Chunking Parallel Loops in the Presence of Synchronization

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

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

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

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

ABSTRACT

Modern languages for shared-memory parallelism are moving from a bulk-synchronous Single Program Multiple Data (SPMD) execution model to lightweight Task Parallel execution models for improved productivity. This shift is intended to encourage programmers to express the ideal parallelism in an application at a fine granularity that is natural for the underlying domain, while delegating to the compiler and runtime system the job of extracting coarser-grained useful parallelism for a given target system. A simple and important example of this separation of concerns between ideal and useful parallelism can be found in chunking of parallel loops, where the programmer expresses ideal parallelism by declaring all iterations of a loop to be parallel and the implementation exploits useful parallelism by executing iterations of the loop in sequential chunks.

Though chunking of parallel loops has been used as a standard transformation for several years, it poses some interesting challenges when the parallel loop may directly or indirectly (via procedure calls) perform synchronization operations such as barrier, signal or wait statements. In such cases, a straightforward transformation that attempts to execute a chunk of loops in sequence in a single thread may violate the semantics of the original parallel program. In this paper, we address the problem of chunking parallel loops that may contain synchronization operations. We present a transformation framework that uses a combination of transformations from past work (e.g., loop strip-mining, interchange, distribution, unswitching) to obtain an equivalent set of parallel loops that chunk together statements from multiple iterations while preserving the semantics of the original parallel program. These transformations result in reduced synchronization and scheduling overheads, thereby improving performance and scalability. Our experimental results for 11 benchmark programs on an UltraSPARC II multicore processor showed a geometric mean speedup of 0.52× for the unchunked case and 9.59× for automatic chunking using the techniques described in this paper. This wide gap underscores the importance of using these techniques in future compiler and runtime systems for programming models with lightweight parallelism.

Categories and Subject Descriptors

D.1.3 [Programming Techniques]: Concurrent Programming; D.3.4 [Programming Languages]: Processors

General Terms

Algorithms, Languages, Performance

1. INTRODUCTION

Historically, the most successful runtimes for shared memory multiprocessors have been based on bulk-synchronous Single Program Multiple Data (SPMD) execution models [6]. OpenMP [17] represents one such embodiment in which the programmer’s view of the runtime is that of a fixed number of threads executing tasks in “work-sharing” parallel constructs. However, modern languages such as Cilk [4], Chapel [13], Fortress [1], and X10 [5] have moved from SPMD to lightweight dynamic Task Parallel execution models for improved programmer productivity. This shift is intended to encourage programmers to express the ideal parallelism in an application at a fine granularity that is natural for the underlying domain, while delegating to the compiler and runtime system the job of extracting coarser-grained useful parallelism for a given target system. A simple and important example of this separation of concerns between ideal and useful parallelism can be found in chunking of parallel loops, where the programmer expresses ideal parallelism by declaring all iterations of a loop to be parallel and the implementation exploits useful parallelism by executing iterations of the loop in sequential chunks [15, 18]. In models like OpenMP, the programmer can guide the implementation by providing chunk policy and chunk size values that can be set dynamically for different platforms.

Though chunking of parallel loops has been employed as a standard transformation for several years, it poses some interesting challenges when the parallel loop may directly or indirectly (via procedure calls) perform synchronization operations such as barrier, signal or wait statements. In such cases, a straightforward transformation that attempts to execute a chunk of loops in sequence in a single thread may violate the semantics of the original parallel program. In this paper, we address the problem of chunking parallel loops that may contain synchronization operations. We present a transformation framework that uses a combination of transformations from past work (e.g., loop strip-mining, interchange, distribution, unswitching) to obtain an equivalent set of parallel loops that chunk together statements from multiple iterations while preserving the semantics of the original parallel program. These transformations result in reduced synchronization and scheduling overheads, thereby improving performance and scalability. Our experimental results for 11 benchmark programs on an UltraSPARC II multicore processor showed a geometric mean speedup of 0.52× for the unchunked case and 9.59× for automatic chunking using the techniques described in this paper. This wide gap underscores the importance of using these techniques in future compiler and runtime systems for programming models with lightweight parallelism.

