Reducing Task Creation and Termination Overhead in Explicitly Parallel Programs

Jisheng Zhao
Dept. of CS, Rice University
6100 Main St, Houston TX, USA
jisheng.zhao@rice.edu

Jun Shirako
Dept. of CS, Rice University
6100 Main St, Houston TX, USA
shirako@rice.edu

V. Krishna Nandivada
IBM India Research Laboratory
EGL, Bangalore, 560071, India
nvkrishna@in.ibm.com

Vivek Sarkar
Dept. of CS, Rice University
6100 Main St, Houston TX, USA
vsarkar@rice.edu

ABSTRACT

There has been a proliferation of task-parallel programming systems to address the requirements of multicore programmers. Current production task-parallel systems include Cilk++, Intel Threading Building Blocks, Java Concurrency, .Net Task Parallel Library, OpenMP 3.0, and current research task-parallel languages include Cilk, Chapel, Fortress, X10, and Habanero-Java (HJ). It is desirable for the programmer to express all the parallelism intrinsic to their algorithm in their code for forward scalability and portability, but the overhead incurred by doing so can be prohibitively large in today's systems. In this paper, we address the problem of reducing the total amount of overhead incurred by a program due to excessive task creation and termination. We introduce a transformation framework to optimize task-parallel programs with *finish*, *forall* and *next* statements. Our approach includes elimination of redundant task creation and termination operations as well as strength reduction of termination operations (*finish*) to lighter-weight synchronizations (*next*). Experimental results were obtained on three platforms: a dual-socket 128-thread (16-core) Niagara T2 system, a quad-socket 16-way Intel Xeon SMP and a quad-socket 32-way Power7 SMP. The results showed maximum speedup of 66.7×, 11.25× and 23.1× respectively on each platform and 4.6×, 2.1× and 6.4× performance improvements respectively in geometric mean related to non-optimized parallel codes. The original benchmarks in this study were written with medium-grained parallelism; a larger relative improvement can be expected for programs written with finer-grained parallelism. However, even for the medium-grained parallel benchmarks studied in this paper, the significant improvement obtained by the transformation framework underscores the importance of the compiler optimizations introduced in this paper.

Categories and Subject Descriptors:
D.3.4 [**Programming Languages**] Optimizations D.3.4 [**Programming Languages**] Compiler

PACT'10, September 11–15, 2010, Vienna, Austria.

General Terms:
Algorithms, Experimentation

Keywords:
Optimization, redundant tasks, barriers, useful parallelism, ideal parallelism

1. INTRODUCTION

Modern languages such as Cilk [5], Chapel [13], Fortress [1], `X10` [9], and Habanero-Java (HJ) [12] have moved from Single Program Multiple Data (SPMD) models to lightweight dynamic Task Parallel execution models for improved programmer productivity. This shift encourages programmers to express the *ideal* parallelism in an application at a fine granularity that is natural for the underlying domain, while the job of extracting coarser-grained *useful* parallelism for a given target system is delegated to the compiler and runtime system.

The fine-grain parallelism specified by the programmer may lead to the creation of excessive tasks and synchronization operations. A common occurrence of this overhead can be seen for parallel loops nested in sequential loops. Figure 1(a) shows a common pattern found in the Java Grande Forum [14] benchmarks and the NAS Parallel benchmarks [8]. The `forall` construct is a parallel loop that creates one task per iteration and waits (via an implicit finish operation) for all of the tasks to terminate before proceeding to the statement following the `forall`. In Figure 1(a), a total of `m`×`n` tasks are created and `n` finish (join) operations are performed. A possible translation (assuming that dependences permit) is shown in Figure 1(b). In this version, only `m` tasks are created and just one join operation is performed; for large values of `n` the performance impact of this transformation can be overwhelming. In both cases, the number of parallel tasks at any point of execution is bounded by `m` so the decrease in overhead is not accompanied by a loss in parallelism. In this paper, we propose a novel transformation framework that helps reduce such task creation and termination overhead by *distilling* `forall` statements in two ways, *fine-grained* and *coarse-grained*, which move `forall` statements from outside-in and inside-out respectively[1]. This framework is presented in the context of *finish*, *forall* and *next* statements in the HJ research language [12], but is also applicable to other task-parallel programming models with similar constructs.

To motivate the challenges in `forall` distillation, Figure 2(a) shows the pedagogical One-Dimensional Iterative Averaging pro-

[1]We discuss differences between our transformation framework and past work on SPMDization transformations in Section 7.

```
for(i=0;i<n;++i){
 forall(j: [1..m]){
    S } }
        (a)
```

```
forall(j: [1..m]){
 for(i=0;i<n;++i){
    S } }
        (b)
```

Figure 1: (a) A predominant pattern found in multiple benchmarks. (b) A possible translation of the pattern (assuming that dependences permit).

gram discussed in [17]. The `forall` loop creates `n` parallel tasks to execute the loop body. These `n` tasks terminate and *join* at the end of the `forall` loop. These task creations and terminations are repeated until the termination of the `while` loop. Since the `while` loop may be repeated a large number of times, such a program incurs a large overhead in terms of task creation and termination. A naive attempt to move the `forall` header outside the serial loop (as shown in Figure 2(b)) would lead to an incorrect translation: in this example, the computation outside the `forall` (`sum and exchange`) in Figure 2(a) should be executed only once per each iteration of the serial loop, and only after the termination of the `forall` loop. In the translated program shown in Figure 2(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. Another problem with the code shown in Figure 2(b) is that there is a data race on `A` and `newA` among the parallel iterations of the `forall` loop, and thus needs to be remediated by inserting additional synchronization operations[2].

In this paper, we address the problem of safe optimization of task creation and termination overheads by `forall` distillation. It uses a combination of classical transformations (for example, loop unswitching, distribution, interchange, fusion) and new transformations to obtain a semantically equivalent translation with reduced overhead. For the classical transformations, a key difference from past work is that the transformations have to be performed on explicitly parallel programs in our case rather than sequential programs. Specifically, our paper makes the following contributions:

1. Fine-grain `forall` distillation: a transformation scheme that reduces the task creation/termination overhead without introducing any additional barriers.

