RI10007 (I1010-001) October 4, 2010
Computer Science

IBM Research Report

Parallelizing Loops in Parallel Programs

Soham S. Chakraborty
Tata Research Development and Design Center (TRDDC)
54-B, Hadapsar Industrial Estate
Pune, MH, 411013
India

V. Krishna Nandivada
IBM Research Division
IBM India Pvt Ltd.
Block – D, Floor 3
Embassy Golf Links Business Park
Bangalore - 560071
India

RI 10007, 01 July 2010 Subject: Parallelization

IBM Research Report

Parallelizing loops in Parallel Programs

Soham S Chakraborty[1]
Tata Research Development and Design Center (TRDDC),
54-B, Hadapsar Industrial Estate,
Pune, MH, 411013, India.

V. Krishna Nandivada
IBM India Pvt Ltd,
IBM Research India
Block – D, Floor 3,
Embassy Golf Links Business Park
Bangalore - 560071

1 Work done at IBM Research India.

Abstract

The advent of multi-core systems has taken away the luxury of automatic speedups from legacy applications, which was otherwise possible in the era of uni-core systems (the single core performance used to double every 18 months or so, thus improving the performance of the application). These applications have to be explicitly re-targeted to derive benefits from the increasing speedups of the multi-core systems. One of the key requirements of achieving better utilization of multiple available cores is that of (further) parallelization of code. This requirement is also pertinent in the context of new task-parallel languages (such as Cilk++, Intel Thread Building Blocks, Java Concurrency, OpenMP, Chappel, Fortress, X10, HJ and so on), where the user may start with a partially parallelized version of the program, and then proceed to further parallelize it in an incremental fashion.

In this paper, we demonstrate the insufficiency of traditional loop parallelization techniques for programs that contain explicit parallelism and then present an extension to the classical loop parallelization techniques for semantic preserving parallelization of loops in parallel programs. We present our techniques in the context of a refactoring framework which can help an application developer to incrementally parallelize loops in explicitly parallel programs. In the process, we extend the state of the art in two aspects: (a) Our parallelization framework depends on the May-Happen-in-Parallel (MHP) information, which needs to be recomputed after every refactoring, and thus the complexity of MHP computation gets added to the cost of our refactoring. Owing to the high computational complexity of the MHP algorithm of Agarwal et al [2] ($O(N^3H)$ to compute the set of all statements that may run in parallel with a given statement, where N is the number of statements in the program and H is the height of the program structure tree (PST)), the complexity of our proposed approach can become $O(N^3H)$. To improve the complexity of our overall algorithm, we present a novel incremental MHP analysis, whose complexity is $O(N^2)$), which in turn helps reduce the complexity of our overall algorithm to $O(N^2)$. (b) To improve the opportunities for parallelization we present a scheme of partial privatization, where we have a handle on the space overhead associated with the traditional privatization techniques. The techniques presented in the paper are not fully automatic - may use some manual intervention to fine tune the scope and effectiveness of the transformation, thereby bypassing some of the pitfalls of automatic parallelization techniques. We have used our techniques on varied parallel benchmark programs and have been able to derive encouraging results.

I. Introduction

In the uni-core era legacy applications routinely enjoyed speedups derived directly from improvements in the underlying hardware. Multi-core systems bring with them a new challenge to the application performance arena: since the multi-core hardware speedup improvements are not resulting from spectacular improvements to the speeds of the individual cores, extending the speedups to the applications targeted to the uni-core systems require either improved runtime (for example, operating system, hypervisor), or improvements to the applications. The former approach can be beneficial to those legacy applications where the source code is not available or where the runtime parameters greatly influence the performance gains, which are otherwise not known at the source code level. One drawback of this approach is that the operating system or hypervisor may not be able to utilize the structure of the program and the expertise of the programmer who might be able to assists with the task. Thus, rewriting (or porting) existing application to suit the needs of multi-core systems is gaining interest. A possible target for rewriting is code parallelization; a tool to selectively *refactor* parts of the code to parallelize would be a great help in this direction. A refactoring tool transforms selected parts of a computer program without modifying its external functional behavior. Such a tool can also be used to incrementally parallelize programs written in task parallel languages (such as Cilk, Chappel, HJ, X10, and so on) which are being targeted for machines with multiple cores.

In this paper we first show the semantics non-preserving nature of the classical loop parallelization techniques in the context of programs that contain explicit parallelism (such as threads), and then propose an extension for semantics preserving loop parallelization. We present these techniques as a refactoring, in an integrated development environment like Eclipse, to incrementally parallelize applications. Such a refactoring can be quite useful for traditional programmers who may want to start with a sequential version of the application (or with few threads) and incrementally increase the parallelism. Another important use of such an approach is to incrementally tune existing applications (serial or parallel), wherein the programmer starts from the existing application (legacy or otherwise) and uses the refactoring tool to improve the performance in a trial-and-error method.

Figure 1 presents a code snippet from the `FluidAnimate` application, taken from the Parsec 2.1 [6] suite[1]. This benchmark computes one time-step of a liquid simulation using Smoothed Particle Hydrodynamics (SPH) method. It is a multi-threaded benchmark written in C++ using pthreads. The `main` function creates multiple threads, and each thread executes the function `AdvanceFramesMT`. Say, the programmer wants to parallelize the loop labeled `L4` in the function `AdvanceFramesMT`. The precondition to apply the Kennedy Allen [20] loop parallelization algorithm is satisfied (individual iterations of the loop are independent), and thus we parallelize the loop. However, output of the resulting program does not match the original. That is, this parallelization is not semantics preserving. As it is discussed later in this section, such a situation may arise only because of the other threads that may be executing in parallel to the loop `L4` (in this example, other parallel threads are created in the `main` function).

[1]We have inlined couple of functions for the ease of presentation.

```
int main(int argc, char **argv){
 ...
 for(int i = 0; i < threadnum; ++i) {
    pthread_create(&thread[i], &attr,
       AdvanceFramesMT, &targs[i]);
 } ... }
void *AdvanceFramesMT(void *args) {
 thread_args *targs=(thread_args *)args;
 for(int i = 0; i < targs->frames; ++i){
 ...
  for(int iz = grids[i].sz;
          iz < grids[i].ez; ++iz)
   for(int iy = grids[i].sy;
           iy < grids[i].ey; ++iy)
    for(int ix = grids[i].sx;
            ix < grids[i].ex; ++ix){// L4
     int index = (iz*ny+iy)*nx+ix;
     Cell &cell = cells[index];
     int np = cnumPars[index];
     for(int j = 0; j < np; ++j) {
      cell.density[j] = 0.f;
      cell.a[j] = ...;
     } } ... } }
```

Fig. 1.   Snippet of FluidAnimate benchmark from Parsec 2.1.

```
final int[]A=new int[N+1];
//initialized to zero.
{
  // Loop L1
  for(int i:[0..N])
    A[i] = i;
}
{
  //correctness assertion
  assert (A[1]>=A[2]-1);
}
```

(a)

```
final int[]A=new int[N+1];
//initialized to zero.
finish {
 async { // X1
  // Loop L2
  for(int i:[0..N])
    A[i] = i; }
 async { // X2
  //correctness assertion
  assert (A[1]>=A[2]-1);}
} // finish
```

(b)

```
final int[]A=new int[N+1];
//initialized to zero.
finish {
 async { // X1
  // Loop L2
  forall(int i:[0..N])
    A[i] = i; }
 async { // X2
  //correctness assertion
  assert (A[1]>=A[2]-1);}
} // finish
```

(c)

Fig. 2.   Asynchrony and loop parallelization. (a) Serial program: loop parallelization is semantics preserving. (b) Equivalent parallel program. (c) Naive loop parallelization (replacing the serial `for` loop by a parallel `forall` loop) of the parallel program may not be semantics preserving.