Categories and Subject Descriptors

D.1.3 [Programming Techniques]: Concurrent Programming; D.3.4 [Programming Languages]: Processors

General Terms

Algorithms, Languages, Performance

1. INTRODUCTION

Historically, the most successful runtimes for shared memory multiprocessors have been based on bulk-synchronous Single Program Multiple Data (SPMD) execution models [6]. OpenMP [17] represents one such embodiment in which the programmer’s view of the runtime is that of a fixed number of threads executing tasks in “work-sharing” parallel constructs. However, modern languages such as Cilk [4], Chapel [13], Fortress [1], and X10 [5] have moved from SPMD to lightweight dynamic Task Parallel execution models for improved programmer productivity. This shift is intended to encourage programmers to express the ideal parallelism in an application at a fine granularity that is natural for the underlying domain, while delegating to the compiler and runtime system the job of extracting coarser-grained useful parallelism for a given target system. A simple and important example of this separation of concerns between ideal and useful parallelism can be found in chunking of parallel loops, where the programmer expresses ideal parallelism by declaring all iterations of a loop to be parallel and the implementation exploits useful parallelism by executing iterations of the loop in sequential chunks [15, 18]. In models like OpenMP, the programmer can guide the implementation by providing chunk policy and chunk size values that can be set dynamically for different platforms.

Though chunking of parallel loops has been employed as a standard transformation for several years, it poses some interesting challenges when the parallel loop may directly or indirectly (via procedure calls) perform synchronization operations such as barrier, signal or wait statements. In such cases, a straightforward transformation that attempts to execute a chunk of loops in sequence in a single thread may violate the semantics of the original parallel program. In this paper, we address the problem of chunking parallel loops that may contain synchronization operations. We present a transformation framework that uses a combination of transformations from past work (e.g., loop strip-mining, interchange, distribution, unswitching) to obtain an equivalent set of parallel loops that chunk together statements from multiple iterations while preserving the semantics of the original parallel program. These transformations result in reduced synchronization and scheduling overheads, thereby improving performance and scalability. Our experimental results for 11 benchmark programs on an UltraSPARC II multicore processor showed a geometric mean speedup of 0.52× for the unchunked case and 9.59× for automatic chunking using the techniques described in this paper. This wide gap underscores the importance of using these techniques in future compiler and runtime systems for programming models with lightweight parallelism.

Categories and Subject Descriptors

D.1.3 [Programming Techniques]: Concurrent Programming; D.3.4 [Programming Languages]: Processors

General Terms

Algorithms, Languages, Performance

1. INTRODUCTION

Historically, the most successful runtimes for shared memory multiprocessors have been based on bulk-synchronous Single Program Multiple Data (SPMD) execution models [6]. OpenMP [17] represents one such embodiment in which the programmer’s view of the runtime is that of a fixed number of threads executing tasks in “work-sharing” parallel constructs. However, modern languages such as Cilk [4], Chapel [13], Fortress [1], and X10 [5] have moved from SPMD to lightweight dynamic Task Parallel execution models for improved programmer productivity. This shift is intended to encourage programmers to express the ideal parallelism in an application at a fine granularity that is natural for the underlying domain, while delegating to the compiler and runtime system the job of extracting coarser-grained useful parallelism for a given target system. A simple and important example of this separation of concerns between ideal and useful parallelism can be found in chunking of parallel loops, where the programmer expresses ideal parallelism by declaring all iterations of a loop to be parallel and the implementation exploits useful parallelism by executing iterations of the loop in sequential chunks [15, 18]. In models like OpenMP, the programmer can guide the implementation by providing chunk policy and chunk size values that can be set dynamically for different platforms.

Though chunking of parallel loops has been employed as a standard transformation for several years, it poses some interesting challenges when the parallel loop may directly or indirectly (via procedure calls) perform synchronization operations such as barrier, signal or wait statements. In such cases, a straightforward transformation that attempts to execute a chunk of loops in sequence in a single thread may violate the semantics of the original parallel program. In this paper, we address the problem of chunking parallel loops that may contain synchronization operations. We present a transformation framework that uses a combination of transformations from past work (e.g., loop strip-mining, interchange, distribution, unswitching) to obtain an equivalent set of parallel loops that chunk together statements from multiple iterations while preserving the semantics of the original parallel program. These transformations result in reduced synchronization and scheduling overheads, thereby improving performance and scalability. Our experimental results for 11 benchmark programs on an UltraSPARC II multicore processor showed a geometric mean speedup of 0.52× for the unchunked case and 9.59× for automatic chunking using the techniques described in this paper. This wide gap underscores the importance of using these techniques in future compiler and runtime systems for programming models with lightweight parallelism.
grained dynamic parallelism. We show the X10 version on page 54.