2. Coarse-grain `forall` distillation: a transformation scheme that replaces task creation/termination operations by lighter-weight barrier synchronizations.

3. Redundant Next/Next-Single Elimination (RNSE): a new algorithm for elimination and strength reduction of barrier operations.

4. Preservation of exception semantics: the transformation framework in this paper respects the exception semantics of the HJ language (derived from the `X10` v1.5 exception model [9]).

5. Experimental results: this framework has been implemented within the HJ compilation system [12]. Our experimental results for five different benchmarks on two different SMP machines show that (a) compared to the performance of the serial benchmarks on a 2-socket 128-thread (16-core) Niagara T2 SMP, the original medium-grained parallel benchmarks achieve a geometric mean speedup of 10.7×, whereas `forall` distillation of the parallel benchmark achieves a geometric mean speedup of over 48.5×. Similarly, on a 4-socket 16-core Intel Xeon SMP, the geometric mean speedups without and with our optimization are 3.4× and 7.0× respectively. (b) `forall` distillation leads to increased scalability with increasing number of processors. (c) `forall` distillation has a positive impact on other optimizations such as loop chunking [20].

Our initial focus is on reducing the task creation and termination overhead in parallel loops because they are commonly used in many parallel applications, and make a convincing case for performance improvement. However, our overall framework, including support for exceptions, should be extensible to other forms of task parallelism as well. The novelty of our contributions includes three aspects: (i) transformation of explicitly parallel programs using a systematic approach consisting of fine-grain and coarse-grain `forall` distillations, (ii) redundant next-single elimination (RNSE), and (iii) extensions for exception semantics. To the best of our knowledge, there has been no past work with any of these three attributes.

In Section 2, we introduce the basic HJ language constructs and the loop transformations used in this paper. Section 3 introduces the proposed transformation framework. Section 4 discusses extensions to the framework discussed in Section 3 to support exceptions in the context of explicit parallelism. Section 5 discusses key implementation issues. The empirical evaluation is presented in Section 6. We discuss related work in Section 7, and conclude in Section 8.

2. BACKGROUND

2.1 Habanero Java (HJ)

Our input programs are written in HJ [12], which extends the Java-based version 1.5 of the `X10` programming language [9] with phasers [15] and other modifications. This section provides a brief summary of HJ's `async`, `finish`, `phaser` and `forall` constructs. This paper focuses primarily on optimizing the `forall` and phaser `next` statements, and its results are applicable to any programming model with similar constructs. Additional HJ constructs that are not central to the paper have been omitted for simplicity.

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*⟩ (logically) in parallel with the parent task. ⟨*stmt*⟩ is permitted to read/write any data in the heap and to read any final local variable belonging to the parent task's lexical environment.

finish

The HJ statement `finish` ⟨*stmt*⟩ causes the parent task to execute ⟨*stmt*⟩ and then wait till all sub-tasks created within ⟨*stmt*⟩ have terminated (including transitively spawned tasks). Operationally, each instruction executed in an HJ task has a dynamically unique *Immediately Enclosing Finish* (IEF) statement instance [15].

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 *multi-exception* [20] formed from the collection of all exceptions thrown by all abruptly-terminating tasks in the IEF. (This is in contrast with the Java model where an exception is simply propagated from a thread to the top-level console.)

[2]Arbitrary usage of barriers can lead to additional challenges [20] e.g., OpenMP prohibits a barrier region from being nested inside a loop region.

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

(a)

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

(b)

Figure 2: (a) One-dimensional iterative averaging example. (b) Naive `forall` distillation may be semantically incorrect.

phasers

The *phaser* construct [15] integrates collective and point-to-point synchronization by giving each task the option of registering with a phaser in *signal-only/wait-only* mode for producer/consumer synchronization or *signal-wait* mode for barrier synchronization. (The optimizations in this paper only use *signal-wait* mode.) In general, a task may be registered on multiple phasers, and a phaser may have multiple tasks registered on it. Two operations on *phasers* that are most relevant to this paper are:

• *new:* When an task A_i performs a `new phaser()` operation, it results in the creation of a new phaser ph such that A_i is registered with ph.

• *next:* The `next` operation has the effect of advancing each phaser on which the invoking task A_i is registered to its next phase, thereby synchronizing all tasks registered on the same phaser. In addition, a `next` statement for phasers can optionally include a *single* statement, `next {S}`. This guarantees that the statement `S` is executed exactly once during the phase transition [23, 15]. The exception semantics for the single statement was unspecified in [15]. We define the exception semantics of the single statement as follows: an exception thrown in the single statement causes all the tasks blocked on that `next` operation to terminate abruptly with a single instance of the exception thrown to the IEF task[3].

forall

The statement `forall (point p : R) S` supports parallel iteration over all the points in region `R` by launching each iteration as a separate `async`, and including an implicit `finish` to wait for all of them to terminate. 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 range as in the case of `forall`.

Each dynamic instance of a `forall` statement includes a distinct phaser object (say, `ph`) that is created implicitly, and is set up so that all iterations in the `forall` are registered on `ph` in *signal-wait* mode[4]. Since the scope of `ph` is limited to the implicit finish in the `forall`, the parent task will drop its registration on `ph` after all the `forall` iterations are created.

2.2 Classical Loop Transformations

This section briefly summarizes some classical loop restructuring techniques that have historically been used to optimize sequential programs [16]:

• **Loop Interchange** results in a permutation of the order of loops in a loop nest, and can be used to improve data locality, coarse-grained parallelism and vectorization opportunities.

• **Loop Distribution** divides the body of a loop into 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 loop-level parallelism.

• **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.

• **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.

3. TRANSFORMATION FRAMEWORK

In this section, we present our transformation framework to reduce the task creation and termination overhead in HJ programs. To simplify the presentation, we will first focus on the restricted case when the code under consideration is known to be exception-free. Later in Section 4, we discuss transformation in the presence of exceptions.