To help explain the relevance of other parallel threads in loop parallelization, we present a toy example written in X10 (the language that we will be using as the reference language in this paper) in Figure 2. [In X10, `async S` creates a new child activity (thread) to execute the statement `S`, and `finish S` is a barrier statement that waits for all the child activities created in `S` to terminate. We use a `forall` loop[2] as a parallel for-loop. A brief background to the X10 language can be found in Section II.]

Figure 2(a) shows a piece of serial code that first creates an array `A` (and initializes all elements to zero). Later it updates the array in a `for` loop. Finally it asserts on a predicate enforcing the dataflow semantics. Figure 2(b) shows an equivalent parallel program that executes the `for` loop and the correctness assertion in two different activities. It can be easily seen that in both these program snippets the assertion would never fail. The traditional loop parallelization techniques would allow the parallelization of the loop `L1` and would lead to a semantically correct translation for the code snippet in Figure 2(a). However, for the code snippet Figure 2(b) such a parallelization (the `for` loop `L2` being turned into a parallel `forall` loop- as shown Figure 2(c)) is not semantics preserving. To realize the same, consider a trace of execution of the program shown in

[2]A `forall (int i:[0..N])` loop creates `N+1` number of activities and waits for all the activities to terminate; each activity executes the body of the loop in parallel for a different value of `i`. In the context of X10 it can be seen as a syntactic sugar for `finish foreach (int i: [0..N])`

Figure 2(c):

1) Activity `X1` starts and `N+1` number of parallel iterations are forked in loop `L2`.
2) The third iteration of the `forall` loop finished execution first (assigns `A[2] = 2`).
3) Activity `X2` (which may run in parallel with `X1` and the different iterations of the `forall` loop) starts and the correctness assertion is tested (`A[1]>=A[2]-1`), which fails because `A[1]= 0`.

Thus, a naive parallelization of the `for` loop in Figure 2(b) may violate the program semantics, because of the interaction between the parallel activities of the newly created `forall` loop and the existing parallel activities. While it is relatively easier to infer (and remove) such dependencies manually in a toy program like the one shown in Figure 2(b), it can be fairly non-trivial to infer (and remove) such dependencies for a complex program like the one shown in Figure 1. Here, because of the interaction between the different threads created in `main` and the iterations of the loop `L4`, independently parallelizing the loop `L4` in is not semantics preserving. Thus, it is important for a loop parallelizer to be aware of the possible asynchrony in the program. In this paper, we present a scheme to parallelize loops in parallel programs in a semantics preserving way.

We present the loop parallelization techniques in the context of a refactoring tool. The developer identifies a loop to be parallelized and invokes our tool. The main challenge of our tool is to identify if the loop parallelization is semantics preserving. Traditional loop parallelization techniques parallelize a loop if the re-ordering of the loop iterations does not change the behavior of that particular loop. In this paper, we go beyond that and reason about the correctness of loop parallelization by considering the impact of re-ordering of the loop iterations on the other activities executing in parallel. We define a notion of *destructive interference* of a loop with the rest of the program to help infer the impact. Intuitively, a loop destructively interferes with a program, if the iteration order of the loop impacts the program output. We summarize our observation as a safety theorem to establish that a loop can be safely parallelized if there is no destructive interference between the loop and the program.

Contributions: Our contributions are summarized below:

• We present a framework to *safely* parallelize loops present in parallel programs. For a loop that destructively interferes with the program we use data privatization to eliminate destructive interference and parallelize the selected loop in a semantic preserving way. Considering the associated large memory overhead of traditional privatization techniques, we present *partial privatization* – an extension to the traditional techniques to reduce the associated large memory overhead.

• Our refactoring framework depends on the May-Happen-in-Parallel (MHP) analysis to identify other activities that may run in parallel with the loop under consideration. This requires recomputation of the MHP information before each transformation, resulting in high computational cost. To improve the speed of the refactoring tool, we present a fast incremental MHP analysis. In this approach given a program and its current MHP information, if a loop is parallelized then the MHP information is updated incrementally by analyzing the transformed code. Compared to the complexity of $O(N^3H)$ that results from using the algorithm of Agarwal et al [2] to compute the set of all statements that may run in parallel with the loop where N is the program size and H is the height of the program structure tree [2], our algorithm has an cost of $O(N^2)$.

• We have applied our parallelization techniques on a varied set of benchmarks taken from Parsec [6] (written in C/C++), HPC Challenge [30], and NewJavaGrande [18] suites (written in X10) and have derived encouraging results.

A. Related Work

Parallelization: Traditional automatic parallelization techniques have been well studied in both research and academic communities [5], [7], [15], [16], [17], [20], [31], and some of these techniques have been extended to introduce parallelism as a refactoring [19], [9], [10], [21], [24]. Sundar et al [32] and Asuncion [3] present techniques to parallelize programs in an incremental fashion. All of these studies concern parallelization in the context of serial code. In this paper, we present a framework to further parallelize already parallel programs. Markstrum et al [25] present a refactoring tool based approach to introduce parallel tasks in X10 programs. Chakraborty and Nandivada [8] present a refactoring scheme to automatically infer distributions for data and computation. In this paper, we present a framework for semantically correct loop parallelization. Such a tool can be used for incremental parallelization of code in steps – the programmer may parallelize some parts of the code first (for instance, by a tool similar to that of Markstrum et al [25]), and then proceed to parallelize loops. Ideally this will be an iterative process.

Privatization: Traditionally privatization techniques [20], [23] have been applied to help improve parallelization of code. One main drawback of these algorithms is that they can incur large memory overhead. We present an extension to the traditional privatization algorithm based on loop tiling [20], called partial privatization, where the developer has a handle on the associated space overhead. To further reduce the memory overhead we can use the techniques presented by Gupta [14] to privatize only relevant portions of arrays.

MHP Analysis: Traditionally May-Happen-in-Parallel (MHP) analysis has been explored in the context of several parallel programming languages [4], [22], [11], [26], [27], [28]. Taylor [33] shows that the hardness of MHP analysis for all pairs of statements in a program is undecidable in general, and is NP-complete for programs that use low level synchronization primitives like Ada *rendezvous*. Agarwal et al [2] extend the traditional MHP analysis to a subset of X10 that does not have low level synchronization primitives. Owing to the high computational costs, these analyses that deal with the whole program

```
1  Function Parallelize($L_1, L_2, success$)
   Input: Loop $L_1$
   Output: Loop $L_2$, boolean $success$
2  begin
3      Matrix $M$ = Direction matrix for the loop $L_1$;
4      $success$ = false ;
5      if $M$ has all '=' entries then
6          $success$ = true ;
7          $L_2$ = parallelized loop for $L_1$, obtained by replacing the header of $L_1$ by forall loop, and appropriate
           transformations to the body;
8      else
9          $L_2 = L_1$; // No change.
10 end
```

Fig. 3. Adapted Loop Parallelization algorithm of Kennedy and Allen.

are not suitable for a refactoring framework, that incrementally parallelizes programs, as the MHP information needs to be computed after each transformation. For any given program, we start with the MHP information (as computed by Agarwal et al [2]) and as the developer parallelizes different parts of the program, we incrementally update the MHP information (without having to recompute the MHP information for the whole program) using a fast algorithm.