Modern languages such as Chapel [13], Fortress [1], iters

operations are inserted to ensure that di

to create a parallel code in Figure 1 is clear enough. The programmer wishes

code in Figure 1 yields unpredictable results on di

diff

of SPMD computations. Attempting to run the OpenMP

a barrier region from being nested inside a loop region. This

with the OpenMP specification because OpenMP prohibits

ray

is to perform iterative averaging on an one-dimensional ar-

Figure 1: One-Dimensional Iterative Averaging Ex-

delta = epsilon+1; iters = 0;
#pragma omp parallel for
for (int j = 1 ; j <= n ; j++ ) {
    while ( delta > epsilon ) {
        #pragma omp barrier
        if (j == 1) {
            delta = sum(diff); iters++;
            temp = newA; newA = oldA; oldA = temp;
        }
        #pragma omp barrier
    }
}

Figure 2: One-Dimensional Iterative Averaging Ex-

delta = epsilon+1; iters = 0;
phaser ph = new phaser(single);
foreach ( point[j] : [1:n:S] ) phased(single(ph)) {
    while ( delta > epsilon ) {
        next { // barrier with single statement
            delta = diff.sum(); iters++;
            temp = newA; newA = oldA; oldA = temp;
        }
    }
}

Figure 3: Naive (Incorrect) Chunking of X10 version
from Figure 2

delta = epsilon+1; iters = 0;
phaser ph = new phaser(single);
foreach ( point[jj] : [1:n:S] ) phased(single(ph)) {
    for (int j = jj ; j <= min(jj+S-1,n) ; j++) {
        while ( delta > epsilon ) {
            next { // barrier with single statement
                delta = diff.sum(); iters++;
                temp = newA; newA = oldA; oldA = temp;
            }
        }
    }
}

Parallel j loop is now expressed as a foreach statement in
X10. All iterations of the foreach are registered on the
same phaser variable, ph. The next statement serves as a
barrier with a single statement [25] that is guaranteed to be
executed by only one thread.

The code in Figure 2 correctly captures the programmer’s
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
is a standard approach to improve the efficiency of a paral-
lel loop. Figure 3 shows the result of performing a chunking
transformation mechanically on the foreach loop, with the
goal of decomposing the foreach loop into chunks of S it-
erations 2. 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 2 since it
contains a next (barrier) operation. In particular, the
transformed version (Figure 3) will attempt to complete all
iterations of the while loop for iteration j before starting
iteration j+1 from the same chunk, which is different from
the semantics of the original code in Figure 2. A similar
problem would arise if the original foreach loop contained
signal and wait operations instead of barrier operations.

In this paper, we address the problem of chunking para-
lel loops that may contain synchronization operations. We
present a transformation framework that uses a combination
of transformations from past work (e.g., loop strip-mining,
interchange, distribution, unswitching) to obtain an equiv-
alent set of parallel loops that chunk statements from mul-
tiple iterations while preserving the semantics of the origi-
nal program. These transformations result in reduced syn-
chronization and scheduling overheads, thereby improving
performance and scalability. Our experimental results for
11 benchmark programs on an UltraSPARC II multicore
processor showed a geometric mean speedup of 0.52× for
the unchunked case and 9.59× for automatic chunking using
the techniques described in this paper. This wide gap un-
scores the importance of using these techniques in com-
piler and runtime systems for programming models with
lightweight parallelism. A hand-coded study of different
chunking policies for two benchmarks revealed the potential
for even greater performance improvements in the future.

\footnote{While X10 is the language used to describe the problem and
our solution, the approach described in this paper is applica-
ble to any language that permits synchronization operations
to occur in a parallel loop.}

\footnote{The 1:n:S notation in the new jj foreach loop is akin to
the low : high : stride triple notation in Fortran 90 [16].}
The rest of the paper is organized as follows. Section 2 includes background on X10 and on classical loop transformations. Section 3 describes the loop chunking transformation framework. Section 4 discusses how the framework in Section 3 can be extended to support exceptions. Section 5 contains our experimental results. Section 6 discusses related work, and Section 7 contains our conclusions.

2. BACKGROUND

2.1 X10 and Phasers

This section provides a brief summary of the \texttt{async}, \texttt{finish}, and \texttt{foreach} constructs introduced in v0.41 of the X10 programming language [5], as well as the phasers extension from [20]. Additional X10 constructs such as places and futures that are not central to the paper have been omitted.

2.1.1 \texttt{async} (stmt)

Async is the X10 construct for creating or forking a new asynchronous activity. The statement, \texttt{async (stmt)}, causes the parent activity to create a new child activity to execute (stmt). Execution of the \texttt{async} statement returns immediately i.e., the parent activity can proceed immediately to its following statement.