We introduce a new compiler optimization phase called `forall` *distillation*. In the HJ program snippet shown in Figure 3(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 *distill* `forall` statements from within `for` loops and `while` loops. Depending on the actual program code, different translations are possible; Figure 3(b) and Figure 3(c) show two translations that distill the `forall` loop in Figure 3(a). We call the first translation a *fine grain* `forall` distillation, and the second one a *coarse grain* `forall` distillation. While both translations are more efficient than the original code, the translation in Figure 3(b) is more efficient than that in Figure 3(c). However, dependences in different part of the code may (or may not) permit either of the translations.

Since the feasibility of either of these translations (fine-grain or coarse grain) depends on the input program, we adopt a two-pronged strategy for `forall` distillation, as shown in the overall block-diagram in Figure 4: first we apply a set of transformations to do fine-grain `forall` distillation. After that, we apply the set of transformations to do coarse-grain `forall` distillation. Finally, we apply some cleanup optimizations to further optimize the generated code. The different sets of transformations in each of these two phases are monotonic — though they may be applied in any order, the resulting transformed code is guaranteed to be the same. We now present the details of each of these phases.

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

[4]For readers familiar with the `foreach` statement in `X10` and HJ, one way to relate `forall` to `foreach` is to think of `forall` ⟨*stmt*⟩ as syntactic sugar for "`ph=new phaser(); finish foreach phased (ph)` ⟨*stmt*⟩".

```
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)

Figure 3: (a) Example program, (b) Fine-grain `forall` distillation: does not need any additional barriers (assuming that dependences permit), (c) Coarse-grain `forall` distillation: requires additional barriers (`next` statements), but is always legal.

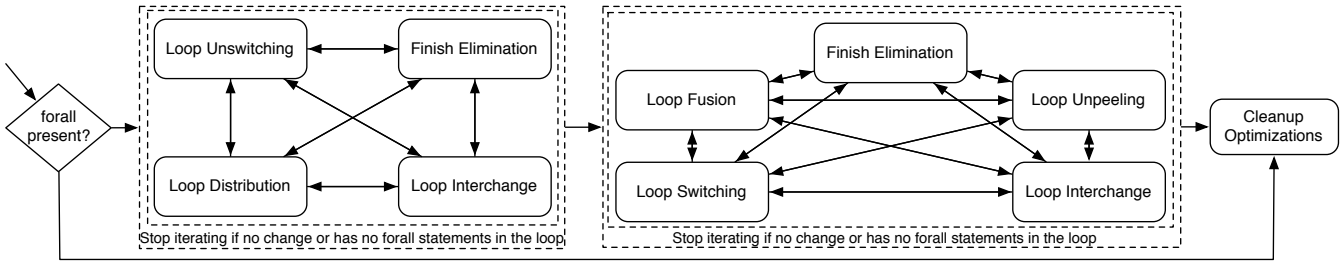

Figure 4: Block diagram for Transformation Framework

3.1 Fine-grain forall distillation

The rules for fine-grain distillation are given in Figure 5. Intuitively, fine-grain `forall` distillation localizes the scope of a `for` loop so as to bring it as close to the inner `forall` loop as possible[5]. For any `for` loop, we repeatedly apply loop distribution, loop unswitching, finish elimination, and loop interchange until (a) no `forall` statement occurs in the body of `for` loops, or (b) no further change is possible.

Loop Interchange

Loop interchange is the key transformation to realize the distillation. To effect this transformation, we assume that the iteration space of the two loops are not dependent on each other. Note that, we do not stop the distillation pass after a successful loop interchange. We keep iterating in search of further gains.

Loop Unswitching, Loop Distribution, and Finish Elimination

Our loop interchange rule discussed above requires that the body of the `for` loop should consist of only a `forall` loop. The aim of the other rules described in Figure 5 is to help fulfill that requirement. We now briefly describe these rules; the assumptions associated with each rule work as preconditions for the rule to be applied.

Loop Unswitching is the classical unswitching transformation for sequential code [16]. The main assumption here is that the predicate `e` is loop invariant.

Loop Distribution is the classical loop distribution transformation for sequential code [16], and is the most important transformation in fine-grain distillation process to avoid the insertion of barriers. The main requirement to apply this transformation is that there is no dependence cycle between the `S1` and `S2`. That is, there should not be case where `S1` depends on `S2` and `S2` depends on `S1`; such a cycle would render the distribution semantically incorrect.

[5]The same approach can be applied to a limited set of while loops, as in Figure 2.

Finish Elimination eliminates the redundant `finish` around a `forall` statement. Such a `finish` is redundant because of the implicit `finish` within the `forall` statement.

3.2 Coarse-grain forall distillation

The rules for coarse-grain distillation are given in Figure 6. Intuitively, coarse-grain `forall` distillation expands the scope of a `forall` so as to bring it as close to the outer `for` loop as possible. For any `forall` loop, we repeatedly apply loop unpeeling, loop fusion, loop switching, finish elimination, and loop interchange until (a) no `forall` statement occurs in the body of `for` loops, or (b) no further change is possible. The idea behind coarse-grain distillation is to replace task creation/termination operations by lighter-weight barrier synchronizations. This enables the programmer to express parallelism at a fine-grained task level, and leave it to the compiler and runtime to map the parallelism to a coarser level that can be implemented more efficiently.

The finish elimination and loop interchange transformations are identical in both coarse-grain and fine-grain `forall` distillation.

Loop Unpeeling, Loop Fusion and Loop Switching

As shown in Figure 6, *Loop Unpeeling* 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 builds on the classical loop fusion transformation for sequential code [16]. It merges two `forall` statements by fusing their bodies, and inserting a `next` (barrier) statement in between. Both of these rules (unpeeling and fusion) use the implicit phaser associated with `forall`.

Loop Switching is based on the inverse of classical loop unswitching transformation discussed in Section 3.1. It expands the scope of the `forall` loop by bringing an `if` statement inside the body of the loop.

3.3 Cleanup Optimizations and Discussion

• The `forall` distillation techniques explained in the prior section

1. Loop Unswitching:

```
for (i: [1..n])
  if (e)   // e is loop invariant
    S;
```
$\Longrightarrow$
```
if (e)
  for (i: [1..n])
    S;
```

2. Loop Distribution:

```
for (i: [1..n])
  { S1; S2; }   // No dependence cycle between S1 and S2
```
$\Longrightarrow$
```
for (i: [1..n]) S1;
for (i: [1..n]) S2;
```

3. Finish Elimination:

