic Computation (PASCO'07)*. ACM Press, New York, 24–32.

Yonezawa, N., Wada, K., And Aida, T. 2006. Barrier elimination based on access dependency analysis for OpenMP. In *Proceedings of the 4th International Symposium on Parallel and Distributed Processing and Applications (ISPA'06),* M. Guo, L. T. Yang, B. D. Martino, H. P. Zima, J. Dongarra, and F. Tang Eds., Lecture Notes in Computer Science, vol. 4330, Springer, 362–373.

Zhao, J. and Sarkar, V. 2011. Intermediate language extensions for parallelism. In *Proceedings of the 5th Workshop on Virtual Machines and Intermediate Languages (VMIL'11)*. 333–334.

Zhao, J., Shirako, J., Nandivada, V. K., and Sarkar, V. 2010. Reducing task creation and termination overhead in explicitly parallel programs. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*. ACM Press, New York, 169–180.

Received December 2011; revised October 2012; accepted November 2012