2.1.2 \texttt{finish} (stmt)

The X10 statement, \texttt{finish (stmt)}, causes the parent activity to execute (stmt) and then wait till all sub-activities created within (stmt) have terminated (including transitively spawned activities). Operationally, each instruction executed in an X10 activity has a unique \texttt{Immediately Enclosing Finish} (IEF) dynamic statement instance.

Besides termination detection, the \texttt{finish} statement plays an important role with regard to exception semantics. An X10 activity 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. X10 requires that if statement S or an activity spawned by S terminates abruptly, and all activities spawned by S terminate, then \texttt{finish S} terminates abruptly and throws a single exception formed from the collection of all exceptions thrown by S or its descendant activities.

2.1.3 \texttt{Foreach}

The statement \texttt{foreach (point p : R) S} supports parallel iteration over all the points in region R by launching each iteration as a separate \texttt{async}. A \texttt{point} is an element of an n-dimensional Cartesian space \((n \geq 1)\) with integer-valued coordinates. A \texttt{region} is a set of points, and can be used to specify an array allocation or iteration constructs as in the case of \texttt{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 \texttt{foreach} statement does not have an implicit \texttt{finish} (join) operation, but its termination can be ensured by enclosing it within a \texttt{finish} statement at an appropriate outer level. Further, any exceptions thrown by the spawned iterations are propagated to its IEF instance.

2.1.4 Phasers

In this section, we summarize the \texttt{phaser} construct introduced in [20] as an extension to X10 clocks [5]. Phasers integrate collective and point-to-point synchronization by giving each activity (task) the option of registering with a phaser in \texttt{signal-only/wait-only} mode for producer/consumer synchronization or \texttt{signal-wait} mode for barrier synchronization. In addition, a \texttt{next} statement for phasers can optionally include a \texttt{single} statement (as in Figure 2) which is guaranteed to be executed exactly once during a phase transition [25].

These properties, along with the generality of dynamic parallelism and the phase-ordering and deadlock-freedom safety properties, distinguish phasers from synchronization constructs in past work including barriers [11, 17], counting semaphores [19], and X10’s clocks [5]. Though phasers as described in this paper 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 [17], Intel’s Thread Building Blocks, Microsoft’s Task Parallel Library, and Java Concurrency Utilities [10].

A \texttt{phaser} is a synchronization object that supports the following five operations by an activity \(A\):

- \texttt{new}: When \(A\) performs a new \texttt{phaser(MODE)} operation, it results in the creation of a new phaser \(ph\) such that \(A\) is registered with \(ph\) according to \texttt{MODE}.
- \texttt{drop}: \(A\) drops its registration on all phasers when it terminates. In addition, when \(A\) executes an end-finish instruction for \texttt{finish} statement \(F\), it completely de-registers 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 [20].
- \texttt{next}: The \texttt{next} operation has the effect of advancing each phaser on which \(A\) is registered to its next phase, thereby synchronizing all activities registered on the same phaser. The semantics of \texttt{next} depends on the registration mode that \(A\) has on each phaser, thereby making it possible for the \texttt{next} statement to be used for both barrier and point-to-point synchronizations [20].
- \texttt{signal}: A \texttt{signal} operation performed by \(A\) is shorthand for a \texttt{ph.signal()} operation performed on each phaser \(ph\) with which \(A\) is registered with a \texttt{signal} capability.
- \texttt{wait}: Like \texttt{next}, the \texttt{wait} operation has the effect of advancing each phaser that \(A\) is registered on to its next phase. However unlike \texttt{next}, the \texttt{wait} operation does not include \texttt{signal} operations on any phasers.

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 [23, 14]:

- \texttt{Strip Mining} is a loop transformation that fragments a single loop into 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.
- \texttt{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.
- \texttt{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.
3. TRANSFORMATION FRAMEWORK

In this section we present our transformation framework to enable chunking of foreach loops containing synchronization operations. To simplify the presentation, this section will focus on the restricted case when the loop body is known to be exception-free. Section 4 discusses how the framework in this section can be extended to support exceptions. The synchronization operation that we will focus on in this description is the next statement for clocks and phasers; as mentioned in Section 2.1, the phaser next statement can be used to support both barrier and point-to-point synchronizations.