```
finish
  forall (point p: R)
    S;
```
$\Longrightarrow$
```
forall (point p: R)
  S;
```

4. Loop interchange:

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

Figure 5: Fine-grain `forall` distillation: rules for loop unswitching, loop distribution, finish elimination and loop interchange

Inter procedural Loop interchange:

```
for (i : [1..n])
 foo();

void foo () {
 forall(point p:R)
 // n does not depend on p
 // R does not depend on i
  S; }
```
$\Longrightarrow$
```
forall(point p:R)
 for (i: [1..n])
  foo();

void foo() {
 S;
}
```

Figure 7: Sample inter-procedural translation rule.

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) that is similar to the synchronization elimination algorithm in [18]. 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 the loop chunking phase explained in [20] to further improve the performance.

• We invoke a post-pass of copy propagation and dead-code assignment elimination, and loop fusion (Rule 2, Figure 6) that helps us further fine tune our output.

• We make a simple inter-procedural extension to all the transformation rules described above. We present a sample inter-procedural transformation for loop interchange in Fig. 7. The remaining rules are omitted due to space limitations.

While the coarse-grain and the fine-grain distillation phases explained in this section consist of multiple transformations, only two of them (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 loop interchange and loop fusion. Traditional Loop interchange has a known history of impact on the cache behavior. For example, loop interchange on the example given below can improve the cache performance of accessing `b[j,i]`, but it will ruin the reuse of `a[i]` and `c[i]` (by introducing memory loads for `a[i]` and `c[i]`, and memory stores for `a[i]` in each iteration).

```
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 can also have a negative effect on load balancing (as the total parallelism is reduced). Identifying the optimal task size is quite challenging a problem in itself and is beyond the scope of this paper. The compiler that invokes our distillation phase is assumed to know the maximum allowed task size, and accordingly can control the distillation phase to generate tasks with optimal size.

We now present the effect of invoking our framework on an input program shown in Figure 8(a). Figures 8(b-h) show the results of applying our transformations on the input program. Figure 9 presents a flow chart explaining the interaction between the different transformations that are invoked in this process. As described in Figure 4, the fine-grain transformation is applied at first. There is no cyclic dependency between `S1` and the rest of the loop body; thus enabling loop distribution (shown in 8(b)). Next, the loop unswitching rule is applied and the conditional construct is moved out of the `for` loop (shown in 8(c)). Next, the loop distribution rule is applied (shown in 8(d)). Note that, due to the cyclic dependency between `S2` and `S3` the loop cannot be further distributed. After the application of the loop interchange rule (shown in Figure 8(e)), there is no more scope for fine-grain distillation and we proceed to apply coarse-grain distillation.

First, the loop unpeeling rule is applied (shown in Figure 8(f)). After that, the loop interchange rule is applied again (shown in Figure 8(g)), and 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 8(h)); this is done in the context of cleanup optimizations. Comparing the the original code (in Figure 8(a)) and the final code (in Figure 8(h)), it can be easily observed that `forall` distillation is not a straightforward

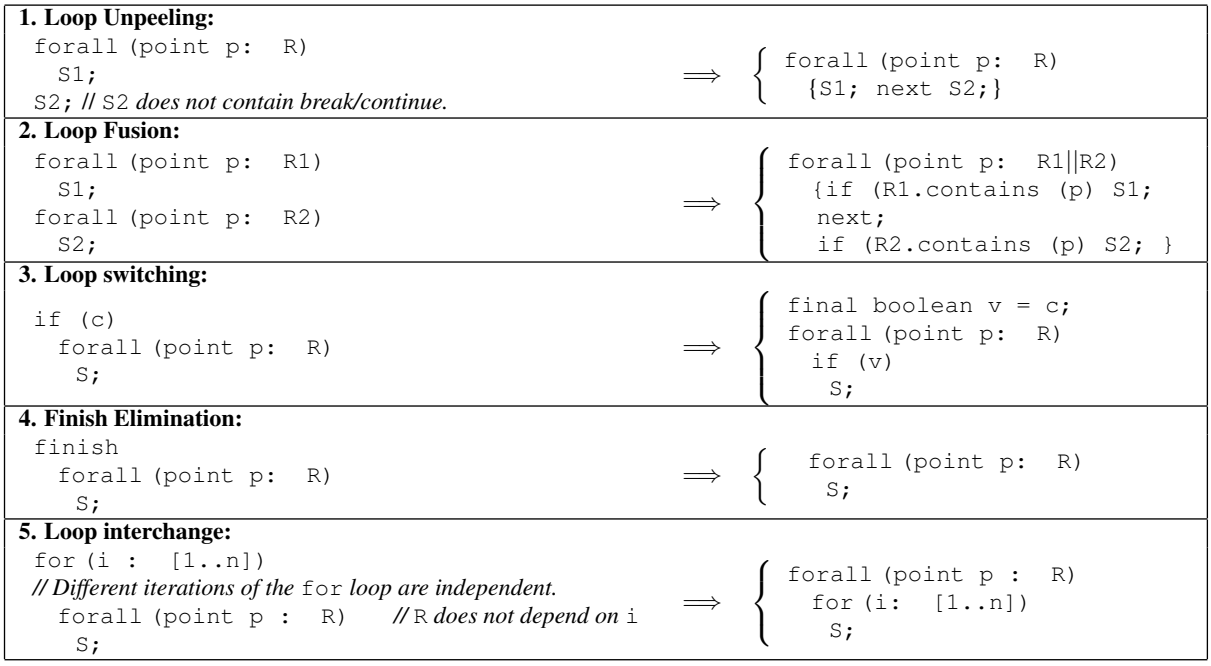

Figure 6: Coarse-grain `forall` distillation: rules for loop unpeeling, loop fusion, loop switching, finish elimination, and loop interchange.

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

Figure 10: Semantically equivalent translation of the code shown in Figure 2.

transformation in general. Likewise, Figure 10 shows the correct transformation for the code snippet shown in Figure 2.