Rest of the paper is organized as follows: We first present a quick introduction to X10 language and traditional parallelization techniques in Section II. We discuss our parallelization techniques in Section III. We present out extensions to the MHP analysis in Section IV. We discuss our evaluation of the presented techniques in Section VI. Finally, we conclude in Section VII, along with a discussion on the future work.

II. Background

X10 *Background:* In this section we present a brief background to some of the relevant features of X10 for this paper by summarizing three constructs: `async`, `finish`, and `foreach`. Details about standard X10 v0.41 can be found in the X10 reference manual [12]. Other X10 constructs like points, regions, places and clocks are not central to the paper and are omitted. In this paper, we assume a simplified syntax: every statement has an unique associated label.

Async is the X10 construct for creating (forking) a new asynchronous activity (thread). The statement, `async S`, causes the parent activity to create a new child activity to execute `S`. Execution of the `async` statement returns immediately i.e., the parent activity can proceed immediately.

`finish S` is a structured barrier statement, wherein the body `S` is executed with a surrounding barrier such that all activities created inside `S` have to terminate before the activity executing the barrier can proceed. At runtime, each instructions executed in an X10 program has an unique associated activity and which in turn has an unique *immediately enclosing finish* (IEF) barrier. Any exception thrown by an activity is propagated to its IEF instance. The IEF catches all the exceptions thrown in the body, bundles them into a single exception of type `MultiException`, and throws the bundled exception as a new exception.

The statement `foreach (int p : [m..n]) S` supports parallel iterations over all the points in the region of indices `[m..n]`, by launching each iteration as a separate `async`. The statement `foreach (int p : [m..n]) S` is equivalent to `for (int p : [m..n]) async {S}`. Similarly, `forall (int p:[m..n]) S` is used as an abbreviation for `finish foreach (int p: [m..n]) S`.

A. Loop parallelization algorithm

In this section, we briefly present the loop parallelization techniques of Kennedy and Allen [20] in the context of serial X10 programs. Figure 3 presents the algorithm to parallelize an input loop `L1`. The output variable `L2` contains the transformed loop, and the status is reflected in the output variable $success$. Since we are only parallelizing a particular loop, the direction matrix [20] M is a column matrix. The algorithm first checks that the loop is parallelizable. If the elements in M are all '=' then the loop carries no dependencies, and it is considered parallelizable. If the loop is parallelizable then it parallelizes the loop by modifying the loop header, and possibly the body (line 7). The modifications to the loop body may involve privatization of scalars, scalar vectorization, loop distribution, code replication, statement reordering, insertion of atomic sections, and other standard helper transformations. The discussions about these helper functions is beyond the scope of this manuscript. Interested reader may refer to [20].

B. May Happen in Parallel Analysis

May-Happen-in-Parallel (MHP) analysis determines if dynamic instance of two statements (or the same statement) may run in parallel. We now briefly describe the intra procedural MHP analysis technique proposed by Agarwal et al [2], that works on the subset of X10 described above.

The MHP algorithm of Agarwal et al takes as input two statements s_1, s_2, and the *program structure tree* for the procedure in which the two statements occur, and updates two output variables: (a) a boolean variable to reflect if s_1 and s_2 may run in parallel, and (b) a condition vector under which they may run in parallel. A program structure tree (PST), is a program representation that compresses an abstract syntax tree to consider only nodes of the following types `root, statement, loop, async, finish` and `isolated`. The `root` type corresponds to the start of the procedure, and the `statement` type corresponds all other statements except `loop, async, finish` and `isolated`.

The parallelization framework presented in this paper needs to identify all the activities that may run in parallel with a given loop. Figure 4 shows a simple extension to the MHP algorithm of Agarwal et al to achieve the same; it invokes the algorithm of Agarwal et al (`AgarwalMHP`) repeatedly for each activity in the procedure. This algorithm updates the map MHP for the statement s_1, and `CS` map remembers the corresponding condition vector.

```
Function computeMHP($s_1$)
Input: Loop $s_1$
Say $pst$ is the PST of the procedure under consideration;
foreach activity $s_2$ in the Program do
    AgarwalMHP($s_1, s_2, pst, mhp, CS$);
    if $mhp$ then
        $\mathsf{MHP}(s_1) = \mathsf{MHP}(s_1) \cup \{s_2\}$;
        $\mathtt{CS}(s_1, s_2) = CS$;
```

Fig. 4. All statements running in parallel.

The complexity of the algorithm `AgarwalMHP` is $O(N^2H)$, where N is the number of the statements in the program, and H is the height of the PST. Thus the overall complexity of the function `computeMHP` is $O(N^3H)$; in the worst case quartic in the program size.

III. Parallel Programs and Loop Parallelization

Traditional auto-parallelization techniques have mostly restricted themselves to parallelization of loops in the context of serial programs. As discussed in Section I, parallelization of loops in parallel programs is more involved; we needs to take into consideration any other activities that might be running in parallel and accessing *shared* data. The data accessed by variables visible across multiple threads is considered shared and such variables are called shared variables.

We first present a few definitions in Figure 5 that will be used in this paper. The set of program labels is given by L. Our internal representation of the program has only simple expressions and hence, each expression has an unique label associated with it. The set of abstract activities [1] is given by $\mathcal{A}$. For each activity x, there exists a set $\mathsf{MHP}(x)$ of activities that may run in parallel with x; P(X) denotes the power set of the set X. To start with, we compute the MHP map using the techniques discussed by Agarwal et al [2], and later present an novel incremental algorithm to update the MHP information (see Section IV) in an efficient way. `Loops` gives the set of labels of all the loops present in the program. Each loop belongs to exactly one abstract activity; we call it as the *container* activity of the loop. We use the map C to return the container activity for any loop. We use a helper map $\mathcal{M}$ to return the set of activities that may run in parallel with the container activity of the input loop. We compute this map by: $\mathcal{M}(L_x) = \mathsf{MHP}(\mathsf{C}(L_x))$. Besides these maps, we identify a subset of all the program variables called *output variables*.

Definition 3.1 (output-variable): A variable whose value is part of the observable behavior of a program is an output variable for the program.

L: Set of labels	MHP: $\mathcal{A} \rightarrow \mathsf{P}(\mathcal{A})$
$\mathcal{A}$: Set of abstract activities	C: `Loops` $\rightarrow \mathcal{A}$
`Loops`$\subseteq$ L : Set of labels of the loops	
$\mathcal{M}$:`Loops` $\rightarrow \mathsf{P}(\mathcal{A})$	

Fig. 5. Basic definitions used in the paper.

```
final int[]B=new int[2];//initialized to zero
final int[]C=new int[2];//initialized to zero
for (i=0;i<2;++i) B[i] = i;
finish {
  /* A0 */ async { for (j=0;j<2;++j)
                      B[j] = ...; }
  /* A1 */ async { C[0] = B[0]; ... }
  /* A2 */ async { C[1] = B[1]; ... }
}
System.out.println (C[0]);
System.out.println (C[1]);
```

Fig. 6. Example of non-destructive interference.