Figure 4 shows a block diagram for our transformation framework. 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 clean-up transformations (the “NO” case in the flow chart). If the loop body contains next statements, then a combination of three transformations — loop distribution, loop interchange, and loop unswitching — is applied repeatedly until a) no next 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 clean-up 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 loop distribution, loop interchange, and loop unswitching as the three transformations to iterate on is to attempt to isolate the next statements by moving the inner parallel loop as far inwards as possible. These 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 Loop distribution is the basic transformation needed for chunking by isolating next operations. Interchange and unswitching increase the opportunities for isolation. Next contraction and choice of chunking policy are used to improve the efficiency of the chunked version. In this work, we assume that all programmer specified conditions guarding a next statement are invariant in the initial foreach loop i.e., the conditions are single-valued [25]. However, as we will see in Section 4, our transformation framework can handle cases when a next statement is guarded by implicit exception conditions.

Figure 5 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 5 will not be legal. The following sections describe the transformations performed by a framework that can lead to a legal chunking.

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 (iteration space) \( R \) to be an ordered pair \((Ig, Ie)\), where \( Ig(R) \) is an iterator over multiple chunks and for each chunk \( Ie(R, g) \) returns an iterator over the different indices in the chunk. In addition to the ability to specify chunks of non-contiguous iterations, this formulation allows us to specify chunking of multi-dimensional loops since regions can be multidimensional in X10. Figure 6 shows the iteration spaces for Block and Cyclic chunking policies for region \( R = [0 : N - 1] \) with \( P \) chunks.
3.2 Loop Interchange, Loop Unswitching, Loop Distribution, Next Contraction

Our serialization mechanism (described in Section 3.3) requires that no next operations appear in any i-foreach construct. In this section, we describe an iterative approach to either move all next operations out of the i-foreach 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 9. We now briefly describe each rule in Figure 9 and summarize their assumptions. If any of the assumptions is not satisfied, then the rule cannot be applied and the compiler will have to conclude that the original foreach loop cannot be chunked.

Rule 1 (Loop Interchange) builds on a well known observation from classical vectorization namely, “a loop that carries no dependences cannot carry any dependences that prevent interchange with other loops nested inside it” [14]. Though this observation was developed for sequential loops that are parallelizable, it is just as applicable to parallel i-foreach loops. Thus, the interchange in Rule 1 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 i-foreach 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 [14]. We also assume that the loop body S does not contain any break or continue statements; support for those statements is more complicated, but can be built on the exception support in Section 4.

Rule 2 (Loop Unswitching) builds on the classical unswitching transformation for sequential code [14]. The main assumption here is that the condition e is independent of the i-foreach loop’s index variable.

Rule 3 (Loop Distribution) builds on another well known observation that a parallel loop can always be fully distributed [14] since a loop-carried dependence is needed to create a distribution-preventing cycle. Hence the i-foreach loops can be fully distributed. The implicit finish operations in i-foreach ensure the correctness of the resulting transformation. As in classical loop distribution, it may be necessary in some cases to perform scalar expansion [14] on any iteration-private scalar variables that may be accessed in both S1 and S2.

<table>
<thead>
<tr>
<th>Chunking Policy</th>
<th>Iteration Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block</td>
<td>{0, 1, \ldots, N/P - 1}, {N/P, N/P + 1, \ldots, 2 \times N/P - 1}, \ldots, {((P - 1) \times N/P, \ldots N - 1}</td>
</tr>
<tr>
<td>Cyclic</td>
<td>{0, P, \ldots}, {1, P + 1, \ldots}, \ldots, {P - 1, 2 \times P - 1, \ldots}</td>
</tr>
</tbody>
</table>

Figure 6: Iteration sets for Block and Cyclic chunking policies for region \(R = [0 : N - 1]\) and \(P\) chunks.

<table>
<thead>
<tr>
<th>foreach (point p: R) phased((phaser-regs)) S</th>
<th>i-foreach (point g: Ig(R)) phased((phaser-regs))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Rightarrow)</td>
<td>i-foreach (point p: Ig(R)) phased((phaser-regs))</td>
</tr>
</tbody>
</table>

Figure 7: foreach Strip Mining Transformation Rule

```plaintext
foreach (point p: R) phased((phaser-regs))
S
⇒
foreach (point g: Ig(R)) phased((phaser-regs))
i-foreach (point p: Ig(R), g) phased S
```

Figure 8: Strip-mining of foreach loop in Figure 5.