4. EXCEPTIONS AND DISTILLATION

In this section, we discuss the impact of exception semantics on the `finish`-distillation techniques discussed in Section 3. 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 per the exception semantics discussed in Section 2, an uncaught exception thrown inside an iteration of a `forall` loop does not terminate the other parallel iterations of the loop. These exceptions are only caught by the surrounding implicit `finish`, after all the activities forked in the `forall` have terminated. This `finish` bundles all the caught exceptions into a `MultiException` and throws it again. Thus, in all the transformations described in Section 3 that involve increasing the scope of a `forall` statement the exception semantics have to be maintained explicitly.

We follow the same overall approach as shown in Figure 4 even in the presence of exceptions. Figure 11 presents the rules to handle exceptions, and are briefly discussed below. Besides presenting a new rule (loop switching (try-catch)), we modify the existing rules for some of the transformations. As we can see, the rules have now become more complicated than the ones in Figure 5 and Figure 6 , thereby underscoring the value of performing these transformations automatically with a compiler rather than depending on programmers to implement these transformations by hand.

The *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 *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 evaluates `S2` in a try-catch block, and saves any thrown exception in `ex`; this variable is checked outside the `forall` loop and if it is set, then it is thrown upwards. 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)* rule generates code to execute each iteration of `S1` inside a try-catch block, and saves the thrown exception in a `MultiException` data structure. After the `forall` loop has terminated, we check if any exception was thrown, and invoke `S2` accordingly.

```
// Original Example Code
THREADS = [0:num_threads-1];
for (int itt=1; itt<=niter; itt++) {
  S1;
  if (serial) {
    forall (point [p]: THREADS) S2;
    S3;
    // Say there is cyclic dependency
    // between S2 and S3
    forall (point [p]: THREADS) S4; } }
                (a)

// After Loop Unswitching
THREADS = [0:num_threads-1];
for (int itt=1; itt<=niter; itt++) S1;
 if (serial) {
   for (int itt=1; itt<=niter; itt++) {
    forall (point [p]: THREADS) S2;
    S3;
    forall (point [p]: THREADS) S4; } }
                (c)

// After Loop Interchange
THREADS = [0:num_threads-1];
for (int itt=1; itt<=niter; itt++) S1;
if (serial) {
  for (int itt=1; itt<=niter; itt++) {
    forall (point [p]: THREADS) S2;
    S3; }
   forall (point [p]: THREADS)
     for (int itt=1; itt<=niter; itt++) S4; }
                (e)

// After Loop Interchange
THREADS = [0:num_threads-1];
for (int itt=1; itt<=niter; itt++) S1;
if (serial) {
   forall (point [p]: THREADS)
     for (int itt=1; itt<=niter; itt++) {
        S2;
        next S3; }
  forall (point [p]: THREADS)
   for (int itt=1;itt<=niter;itt++) S4; }
                (g)

// After Loop Distribution
THREADS = [0:num_threads-1];
 for (int itt=1; itt<=niter; itt++)
    S1;
 for (int itt=1; itt<=niter; itt++) {
   if (serial) {
     forall (point [p]: THREADS) S2;
     S3;
     forall (point [p]: THREADS) S4; } }
                (b)

// After Loop Distribution
THREADS = [0:num_threads-1];
for (int itt=1; itt<=niter; itt++) S1;
if (serial) {
   for (int itt=1; itt<=niter; itt++) {
     forall (point [p]: THREADS) S2;
     S3; }
   for (int itt=1; itt<=niter; itt++)
     forall (point [p]: THREADS) S4; }
                (d)

// After Loop Unpeeling
THREADS = [0:num_threads-1];
for (int itt=1; itt<=niter; itt++) S1;
if (serial) {
  for (int itt=1; itt<=niter; itt++)
     forall (point [p]: THREADS) {
          S2;
          next S3; }
  forall (point [p]: THREADS)
   for (int itt=1;itt<=niter;itt++) S4; }
                (f)

// After Loop Fusion
THREADS = [0:num_threads-1];
for (int itt=1; itt<=niter; itt++) S1;
if (serial) {
   forall (point [p]: THREADS) {
     for (int itt=1; itt<=niter; itt++){
         S2;
         next S3; }
     for(int itt=1;itt<=niter;itt++) S4; } }
                (h)
```

Figure 8: Applying the `forall` distillation described in Figure 4. (a) the input program, (b) fine-grain distillation: loop distribution, (c) fine-grain distillation: loop unswitching, (d) fine-grain distillation: loop distribution, (e) fine-grain distillation: loop interchange, (f) coarse-grain distillation: loop Unpeeling, (g) coarse-grain distillation: loop interchange. (h) cleanup optimization: loop fusion. The changes are shown in bold.

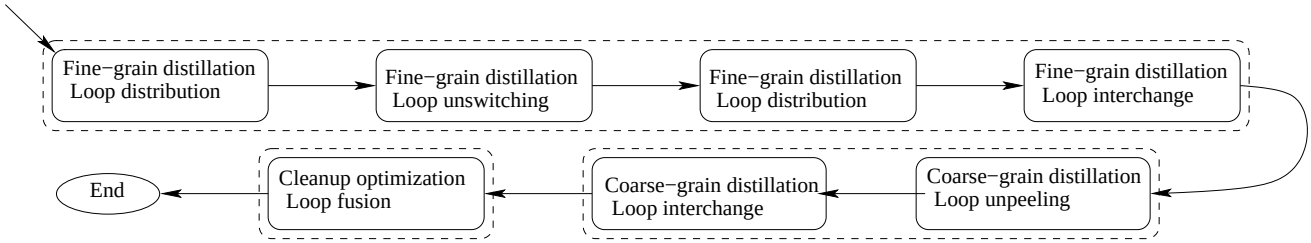

Figure 9: A flow chart describing the different invoked transformations for the example program 8(a)

Loop Distribution:

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

$\Longrightarrow$

```
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;
```

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;
```

$\Longrightarrow$

```
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;
```

$\Longrightarrow$

```
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;
```

$\Longrightarrow$