From an application developer perspective (who uses our refactoring framework) the observable behavior of a program consists of only the program output. Thus the argument to `System.out.println` is an output variable[3]. We now define *destructive interference* that is central to our parallelization algorithm presented later in this section.

Definition 3.2 (Destructive interference): Given a program `P` and a loop `L` therein, if the value of any output-variable in `P` depends on the iteration order of `L` then `L` is said to destructively interfere with `P`.
An example of destructive interference can be seen in Figure 2(b), where the loop `L2` destructively interferes with the program: the value of the output-variable (argument to `assert`) depends on the order in which the iteration of the loop are executed.

Figure 6 presents a skeleton of a program in bounded buffer producer consumer paradigm, where the producer activity `A0`, and the two consumer activities `A1` and `A2` are all running in parallel. The producer (`A0`) updates the array `B`, and consumers (`A1` and `A2`) consume the data and write to a shared variable `C`. This program has no destructive interference since the value output by the program does not depend on the order in which `B[0]` and `B[1]` are updated (each being updated in a different iteration). And we observe that it is *safe* to parallelize the loop in the first `async` block. We first present the definition of safety, and then summarize our key finding as a theorem.

Definition 3.3 (Safety): Say `P` is a program and `P'` is the transformed program obtained by applying a transformation `F` on `P`. The transformation `F` is considered safe, if the set of observable behaviors of `P` and `P'` are identical.

Note that, we are considering the equivalence of the original and the transformed programs only in terms of the observable behavior; this is sufficient from the perspective of the developer using the refactoring tool. Another point to note is that, an asynchronous program `P` can have more than one correct observable behavior (for instance, because of multiple write statement executing in parallel). Hence, we base our notion of safety on the comparison of the set of observable behaviors of `P` and `P'` – that is every observable behavior of `P'` is also shown by `P` and vice versa. Such an equivalence is also dependent on the underlying memory model. Even using a stricter memory model like sequential consistency, prior research [33], [13] has shown the problem to be NP-complete (thus it is applicable to weaker memory models as well). We now summarize our central theorem.

Theorem 3.4: Given a program `P` and a loop `L` therein, if `L` can be parallelized by the algorithm presented in Figure 3, the parallelization transformation is safe if and only if there is no destructive interference between `L` and `P`.

Proof: (*Sketch*)
// By the definition of destructive interference
(`L` destructive interferes with `P` $\Leftrightarrow$ iteration order of `L` has to be preserved)
$\Leftrightarrow$
// Applying the rule $(p \Leftrightarrow q) \Leftrightarrow (\neg p \Leftrightarrow \neg q)$
($\neg$(`L` destructive interferes with `P`) $\Leftrightarrow$ $\neg$ (iteration order of `L` has to be preserved))
$\Leftrightarrow$
// Simplifying the negation expressions
(`L` does not destructive interfere with `P` $\Leftrightarrow$ iteration in `L` can be executed in any order)
$\Leftrightarrow$
// By the definition of Safety
(`L` does not destructive interfere with `P` $\Leftrightarrow$ it is safe to parallelize `L`)

Hence the proof. ■

It may be noted that our notion of destructive interference is stricter than that of the "data race" [29]. For instance, the example in Figure 6 has data-race, but the loop can still be parallelized, because of the absence of destructive interference.

[3]Our internal representation is in three address format, and thus `System.out.println` will take only one argument. Further, assertions and uncaught exceptions are also observable and are treated similarly.

Thus, using data-race instead of destructive interference would lead to parallelization of fewer programs. Further, the notion of parallelism-inhibiting edges described for loop fusion [20] is stronger than our notion of destructive interference and is insufficient.

Considering the Theorem 3.4 and the complexity of establishing the safety of parallelization (NP-complete), the hardness of establishing destructive interference is readily understood. To make the problem tractable, we now present a conservative definition of destructive interference based on the following observation and lemma:

Observation: A conservative over-approximation of the set of output-variables is given by the set of *pseudo-output*-variables. A pseudo-output-variable (`POV`) can be an argument to a function call, or be an argument to a return statement, or be stored in a shared variable, or be the argument of an output statement. It may be noted that, we may omit the arguments to a function call from the set of pseudo-output-variables, if source code for the function is available.

Lemma 3.5: The value of a `POV` in an activity, depends on the iteration order of a loop, only if the `POV` depends on at least two of the shared elements updated in the loop body.

Proof: (*Sketch*) Proof by contradiction: Say the value of a `POV` x (accessed in an activity A) depends on a single shared variable v updated in Loop `L` that runs in parallel with A, and reordering the iterations in `L` may result in a new value for x that was otherwise not possible. But this is a contradiction: since A and `L` are running in parallel, x can be computed based on the value of v computed in any iteration of `L` or the original value of v, and hence reordering iterations of the loop `L` would not lead to any new value for x. Hence the proof. ■

Conservative destructive interference: For a given loop L and an activity $A \in \mathcal{M}(L)$, L destructively interferes with A if any `POV` in A depends on the updates of two or more shared variables in L.

A loop `L` destructively interferes with a program `P` *iff* $\exists A \in \mathcal{M}(L)$, such that `L` destructively interferes with `A`. We use `CDI(L, A)` to say that there is conservative destructive interference between `L` and activity `A`. Similarly, we use `CDI(L, P)` to say that there is conservative destructive interference between `L` and program `P`. We now present the modified theorem based on the conservative destructive interference.

Theorem 3.6: Given a program `P` and a loop `L` therein, if `L` can be parallelized by the algorithm presented in Figure 3, the parallelization transformation is safe if there is no conservative destructive interference between `L` and `P`.

Proof: (*Sketch*)

// From the definition of conservative destructive interferences

($\exists A \in \mathcal{M}$(L): a `POV` in A depends on the updates of two or more shared variables in `L` $\Rightarrow$ `CDI(L, A)`)

$\Rightarrow$

// From Lemma 1

(iteration order of `L` has to be preserved $\Rightarrow$ $\exists A \in \mathcal{M}$(L): `CDI(L, A)`)

$\Leftrightarrow$

// Applying the rule $(p \Rightarrow q) \Leftrightarrow (\neg q \Rightarrow \neg p)$

($\neg$($\exists A \in \mathcal{M}$(L): `CDI(L, A)`) $\Rightarrow$ $\neg$ (iteration order of `L` has to be preserved))

$\Leftrightarrow$

// Definition of `CDI(L,P)`*, and simplifying negation expression*

($\neg$`CDI(L,P)` $\Rightarrow$ iteration order of `L` need not be preserved)

$\Leftrightarrow$

// By the definition of Safety

($\neg$`CDI(L,P)` $\Rightarrow$ it is safe to parallelize `L`)

Hence the proof. ■

We now revisit the code snippet shown in Figure 6. The `POV C[0]` in activity `A1` depends only on one shared variable (`B[0]`) in the loop. Similarly, the `POV C[1]` in activity `A2` depends only on one shared variable (`B[1]`) in the loop. Hence, the loop does not destructively interfere with any activities executing in parallel and is safe to parallelize.