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

```
1. **Loop Interchange:**

   i-foreach (point p : R1) phased
   for (point q : R2)
   // R2 is assumed to be independent of p
   S
   \(\Rightarrow\)
   \(\Rightarrow\)
   \(\Rightarrow\)
   i-foreach (point p : R1) phased
   for (point q : R2)
   S

   **Figure 9:** Rules for Loop Interchange, Loop Unswitching, Loop Distribution, and Next Contraction

2. **Loop Unswitching:**

   i-foreach (point p : R1) phased
   if (e)
   // e is assumed to be independent of p
   S
   \(\Rightarrow\)
   \(\Rightarrow\)
   \(\Rightarrow\)
   i-foreach (point p : R1) phased
   if (e)
   S

3. **Loop Distribution:**

   i-foreach (point p : R1) phased
   { S1; S2; }
   \(\Rightarrow\)
   \(\Rightarrow\)
   \(\Rightarrow\)
   i-foreach (point p : R1) phased
   S1;
   i-foreach (point p : R1) phased
   S2;

4. **Next Contraction:**

   i-foreach (point p : R1) phased
   // Region R1 is assumed to be non-empty.
   next
   \(\Rightarrow\)
   \(\Rightarrow\)
   \(\Rightarrow\)
   next

**Figure 10:** Applying our iterative transformation framework on the strip-mined code in Figure 8. The changes in each step are shown in bold face.
**4. EXTENSIONS FOR EXCEPTIONS**

In this section, we discuss rules to perform loop-chunking transformations in the presence of exceptions. The rules in this section are presented in the context of the 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 2.1, an uncaught exception thrown inside an *async* statement terminates the *async* but not its parent activity. The enclosing *finish* statement captures all the exceptions that are thrown inside its body, aggregates them 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 (does not impact the execution of other iterations), as each iteration of the *foreach* statement can be viewed as an independent *async* statement.

We first discuss the exception semantics of the *foreach* statement. An *foreach* statement is a temporary placeholder for a sequential *for* loop. Any exception thrown in a sequential *for* loop typically terminates the loop. However, since the *for* loop generated from the *foreach* statement is originally part of a *foreach* statement, we must execute each iteration of the *foreach* regardless of exceptions thrown in other iterations. Thus, we define the exception semantics of the *foreach* as follows: all the exceptions thrown by different iterations of the *foreach* are thrown as independent asynchronous exceptions i.e., are inserted into the MultiException collection collected at the explicit IEF (Immediately Enclosing Finish) instance for the *foreach* (ignoring implicit *finish* operations in *foreach* statements).

We follow the same overall approach as shown in Figure 4 even in the presence of exceptions. We however modify the rules for some of the transformations. Figure 13 presents the modified rules to handle exceptions, which are also briefly discussed below. As we can see, the rules have now become more complicated than the rules in Figure 9, thereby underscoring the value of performing these transformations automatically with a compiler rather than depending on programmers to implement these transformations by hand.

**Strip mining:** We re-use the strip mining rule presented in Figure 7; the exception semantics of the *foreach* statement guarantees correct translation, keeping in mind that the implicit *finish* in an *foreach* does not collect exceptions like an explicit *finish*.

**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*, but does not impact other iterations of the *foreach* loop. Thus, in the transformed program, for any iteration of the outer sequential *for* loop, the inner *foreach* should be invoked at program point *Q* only if no 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...
1. Loop interchange:

\[
\text{i-foreach } (p: \text{Ie}(R, g)) \text{ phased } \Rightarrow \begin{cases}
\text{boolean } c; \text{ Exception EX = null; try } \{s1; c = e;\} \\
\text{catch (Exception ex) } \{\text{EX} = \text{ex}; c = false;\} \\
\text{if (EX \neq \text{null}) foreach (p: \text{Ie}(R, g)) throw EX; Region newR = new Region(\text{Ie}(R, g));} \\
\text{ Exception [] exArr = new Exceptions[\text{newR.size()}];} \\
\text{for (;c;)} \{ \\
\text{for (q: newR) if (exArr[q] \neq \text{null}) newR.remove(q);} \\
\text{i-foreach (p: newR) phased } \\
\text{// might have to do renaming try } \{ \\
\text{S;} \\
\text{s2; c = false; c = e;} \\
\text{catch (Exception e) } \{\text{exArr[p] = e; }\} \\
\text{\}} \\
\text{\}} \\
\text{foreach (p: \text{Ie}(R, g)) throw exArr[p];}
\end{cases}
\]

2. Loop unswitching:

\[
i-\text{foreach } (p: \text{Ie}(R, g)) \text{ phased } \Rightarrow \begin{cases}
\text{boolean } c; \text{ Exception EX = null; try } \{c = e;\} \\
\text{catch (Exception ex) } \{\text{EX} = \text{ex}; c = false;\} \\
\text{if (EX \neq \text{null}) foreach (p: \text{Ie}(R, g)) throw EX; if (c)} \text{ i-foreach (p: \text{Ie}(R, g)) phased } S
\end{cases}
\]

3. Loop unswitching (try-catch):

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

4. Loop distribution:

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

5. Next-Contraction

\[
i-\text{foreach } (p: \text{Ie}(R, g)) \text{ phased } \Rightarrow \{ \text{next} \}