```
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; }
```

$\Longrightarrow$

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

Figure 11: `forall` **distillation in the presence of exceptions.**

	Data Size	Loop Fusion	Loop Unpeeling	Loop Interchange	Loop Unswitching	Finish Elimination	Loop Distribution
MG	A	8	16	2	4	1	0
CG	A	4	6	2	1	0	0
SOR	C	0	1	2	0	0	0
LUFact	C	0	2	1	1	0	0
Moldyn	B	6	4	0	0	0	0

Table 1: The number of different transformations applied to each benchmark.

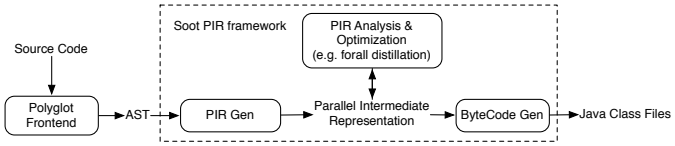

Figure 12: Habanero-Java Compiler Framework

5. IMPLEMENTATION

In this section we briefly discuss some of the implementation details of our `forall` distillation in the context of the Habanero-Java Compiler framework (HJC) [12] which translates Habanero-Java (HJ) (see Section 2) source code to Java bytecode. Figure 12 presents the basic structure of the HJC compiler. The Polyglot-based front-end for HJ was modified to emit a new Parallel Intermediate Representation (PIR) extension to the Jimple intermediate representation used in the SOOT bytecode analysis and transformation framework [22]. In addition to standard Java operators, the PIR includes explicit constructs for parallel operations such as `async`, `finish`, and `forall`.

The transformations described in Section 3 and Section 4 are implemented in HJC as an additional optimization pass over the PIR. The HJC compiler builds a Region Structure Tree (RST) [24] before invoking the transformation process, where a *region* represents an HJ parallel construct (e.g., `async`, `finish`, `forall`), or a sequential Java loop constructs (e.g., `for`, `while`), or a method call. Method-call regions are used to enable inter-procedural analysis in the HJC compiler. Our optimization framework processes the RST nodes in a post-order traversal, starting from the main method of the HJ program. During different transformations the RST may have to be rebuild to keep it consistent with the transformed program. Instead of rebuilding the whole RST after each transformation, we rebuild the RST only when the transformation requires a change in the RST, and further we rebuild only part of the RST. Thus, we rebuild the RST starting at the new `forall` node after each invocation of loop-fusion, finish-elimination, and loop interchange, and starting at the two `for` loops after loop-distribution. After all the transformations the generated PIR is translated to Java bytecode. For the results reported in this paper, all transformed Java class files were executed using the HJ work-sharing runtime based on the ThreadPoolExecutor utility [3].

6. EMPIRICAL EVALUATION

In this section, we present experimental results for the optimizations described in this paper using the Habanero-Java compiler and runtime system [12].

6.1 Experimental Setup

We discuss results obtained for two NAS Parallel [8] benchmarks (CG and MG) and three Java Grande Forum (JGF) [14] benchmarks (SOR, LUFact and MolDyn). The experimental results were obtained with the following variants to evaluate the impact of `forall` distillation and the other cleanup optimizations.

1. Java serial: Original serial Java version form the benchmark site. This version is used as the baseline for all speedup results.

2. Unopt: Medium-grained parallel HJ version using the `finish`, `async`, and `forall` constructs, with none of the distillation optimizations described in this paper. As described in [9], this corresponds to a high productivity variant for single place execution, and derives easily from the sequential version of the benchmark.

3. Opt: Code generated by applying the `forall` distillation transformation described in Section 3.

4. Opt + RNSE: Code generated by the compiler by both applying `forall` distillation and redundant next/nextsingle elimination (RNSE).

5. Java manual: Hand-coded parallel Java versions of the benchmarks obtained from the benchmark web sites (used to calibrate our compiler optimizations).

The data size used for each benchmark is shown in the second column of Table 1; we used the largest input size for all the benchmarks except CG and MG for which the No-Distillation versions could not complete execution for the larger sizes. All unoptimized and optimized variants were compiled by enabling loop chunking of `forall` loops with a block scheduling policy [20].

Table 1 presents a report on the number of different transformations applied to each of the benchmarks. It can be seen that on all the benchmarks the loop fusion and loop interchange (the transformations that actually lead to reduction in the activities and barriers) transformations were invoked at least once. However, loop distribution was never performed in these benchmarks, as the preconditions on this transformation was not satisfied. However, as we show in Figure 8, there are other examples where loop distribution may be performed.

We used three platforms for our experimental evaluation: (a) a 128-way (dual-socket, 8 cores $\times$ 8 hardware threads/core per socket) 1.2 GHz UltraSPARC T2 (Niagara 2) with 32 GB main memory running Solaris 10 and Sun JDK 1.5 32-bit version; (b) 16-way (quad-socket, quad-core per socket) Intel Xeon 2.4GHz system with 30GB of memory and running Red Hat Linux (RHEL 5) and Sun JDK 1.6 64-bit version; and (c) 32 way (quad chip, 8 cores per chip) 3.55GHz Power7 with 256 GB main memory running Red Hat Linux (RHEL 5.4) and IBM JDK 1.6 64-bit version. 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 in [11]. 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.

6.2 Experimental Results