It may be noted that conservative destructive interference is a sufficient condition and not a necessary one for loop parallelization. Figure 7 shows an example program with destructive interference, where it is safe to parallelize the loop. The loop `L1` updates the variable `x`, and based on the iteration order, the value of `x[0]` will be `1` and the value of `x[1]` will be `2`, or vice versa. Which in turn, impacts the `POV`s `y[0]` and `y[1]`. Thus, the iteration order in which the shared variables (`x[0]` and `x[1]`) in Loop `L1` are updated affects the values of the output variables `y[0]` and `y[1]`. In other words, `L1` destructively interferes with `A2`. However, it is still safe to parallelize `L1` (the output is one of `0, 1, 2` or `3` irrespective of if the loop is parallelized or not).

A. Loop parallelization algorithm

Traditional loop parallelization techniques discussed in Section II-A process each of the loops in an independent fashion - the decision to parallelize the loop and the resulting parallelized code depended only on the loop, and not on any other parts

```
Stack stk = new Stack();
stk.push(1); stk.push(2);
final int[]x=new int[2]; //initialized to zero
final int[]y=new int[2]; //initialized to zero
finish {
  async { // A1
     for (i=0;i<2;++i) { // L1
           x[i] = stk.pop();
     }
  }
  async { // A2
     y[0] = x[0];
     y[1] = x[1];
  }
}
System.out.println(y[0] + y[1]);
```

Fig. 7. Destructive non-interference is not necessary for safe parallelization.

of the program. As explained in Section I, we cannot parallelize loops in asynchronous programs in a standalone mode. Both the decision to parallelize and the generated code depends on other interfering activities.

It can be easily seen that the traditional loop parallelization algorithms defined for serial programs can also be directly applied to all those loops that do not destructively interfere with the program (with other activities). A naive approach to parallelize loops in asynchronous programs is to first check if the given loop destructively interferes with any other activity that may run in parallel and parallelize only if it does not.

The all-or-none approach of the naive approach is overly conservative. A straight forward way to eliminate destructive interference would be to privatize all the shared variable accesses (present between the loop and the activities running in parallel), and use copy-back [7] operations to reflect the modifications at different activities. These copy-back operations can be fairly expensive as they may incur significant memory operations cost (multiple writes to the same memory location need to be identified, and a private copy of the variable is required for each parallel write), and may further require some synchronization operations or runtime evaluation to identify the last-writes [34]. In this section we present our scheme that avoids the use of copy-back operations altogether in the interfering activities. Unlike the prior research work on privatization, where the goal is to help make the loop parallelizable by removing the loop carried dependences, our attempts at privatization is directed at removing the dependence across multiple activities.

Before we present our algorithm, we first present a categorization of loops: For any loop L_i, say `V` be the set of variables updated in L_i and are also accessed (read and/or written) in the activities contained in the set $\mathcal{M}(L_i)$. Loop L_i is categorized as `read only`, if the variables present in `V` are only read in L_i. Loop L_i is categorized as `read & write`, if the variables present in `V` may be both read and written in L_i. We present our loop parallelization algorithm in Figure 8. We address the problem of parallelizing the input loop L_i based on its categorization.

case I: L_i is `read only` with respect to `V`; no destructive interference possible. We can parallelize L_i using the algorithm discussed in Figure 3 (if the algorithm permits parallelization).

case II: L_i is `read & write` with respect to `V`. The algorithm presented in Figure 3 will parallelize a loop, only if different iterations of the loop are independent. Thus, if the algorithm parallelizes a `read & write` loop, it will be a valid transformation in isolation. However, if the activities in the set $\mathcal{M}(L_i)$ read two or more of the shared variables that are written in L_i (from the set `V`), then parallelizing L_i may have an impact on their semantics. As discussed earlier in this section, it depends if L_i interferes with the activities in the set $\mathcal{M}(L_i)$ in a *destructive* way. We use the function `ChkCDI` to compute conservative destructive interference. It takes as input a loop L and an activity A that may run in parallel with L. It updates two output parameters: (i) boolean variable $interferes$ is set if L destructively interferes with A, and (ii) V' is updated with the set of variables that contribute to the destructive interference.

For all the activities destructively interfering with the input loop, we identify all the variables contributing to the destructive interference and introduce privatization code before the invocation of the first of the interfering activities. The rest of the reads of the shared variables are replaced with reads to the privatized copies. Thus, both the decision to parallelize a loop and the generated code depend on the of the other interfering activities (privatized variables are updated in the activities that may run in parallel).

Figure 9 presents an algorithm to insert privatization code for a shared variable v, for all the destructively interfering activities of the input activity A. It uses a helper function $\mathsf{Slice}(L_2, L_1, v)$ that returns the set of labels of statements part of the backward slice for the variable v from L_2 to L_1. We first identify all the interfering activities, and identify the point where we can introduce the privatization code; this point is the last program point at which the variable is updated before the invocation

```
Function ParallelizeNew($L_i$, $L_o$, $success$)
Input: Loop $L_i$
Output: Loop $L_o$, boolean $success$
begin
    $V$ = set of variables shared between $L_i$ and $\mathcal{M}(L_i)$;
    switch (category of loop $L_i$) do
        case read only:
            Parallelize($L_i, L_o, success$)
        case read & write:
            Parallelize($L_i, L_o, success$);
            if $V$ is empty OR $\neg$ success then
                return;
            foreach $A_t \in \mathcal{M}(L_i)$ do
                boolean $interferes$;
                ChkCDI($L, A_t, interferes, V'$);
                if $interferes$ EQ $true$ then
                    privatize($V', A_t$);
end
```

Fig. 8. Algorithm to parallelize a loop in an asynchronous program

```
Function privatize($V, A$)
Input: Variables $V$, Activity $A$
begin
    Let $L_1$ be the label of the statement creating the activity $A$;
    $V' = V \cap$ set of variables accessed in $A$;
    foreach $v \in V'$ do
        Insert privatization code ($private_v = v$;) before $L_1$;
    foreach statement $s$ present in the activity $A$ do
        $v$ = the variable defined in the statement $s$;
        if $v \in V'$ then
            Insert code $privatize_v = v$; after $s$;
        $v_1, v_2$ = variables used in the statement $s$;
        if $v_1 \in V'$ then
            Replace $v_1$ with $privatize_{v_1}$ in $s$;
        if $v_2 \in V'$ then
            Replace $v_2$ with $privatize_{v_2}$ in $s$;
end
```

Fig. 9. Privatization algorithm

of any of the interfering activities. We insert the privatization code in line 5. After each of the write statements reaching the interfering activities, and present in the interfering activities, we add privatization code (line 10); privatization code for an array would involve inserting a loop to initialize the privatized copy. All the reads to the variable v in the interfering activities are replaced by reads to the variable $privatize_v$ (line 14). Figure 10 shows the privatization required for parallelizing the for-loop in Figure 2(b).

The paralleliztaion algorithm shown in Figure 8 requires up-to-date may-happen-in parallel (MHP) information (to compute the $\mathcal{M}$ map in Line 3). And we invoke the computeMHP algorithm discussed in Section II, everytime we require the MHP information.