\]

Figure 13: Rules for loop interchange, unswitching, distribution, and next contraction in the presence of exceptions.
(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.

**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 exception that might be thrown in the try-block. We do so, by first unswitching and then enclosing the inner i-foreach with a finish statement. As shown in Rule 3, 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-foreach loop over all the points for which we had 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.

**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 executed S1 normally (did not throw an exception outside) and use it to iterate over S2 as shown in Rule 4.

**Next simplification:** This rule is same as the rule presented in Figure 9.

Serializaton of i-foreach statements must respect their exception semantics. We present below the rule for serialization in the presence of exceptions.

\[
\text{i-foreach (p: Ie(R, g)) phased} \rightarrow \begin{cases} \text{for (p : Ie(R, g))} \\
\text{try} \{ S \} \\
\text{catch (Exception e)} \\
\text{(async throw e)} \end{cases}
\]

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.

5. EXPERIMENTAL RESULTS

In this section, we present experimental results obtained using the compiler framework shown in Figure 14. The input programs are written in X10 (v.1.5) language [5] extended with phasers [20], but the approach is applicable to chunking of parallel loops with synchronization in other languages as well. We modified the Polyglot-based front-end for X10 [24] to emit a new Parallel Intermediate Representation (PIR) extension to the Jimple intermediate representation in the SOOT bytecode analysis and transformation framework [22]. The PIR includes explicit constructs for parallel operations such as foreach, async and finish. The transformations described in Section 3 are performed in the PIR Analysis & Optimization component, after which the PIR is translated to Java bytecodes. The transformed Java class files are executed using the X10 runtime based on the ThreadPoolExecutor utility from the java.util.concurrent library [2].

In this section, we report results for the 11 benchmarks listed in Table 1. The asterisk-ed benchmarks contain foreach loops with phaser next operations\(^\ddagger\); compiler analysis was necessary to establish the absence of next operations for the other benchmarks. All results were obtained on a 64-way (8 cores x 8 threads per core) 1.2 GHz UltraSPARC T2 (Nagara 2) with 32 GB main memory running Solaris 10, using a Java 5 Runtime Environment (build 1.5.0_12-b04) with Java HotSpot Server VM (build 1.5.0.12-b04, mixed mode) and the “-Xms1000M -Xmx1000M” options to set the heap size to 1GB. For all 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 time in the performance results, in accordance with the methodology reported in [9]. For the X10 runtime options, -NUMBER_OF_LOCAL_PLACES was set to 1 and -INIT_THREADS_PER_PLACE was set equal to the number of worker threads for which the measurement was being performed.

5.1 Performance Results for Automatic Chunking of Foreach Loops

In this section, we present results on a 64-way Sun UltraSPARC T2 server for the following variants of each benchmark:

- **Serial.** Sequential Java version without any parallel con-
5.2 Hand-coded Comparison of Different Chunking Policies

The previous section presented performance results for automatic chunking of foreach loops using a fixed static chunking policy. It is well known that different chunking policies may be best for different parallel loops depending on the load imbalance and locality across different iterations. In this section, we use hand-coded transformations to explore the impact of chunking policy on two benchmarks, LUFact and MolDyn. In future work, we plan to obtain these measurements automatically by allowing the programmer to annotate a foreach loop with a desired chunking policy (as in OpenMP), and by also investigating automatic selection of chunking policies in the compiler.

These hand-coded performance results include some additional transformations that we expect to include in a future version of our compiler. Loop-invariant-code-motion (LICM) moves the loop invariant code out of the foreach loop. Given the example in Figure 10(c), if all of the operands...
4. *H* are assigned to the
*in Dyn with Hand-Optimized Chunking Policies for 64
Figure 18: Speedup for JGF LUFact and Mol-
block size for each chunk is
SPARC II. As we saw earlier, the versions without foreach
keeps a history of executed chunks and accessed data, and
ment techniques in our implementation where each activity
sharing parallel loops in an SPMD execution model except

2. Block version includes the additional hand-coded optimiza-
processor) as in the Chunk case in the previous section.

1. Statically divide the
processors (one chunk of contiguous iterations per
The main difference between Chunk and Block is that the
Block version includes the additional hand-coded optimiza-
ations listed above.

2. Cyclic. Perform a cyclic partitioning of the iterations
into *P* chunks of non-contiguous iterations, so that each