For all the benchmarks shown in Table 1, we measured the speedup relative to the serial version. Figure 13 shows the results obtained on Niagara T2, Intel Xeon, and IBM Power7 in three columns; we plot the speedup (relative to the serial execution) for varying number of worker threads (cores/hardware threads). On Power7, we could not get reliable numbers for MG; we are exploring the possible causes. An shown in the rest of the charts, `forall` distillation leads to significant improvements and it scales well with the increase in the number of processors.

The amount of gains in the Distillation version (compared to the No-Distillation 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.

Loop chunking [20] is a part of all the parallel HJ variants. As it can be seen, `forall` distillation impacts the performance due to chunking in a positive way.

We also measured the impact of `Redundant Next/Next-Single Elimination` (RNSE) phase to eliminate the redundant barrier operations (see Section 3.3). In benchmarks like JGF.SOR-C and NPB.MG-A we see the visible benefits.

Compared to the hand optimized Java version, the optimized HJ version performs better on Niagara and Power7, but not so well on Xeon: We derive performance benefits mainly from reduction in activities and reduction in barriers. We found that, compared to

Niagara and Power7 the cost of barrier operations on the Xeon system was minimal, and the Java version where the code is in SPMD form does not incur much overhead in terms of additional activities. That's the main reason for the performance difference. Further, we see that for higher number of hardware threads ($\geq$ 64), the Java version does not scale as well as the HJ version. We attribute it to the presence of reduced number of tasks and barriers in the optimized HJ code.

Modifications to Moldyn : the original hand optimized Java version of Moldyn compared quite poorly to the optimized HJ version. We found the reasons to be related to the data boundary issues; we padded the arrays therein (to cache line boundaries) to derive the improved results presented in this paper.

It can be seen on the Niagara machine MG, SOR, and LUFact show a drop in performance for more than 64 threads. This is attributed to the fact that the platform contains two UltraSPARC T2 chips and the communication across these two chips is more expensive than that within a chip. Similarly, on the Xeon system these benchmarks show a drop in performance for more than 8 threads

It can also be seen that for CG and LUFact, the No-Distillation version does not scale well for more than 32 threads. Compared to that the Distillation versions did not show scalability issues. This is because of the reduced task creation, termination, synchronization and scheduling overheads arising out of our transformations.

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 it 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).

The geometric mean of speedup ratio on the three systems (utilizing the maximum number of threads or cores) is shown below:

System	threads/cores	Unopt	Opt	Opt+RNSE
Niagara	128	10.7×	45.8×	48.5×
Xeon	16	3.44×	6.98×	7.27×
Power7	32	2.69×	16.4×	17.2×

6.3 Distillation and Data Locality

The improvements shown in Figure 13 result from two factors: (a) direct improvements: reduced task creation, termination, synchronization and scheduling overheads, and (b) indirect improvements: some of the transformations like loop interchange and loop fusion may improve locality. We now present a study to understand the contribution of these factors in the improvements cited in Section 6.2. To understand the impact of these two underlying factors, we conducted a simple experiment: for each of the benchmarks presented in Table 1, we compared the following three versions:

- Unopt: parallel version of the benchmark with no distillation.
- Opt: manually apply the `forall` distillation.
- 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 1(b) is generated by adding the following compensation code:

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

Table 2 presents the execution time numbers for each of the three versions of the benchmarks shown in Table 1.

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 schedule 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.

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

Table 2: Execution time (in seconds) numbers to identify the impact of locality

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

7. RELATED WORK

There has been a lot of past work on reducing thread creation and synchronization overheads. These include SPMDization [6, 2, 21, 4], synchronization optimizations [7], and barrier elimination [21]. Cytron et al. [19] present an approach for transforming code written in fork-join style to SPMD code. Tseng [21] follows up on Cytron et al. in 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` distillation has similarities to the traditional SPMDization techniques. Some of the rules like loop fusion II, and loop interchange described in Section 3.2 are similar to the translation scheme suggested by Tseng [21]. There are three main differences though: (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 not necessarily an SPMD program. (c) We handle programs with exceptions and perform further cleanup optimizations to gain performance.

A recent work on applying SPMDization to task-parallel languages is by Bikshandi et al. [4], 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 compiler backend. Further, we handle programs with arbitrary `async` operations, `forall` loops, and exceptions

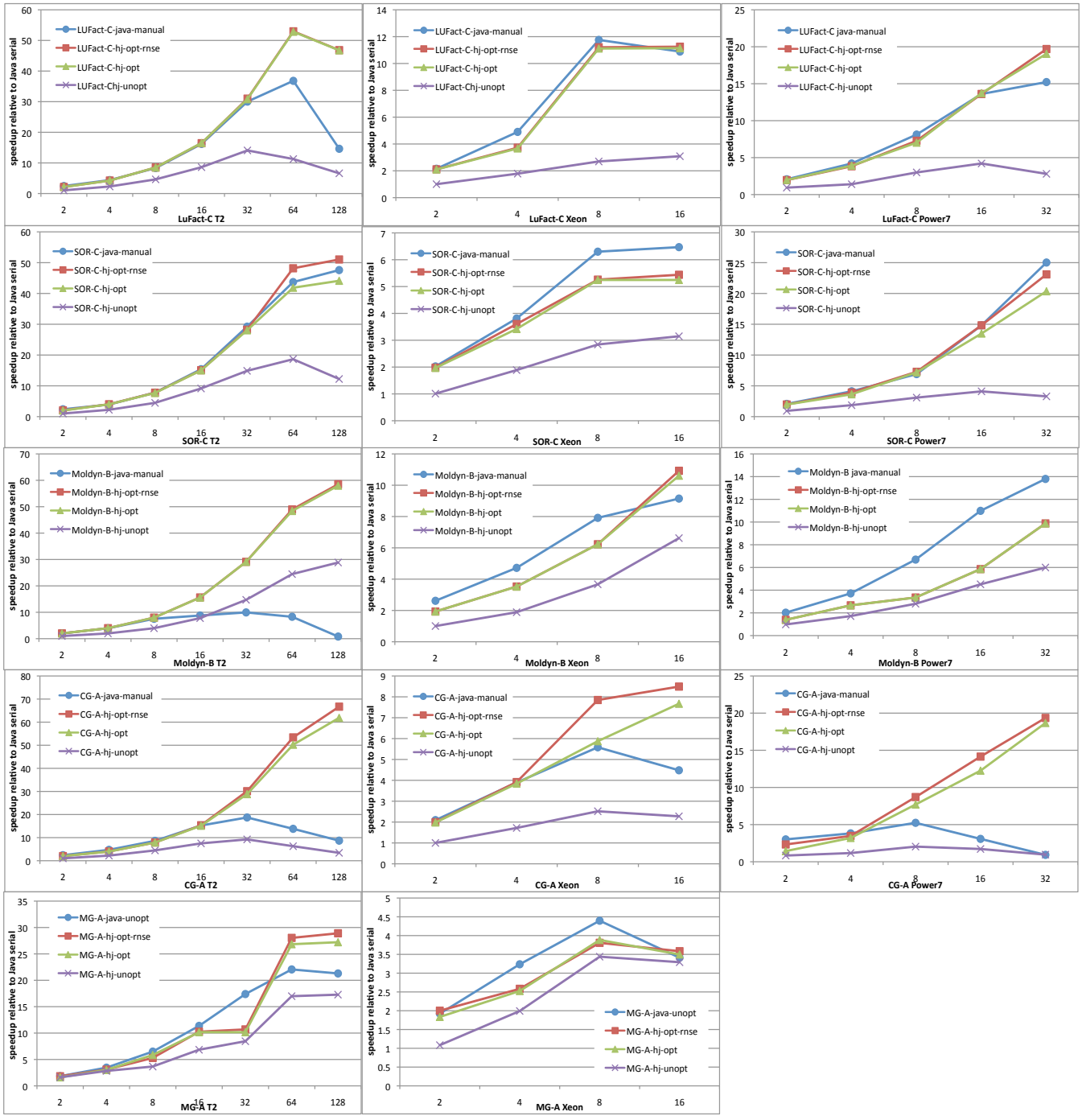

Figure 13: Distillation results from Niagara T2, Intel Xeon SMP and Power7 SMP