It may be noted that the **case I** mentioned above is a special case of the **case II**. We separate these, so as to avoid privatization

```
final int[]A=new int[N+1];
final int AT[1] = A[1];
final int AT[2] = A[2];
async {// Loop
       for (int i:[0..N])
                 A[i] = i;
}
async {
       assert(AT[1]>=AT[2]-1);
}
```

Fig. 10. Privatization required for Figure 2(b)

Input Loop

```
L: for (i=0;i<n;i++){
     A[i] = i;
   }
```

Transformed loop
Step 1: Tiling

```
L1:for(int j=0;j<n/B;j++){
 L2:for(i=j;(i<j+B && i<n);i++){
      A[i] = i ; } }
```

Step 2: Parallelize

```
L1:for(j=0;j<n/B;j++){
   parallelize(L2, success)
  }
```

Fig. 11. An example illustrating partial privatization via tiling.

in some cases.

Complexity: The complexity of each of the functions `Parallelize`, `ChkCDI`, and `privatize` is $O(N)$, where N is the number of statements in the program. Thus the cost of the *switch* statement is $O(N^2)$. The cost to compute the $\mathcal{M}$ map is $O(N^3H)$, where H is the height of the PST, and in the worst case it becomes $O(N^4)$. Thus, because of the expensive MHP function, the overall complexity of our `ParallelizeNew` algorithm is $O(N^4)$. In the Section IV we present extensions to the MHP analysis to improve upon this complexity.

B. Partial Privatization via Loop Tiling

Traditionally array privatization has been applied as a means to improve the chances of parallelization. This involves creating a copy of the input array and using the privatized copy at program points requiring private access. But creating a copy of a large sized array can be expensive and restrictive in terms of memory space. In this section, we extend the traditional notions of loop tiling to help reduce the memory space overhead that is otherwise traditionally associated with privatization. This scheme can be especially useful for privatizing large sized arrays.

We use a technique that uses the classical array privatization algorithm of Li [23], and loop tiling [20] techniques as black boxes to privatize a given array A, accessed within a loop L. We illustrate our idea using an example shown in Figure 11. Say the input loop is to be parallelized after the array A has been partially privatized. Our proposed technique would first tile the loop (say, using a user specified blocking faction `B`), and then invoke the parallelization algorithm on the inner loop, such that the sub-array accessed within the inner loop is fully privatized. This partial privatization scheme is can overcome some possible memory overhead issues, by reducing the memory requirement (at the cost of reduction in parallelism).

C. Exception handling

Exception semantics of the underlying language plays an important role in loop parallelization. For instance, as per X10 semantics an exception thrown inside an iteration of a parallel loop (`foreach`) does not terminate the execution of other parallel iterations of the loop. Such exceptions are only caught by the surrounding `finish`, after all the activities forked in the `finish` have terminated. In our loop parallelization algorithm presented in Figure 3, we replace the header of the loop to be parallelized by a `forall` (`finish foreach`) loop. Thus exceptions thrown in any one parallel iteration are caught (by the `finish` surrounding the `foreach`) only after all the other parallel iterations have terminated. This is counter to the original semantics of the program.

Another issue that is specific to X10 is that a `finish` catches all the exceptions thrown in the body, bundles them into a single exception of type `MultiException` and throws the bundled exception as a new exception. Thus the existing exception handlers around the transformed loop have to be rewritten to catch `MultiException`, as the new parallel loop would only throw an exception of type `MultiException`.

Our refactoring tool informs the programmer of the exception semantics and let the programmer make explicit code changes. For instance, the programmer may choose the rewrite the code to *catch* `MultiException` and based on the exception thrown it may execute appropriate code. A trivial extension to our framework can highlight the impacted *catch* blocks in the program editor. A rigorous study of parallelization in the presence of exceptions is left as a future work.

IV. Incremental May-Happen-In-Parallel(MHP)

The parallelization algorithm discussed in section III, depends on the MHP analysis for safe parallelization of loops. As discussed at the end of Section III the complexity of the MHP analysis presented by Agarwal et al [2] is the chief contributor to complexity of our parallelization algorithm. In this section we present a novel incremental MHP analysis that reduces the complexity of MHP analysis and thereby improving the complexity of our loop parallelization algorithm. We ensure that there is no loss of precision compared to the MHP result achieved from reanalyzing the whole program.

Say, a loop L, contained inside an activity A_x, is parallelized and the resulting activities created by the parallel loop are represented by a single activity A_L. Our incremental MHP algorithm is based on the following observations: (a) For the activities created in A_x that are created after the termination of A_L, they do not run in parallel with A_L. (b) A_L runs in parallel with the $\mathcal{M}(L)$.

Figure 12 presents our incremental MHP algorithm. It takes as input a loop L and the new abstract activity A_L corresponding the parallel iterations of the transformed loop. It updates the MHP and `CS` maps, based on the two observations presented above.

```
1  Function incrementalMHP($L, A_L$)
   Input: Loop $L$, $\mathcal{A}$ $A_L$
2  begin
3      $A = \mathsf{C}(L)$;
4      $m = \mathcal{M}(L)$;
5      foreach $s \in \mathsf{AsyncsIn}(A)$ do
6          if $L$ predominates $s$ then
7              $m = m - \{s\}$;
8              CS($A_L, s$) = $\phi$;
9      $\mathsf{MHP}(A_L) = m$;
10     foreach $a \in \mathsf{MHP}(A)$ do
11         $\mathsf{MHP}(a) = \mathsf{MHP}(a) \cup \{A_L\}$;
12         CS($A_L, a$) = CS($a, A$);
13 end
```

Fig. 12. Incremental MHP Analysis

The incremental MHP in Figure 12 works as follows

- If loop L predominates any `async` statement `s` in any of the activities in A (given by $\mathsf{AsyncsIn}$) then we exclude s from $\mathsf{MHP}(A_L)$.
- foreach activity $a \in MHP(A)$, add A_L to $\mathsf{MHP}(a)$.

Since the lifetime of A_L does not exceed that of A this local analysis suffices and is as precise as complete (re)analysis of MHP sets (achieved by invoking `computeMHP`, Figure 4).

We invoke the `incrementalMHP` algorithm after each successful transformation in the function `parallelizeNew` shown in Figure 8. The complexity of computing incremental MHP in each invocation is $O(N^2)$, and this leads to an overall complexity of $O(N^2)$ for the parallelization algorithm. Note that, in this new approach the computaion of $\mathcal{M}(L_i)$ in line 3 of Figure 8 needs just a table look up and does not need to recompute the MHP sets, which is otherwise required in the absence of the call to `incrementalMHP`.

V. Overall view and Discussion

The overall block diagram for our proposed parallelization framework is given in Figure 13. Our proposed parallelization techniques are targeted as a refactoring tool in an integrated development environment like Eclipse. We compute the complete MHP information once (by invoking `computeMHP`) (we can do it when Eclipse opens this particular file), and then for each loop parallelization refactoring, we invoke `ParallelizeNew`, or `Parallelize` depending on if the program contains other parallel activities or not. After that we invoke `incrementalMHP` to set the MHP information up-to-date. By ignoring the one time cost of invoking the `computeMHP` (can be ignored for a sufficiently large number of invocations of the parallelization algorithm), each individual parallelization refactoring has an overall complexity of $O(N^2)$.