chunk executes iterations that are *P* apart.

3. Block-cyclic. Divide the iteration space into *H* × *P* con-
tiguous chunks, where *H* is the number of “hops” and the
block size for each chunk is *N*/(*H* × *P*). These *H* × *P* chunks
are assigned to the *P* processors in a cyclic manner. We used
*H* = 4 for the results in this section.

4. Dynamic. Create one activity per processor at the outer
level, but enable the activities to dynamically share chunks of
parallel loop iterations, with chunk size *N*/(*H* × *P*) and
*H* = 4 as in the Block-cyclic case. This is analogous to work-
sharing parallel loops in an SPMD execution model except that
the Dynamic policy is augmented with locality improvement
techniques in our implementation where each activity
keeps a history of executed chunks and accessed data, and
uses it to take the next chunk.

Figure 18 shows the speedup for the JGF LUFact and MolDyn benchmarks when using all 64 threads on an UltraSPARC II. As we saw earlier, the versions without foreach
chunking are slower than serial execution for LUFact and MolDyn (even with the hand-coded optimizations). Further,
the NoChunk version did not complete for Size B for MolDyn
due to OutOfMemory and other runtime errors resulting
from the creation of too many activities. On the other hand,
all the chunked versions show good speedup. The speedups
relative to the serial Java version for LUFact with Size C
(the largest size for LUFact) are 30.4× for Block, 39.4× for
Cyclic, 34.9× for Block-Cyclic, and 35.9× for Dynamic. It
is not surprising that Cyclic yields the best performance for
LUFact, since the work in each parallel iteration is embodied
in a triangular sequential loop. However, the other policies
deliver reasonable performance as well.

For MolDyn, the speedups relative to the serial Java
version for Size B (the largest size for MolDyn) are 27.9× for
Block, 40.0× for Cyclic, 39.4× for Block-Cyclic, and 40.7×
for Dynamic. The Dynamic policy worked best for MolDyn
because it was able to address load balancing issues without
compromising data locality.

6. RELATED WORK
There has been a lot of past work on reducing synchron-
ization and thread creation overheads. These include SP-
MDization [3], synchronization optimizations [8], and bar-
errier elimination [21]. Researchers have studied the impact
of loop chunking on different parameters of interest. Hari
et al. [12] use loop chunking as a means of efficient schedul-
ing temperature-aware code. OpenMP 3.0 [17] supports
different loop scheduling policies, as specified by the pro-
gressor, 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 schedul-
ing [14]. Akin to chunking, loop scheduling has been di-
ected 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 coalescing) to identify chunks of iter-
ations 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
paper, for languages that support dynamic parallelism with
fine grain synchronization.

7. CONCLUSIONS AND FUTURE WORK
In this paper, we presented a transformation framework
for chunking parallel loops in the presence of synchroniza-
ton operations and exceptions. We presented a systematic
method that extends past classical loop transformation tech-
niques to automate the loop chunking procedure in a safe
way. These transformations resulted in reduced synchron-
zation and scheduling overheads, thereby improving per-
formance and scalability. Our experimental results for 11
benchmark programs on an UltraSPARC II multicore pro-
cessor showed a geometric mean speedup of 0.52× for the
unchunked case and 9.59× for automatic chunking using the
techniques described in this paper. This wide gap undersc-
res the importance of using these techniques in future
compiler and runtime systems for programming models with
lightweight parallelism.
We have also identified opportunities for further refinement of the approach presented in this paper. As mentioned earlier, our framework follows an all-or-nothing approach with respect to transforming \texttt{i-foreach} loops; however, this could be extended to transform selected \texttt{i-foreach} loops and leave the others unchanged. In this work, we also assumed that all programmer-specified conditions guarding a \texttt{next} statement are \textit{invariant} in the initial \texttt{foreach} loop, even though we could handle cases when a \texttt{next} statement is guarded by implicit exception conditions. An extension that supports arbitrary conditional expressions as guards for \texttt{next} operations is a challenging subject for future research.

Acknowledgments

We would like to thank all members of the X10 team at IBM and the Habanero group at Rice for valuable discussions and feedback related to this work, especially Vijay Saraswat, Igor Peshansky, Nate Nystrom, and Pradeep Varma. We are also thankful to all X10 team members for their contributions to the X10 software used in this paper. We gratefully acknowledge support from an IBM Open Collaborative Faculty Award. This work was supported in part by the National Science Foundation under the HECURA program, award number CCF-0833166. Any opinions, findings or conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation. 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.

8. REFERENCES