Nicolau et al. [18] present an approach to optimize point-to-point synchronization by eliminating redundant `wait` operations. Their approach has similarities only to our post-optimization pass, where we eliminate some redundant barriers.

Ferrero et al. [10] 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` distillation phase can be invoked as a postpass to their phase to further increase the gains.

Shirako et al. [20] present a scheme to reduce the number of dynamic activities and barriers by chunking of parallel loops. We show that our `forall` distillation framework can be deployed along with the loop chunking phase to realize further gains.

8. CONCLUSION

In this paper, we presented the compiler transformation techniques for eliminating the redundant task creation/termination and synchronization operations in task-parallel languages. We presented a systematic method that extends past classical compiler transformation techniques to automatically translate the task-parallel code into distilled code with coarser-grain tasks in a safe way. These transformations resulted in reduced task creation, termination, synchronization and scheduling overheads, thereby improving performance and scalability. Our experimental results for 5 benchmark programs on an UltraSPARC II multicore processor, Intel Xeon SMP and Power7 SMP showed 4.6×, 2.1× and 6.4× performance improvements respectively in geometric mean related to non-optimized parallel codes. This wide gap underscores the importance of using these techniques in future compiler and runtime systems for programming models with lightweight parallelism.

Acknowledgments

We would like to thank members of the Habanero group at Rice and the X10 team at IBM for valuable discussions related to this work. We gratefully acknowledge support from 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 Expedition in Computing Award CCF-0926127. Finally, we would like to thank the anonymous reviewers for their comments and suggestions, and Doug Lea for providing access to the UltraSPARC T2 SMP system used to obtain the performance results reported in this paper.

9. REFERENCES

[1] E. Allan et al. The Fortress language specification version 0.618. Technical report, Sun Microsystems, April 2005.

[2] S. P. Amarasinghe and M. S. Lam. Communication Optimization and Code Generation for Distributed Memory Machines. In *Proceedings of the conference on Programming language design and implementation*, pages 126–138. ACM, 1993.

[3] R. Barik et al. Experiences with an SMP implementation for X10 based on the Java concurrency utilities. In *Proceedings of the Workshop on Programming Models for Ubiqui- tous Parallelism, Seattle, Washington*, 2006.

[4] G. Bikshandi et al. Efficient, portable implementation of asynchronous multi-place programs. In *Proceedings of the symposium on Principles and practice of parallel programming*, 2009.

[5] R. D. Blumofe et al. CILK: An efficient multithreaded runtime system. *Proceedings of Symposium on Principles and Practice of Parallel Programming*, pages 207–216, 1995.

[6] R. Cytron, J. Lipkis, and E. Schonberg. A Compiler-Assisted Approach to SPMD Execution. *Supercomputing*, Nov 1990.

[7] P. C. Diniz and M. C. Rinard. Synchronization transformations for parallel computing. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 187–200. ACM, 1997.

[8] D. H. Bailey et al. The nas parallel benchmarks, 1994.

[9] P. Charles et al. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the conference on Object oriented programming, systems, languages, and applications*, pages 519–538, 2005.

[10] R. Ferrer, A. Duran, X. Martorell, and E. Ayguadé. Unrolling Loops Containing Task Parallelism. In *Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing*, Sep 2009.

[11] Andy Georges et al. Statistically Rigorous Java Performance Evaluation. *SIGPLAN Not.*, 42(10):57–76, 2007.

[12] Habanero Java. http://habanero.rice.edu/hj, Dec 2009.

[13] Cray Inc. The Chapel language specification version 0.4. Technical report, Cray Inc., February 2005.

[14] The Java Grande Forum benchmark suite. http://www.epcc.ed.ac.uk/javagrande/javag.html.

[15] J.Shirako, D.M.Peixotto, V.Sarkar, and W.N.Scherer III. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 277–288, New York, USA, 2008. ACM.

[16] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[17] Calvin Lin and Lawrence Snyder. *Principles of Parallel Programming*. Addison-Wesley, 2008.

[18] A. Nicolau, G. Li, A.V. Veidenbaum, and A. Kejariwal. Synchronization optimizations for efficient execution on multi-cores. In *Proceedings of the 23rd international conference on Supercomputing*, pages 169–180, New York, NY, USA, 2009. ACM.

[19] R.J.Cytron, J.T.Lipkis, and E.T.Schonberg. A compiler-assisted approach to SPMD execution. In *Proceedings of Supercomputing*, 1990.

[20] Jun Shirako, Jisheng Zhao, V. Krishna Nandivada, and Vivek Sarkar. Chunking parallel loops in the presence of synchronization. In *ICS*, pages 181–192, 2009.

[21] C. Tseng. Compiler optimizations for eliminating barrier synchronization. In *Proceedings of the symposium on Principles and practice of parallel programming*, pages 144–155, New York, NY, USA, 1995. ACM.

[22] R. Vallée-Rai et al. Soot - a Java Optimization Framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[23] K. Yelick et al. Productivity and performance using partitioned global address space languages. In *Proceedings of the international workshop on Parallel symbolic computation*, pages 24–32, New York, USA, 2007. ACM.

[24] J. Zhao and V. Sarkar. A hierarchical region-based static single assignment form. Technical Report TR09-9, Department of Computer Science, Rice University, December 2009.