• Our incremental MHP algorithm achieves the improvements in the complexity by (a) distributing the work across each invocation of the parallelization algorithm, (b) focussing on only one type of parallel constructs (namely the parallel loops). We are currently working on designing a full fledged incremental MHP algorithm that can incrementally update the MHP

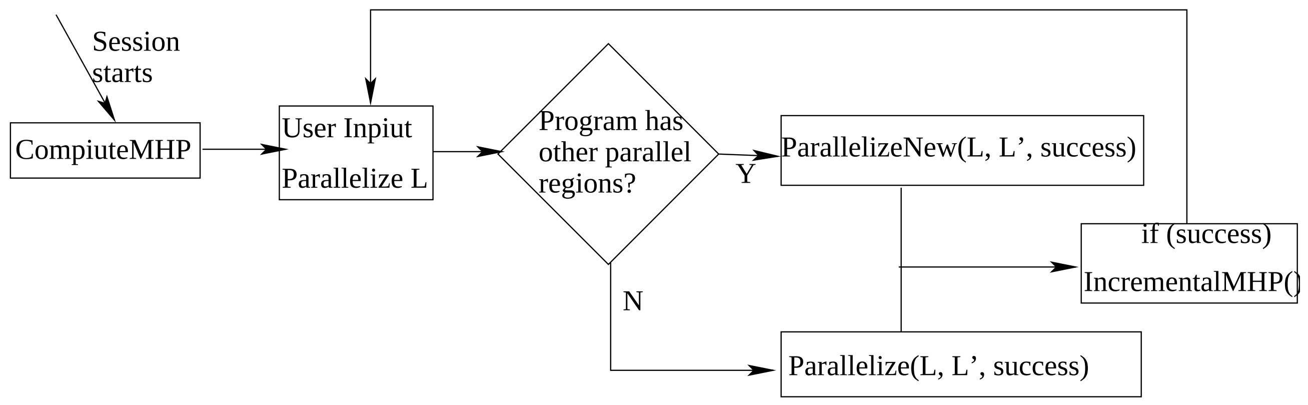
Session starts
CompiuteMHP
User Inpiut
Parallelize L
Program has other parallel regions?
Y
N
ParallelizeNew(L, L', success)
Parallelize(L, L', success)
if (success)
IncrementalMHP()

Fig. 13. Block diagram illustrating a typical invocation of our parallelization framework

information for each type of program update (such as introduction of `finish`, `async` and so on) possible in an development environment. Owing to the complex issues involved, that problem needs to be handled in a standalone fashion and is beyond the scope of this paper.
• In this paper, our incremental MHP analysis does not take into consideration multiple places of the activities. While this remains an interesting exercise, there is limited use of such an analysis in our current set up.
• The usefulness of the proposed incremental MHP analysis goes beyond just loop parallelization. It can be deployed in many integrated development environments where the underlying tool need to maintain up-to-date MHP information.
• Even though we present our techniques in the context of X10, these can be applied to other task parallel languages like HJ, UPC and languages that support parallel loops and threads (such as C++ with pthreads).
• The underlying memory model plays an important role in guaranteeing the correctness of parallelization. We have discussed the hardness of establishing program equivalence even under strict models like sequential consistency in Section III. We use properties of conservative destructive interference (Theorem 3.6) to establish the correctness.
• The techniques presented in the paper are not fully automatic and may need some manual intervention to fine tune the scope and effectiveness of the transformation. Two such instances are: (a) identifying a blocking factor for partial privatization, and (b) modifying the code for semantics preserving exception handling.

VI. Evaluation

In this section, we discuss our experience in applying the techniques presented in this paper onto real world examples. We show the applicability of our techniques, by presenting our experience with diverse benchmarks. Our evaluation re-establishes the insufficiency of traditional loop parallelization algorithms in the context of parallel programs and shows how proposed techniques perform semantics preserving transformations. We have picked the examples from three different benchmark suites (NewJavaGrande Benchmark suite [18], HPC Challenge benchmark suite [30], and Parsec suite [6]), so as to establish the applicability of our techniques.

Figure 14 summarizes the benchmark characteristics in the first five columns. All of these benchmarks have serial loops present in parallel code, and in six out of the nine benchmarks the loops destructively interfere with the benchmark programs. We found that our definition of conservative destructive interference captured all of these cases. The sixth column in Figure 14 presents the result of applying the parallelization algorithm of Kennedy and Allen [20]. It shows that the first three benchmarks where the loops under consideration did not destructively interfere with the programs, the algorithm of Kennedy and Allen [20] resulted in semantically correct translation. We establish the semantic correctness by comparing the output of the translated program with that of the original program. For the rest of the benchmarks it resulted in incorrect translation (generates incorrect results). The last column in Figure 14 shows the result of applying our parallelization algorithm described in Figure 8; for all the benchmark programs our parallelization algorithm resulted in semantically correct translation. We now present a few details from this study.

Parallelizing in the absence of destructive interference (BlackScholes, Swaptions and RandomAccess): BlackScholes is from the Parsec [6] benchmark suite. This benchmark uses Black-Scholes Partial Differential Equation (PDE) to do option pricing. This benchmark, along with Swaptions and FluidAnimate, is a multi-threaded benchmark written in C++ using pthreads. Figure 15(a) shows the snippet of the BlackScholes benchmark ported to X10. The goal here is to parallelize the middle for-loop (`j loop`). Our algorithm identifies that the for-loop may run in parallel with other activities forked from the iterations of the outer loop. However, it finds that there is no destructive interference and thus parallelizes the loop (Figure 15(b)), without any need for privatization. We have observed similar patterns in the Swaptions benchmark of the Parsec suite, and in the RandomAccess benchmark from the HPC Challenge benchmark suite [30].

Benchmark	No of Lines	loops present in parallel code?	destructive interference?	conservative destructive interference?	Semantically correct parallelization ?	
					by Kennedy Allen [20]	by Figure 8
BlackScholes	1661	Y	N	N	Y	Y
Swaptions	1615	Y	N	N	Y	Y
RandomAccess	270	Y	N	N	Y	Y
Moldyn	635	Y	Y	Y	N	Y
Series	483	Y	Y	Y	N	Y
Sor	176	Y	Y	Y	N	Y
Sparsemult	260	Y	Y	Y	N	Y
LU	243	Y	Y	Y	N	Y
FluidAnimate	3492	Y	Y	Y	N	Y

Fig. 14. Characteristics and Scope of Parallelization in Parallel Programs: A Comparative Study

```
for(int k = 0; k < threadnum; ++k){
  ...
  async {
  ...
  for (j=0; j<NUM_RUNS; j++) {
    ...
    for (i=start; i<end; i++) {
      price[i] += BlkSchlsEqEuroNoDiv(
                    sptprice[i],strike[i],
                    rate[i],volatility[i],
                    otime[i], otype[i],0);
  } } }
```

(a)

```
for(int k = 0; k < threadnum; ++k){
  ...
  async {
  ...
  finish foreach(j=0; j<NUM_RUNS; j++) {
    ...
    for (i=start;i<end;i++){
       price[i] += BlkSchlsEqEuroNoDiv(
                     sptprice[i],strike[i],
                     rate[i],volatility[i],
                     otime[i], otype[i],0);
  } } }
```

(b)

Fig. 15. (a) Parallel BlackScholes, (b) Further parallelized BlackScholes. Modifications shown in **bold**.

Parallelizing in the presence of destructive interference (Moldyn, Series, Sor, Sparsemult, LU, FluidAnimate): Moldyn is part of the NewJavaGrande [18] benchmark suite ported to X10. Figure 16(a) shows the parallel version of a part of the Moldyn benchmark. The goal is to further parallelize the program by parallelizing the inner (broadcast) loop. Figure 16(b) shows the transformation as realized by our algorithm. Our parallelization algorithm identifies that due to the shared variable `P`, the loop destructively interferes with the activity (created by the outermost foreach loop). The algorithm privatizes the variable `P` before the outer most `foreach` loop, and parallelizes the broadcast loop; it also uses the privatized copy `PP` in the reduction loop. We have identified similar opportunities in Series, Sor and Sparsemult benchmarks of the NewJavaGrande suite, in the LU benchmark of the HPC Challenge benchmark suite, and in the FluidAnimate benchmark from the Parsec suite.

```
void run() {
 ...
 finish foreach (...)  {
  ...
  // sum reduction
  for (point [j]: R) {
    t.vir += P[j].vir;
    t.epot += P[j].epot;
    ...
  }
  // broadcast
  for (point [j]: R) {
      P[j].vir = t.vir;
      P[j].epot = t.epot;
      ...
  }
 }
}
```

(a)

```
void run() {
 ...
 for (point [j]: R)  // privatization
      PP[j] = P[j];
 finish foreach (...)  {
  ...
  // sum reduction
  for (point [j]: R) {
    t.vir +=  PP[j].vir;
    t.epot +=  PP[j].epot;
    ...
  }
  // broadcast
  finish foreach  (point [j]: R) {
      P[j].vir = t.vir;
      P[j].epot = t.epot;
      ...
  } } }
```

(b)

Fig. 16. (a) Parallel Moldyn, (b) Further parallelized Moldyn. Modifications shown in **bold**.

An interesting point to note in both the examples shown in Figure 15 and Figure 16 is that the actual transformation may look fairly simple and can be done manually. However, the key challenge lies in identifying if there exists destructive interference and, if so, introducing the relevant privatization code at an appropriate program point so as to preserve the original program semantics.

VII. Conclusion and Future work

In this paper, we present a framework for helping application developer to incrementally parallelize loops in parallel programs. We show that the decision to parallelize a loop and the generated code depend not only depend on the particular loop, but also on the other activities that may be running in parallel with the loop. We present an extension to the traditional loop parallelization algorithm to safely parallelize loops in the presence of other parallel activities. To help improve the efficiency of the generated code, we introduce an extension to the traditional privatization techniques to reduce the associated memory overhead. In the process, we identify that the complexity of using the traditional MHP algorithm to determine all the statements that may run in parallel with the loop (quartic in the program size) is a bottleneck for the parallelization algorithm; we present a new efficient incremental MHP analysis (complexity - quadratic in the program size) to improve the efficiency. Such a fast incremental algorithm is particularly suitable for Eclipse type of programming environments, where the application developer is likely to continuously modify the code by writing new parallel loops or parallelizing existing loops. We show the applicability of the presented techniques over benchmarks spanning three different benchmark suites. In this paper, we use X10v1.4 as the basis language for discussing the techniques on parallelizations and distributions. However, as shown in section VI the techniques discussed here are general enough and can be applied to other language frameworks as well.

During our study of many asynchronous programs we have identified new refactoring patterns that involve parallelization of loops guarded by synchronizations such as clocks [12]. Identifying further refactoring patterns and possible optimizations in the generated code would be an interesting area to explore. Implementing the framework in Eclipse type of environment is an involved exercise in itself and is left as future work.

Acknolwedgements

We thank Raghavan Komondoor, Prateek Mehta and R K Shyamasundar for some initial discussions on this paper.

References

[1] S. Agarwal, R. Barik, V.K. Nandivada, R.K. Shyamasundar, and P. Varma. Static detection of place locality and elimination of runtime checks. In *Proceedings of the APLAS*, pages 53–74. LNCS, 2008.

[2] S. Agarwal, R. Barik, V. Sarkar, and R.K. Shyamasundar. May-happen-in-parallel analysis of X10 programs. In *Proceedings of the 12th Symposium on Principles and practice of parallel programming*, pages 183–193. ACM, 2007.

[3] A. Asuncion. Incremental parallelization using navigational programming: A case study. In *Proceedings of the International Conference on Parallel Processing*, pages 611–620. IEEE Computer Society, 2005.

[4] R. Barik. Efficient computation of may-happen-in-parallel information for concurrent java programs. In *The International Workshop on Languages and Compilers for Parallel Computing*, 2005.

[5] M. M. Baskaran, N. Vydyanathan, U. K. R. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. In *Proceedings of the symposium on Principles and practice of parallel programming*, pages 219–228. ACM, 2009.

[6] C. Bienia, S. Kumar, J. Pal Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of PACT*, October 2008.

[7] M. Burke, R. Cytron, J. Ferrante, and W. Hsieh. Automatic generation of nested, fork-join parallelism. pages 71–88, 1989.

[8] S. S. Chakraborty and V. K. Nandivada. Inferring arbitrary distributions for data and computation. In *SPLASH Onward!(To appear)*. ACM, 2010.

[9] D. Dig, J. Marrero, and M.D. Ernst. Refactoring sequential java code for concurrency via concurrent libraries. In *Proceedings of the ICSE*, pages 397–407. IEEE Computer Society, 2009.

[10] D. Dig, M. Tarce, C. Radoi, M. Minea, and R. Johnson. Relooper: refactoring for loop parallelism in java. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 793–794, New York, NY, USA, 2009. ACM.

[11] E.Duesterwald and M.L.Soffa. Concurrency analysis in the presence of procedures using a data-flow framework. In *Proceedings of the symposium on Testing, analysis, and verification*, pages 36–48. ACM, 1991.

[12] V. Saraswat et al. Report on the experimental language X10, x10.sourceforge.net/docs/x10-101.pdf, 2006.

[13] K. Gharachorloo and P. B. Gibbons. Detecting violations of sequential consistency. In *Proceedings of the symposium on Parallel algorithms and architectures*, pages 316–326. ACM, 1991.

[14] M Gupta. On privatization of variables for data-parallel execution. In *Proceedings of the International Symposium on Parallel Processing*, pages 533–541. IEEE Computer Society, 1997.

[15] R. Gupta, S. Pande, K. Psarris, and V. Sarkar. Compilation techniques for parallel systems. *Parallel Computing*, 25:13–14, 1999.

[16] M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, S. Liao, E. Bugnion, and M.S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, 1996.

[17] J. Hordijk and H. Corporaal. The potential of exploiting coarse-grain task parallelism from sequential programs. In *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*, pages 664–673, London, UK, 1997. Springer-Verlag.

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

[19] J.R.Allen and K.Kennedy. PFC: A program to convert fortran to parallel form. In *Proceedings of the Supercomputers: Design and Applications*, pages 186–203. IEEE Computer Society Press, August 1984.

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

[21] K. Kennedy, Kathryn S. McKinley, and C-W. Tseng. Analysis and transformation in the parascope editor. In *Proceedings of the ICS*, pages 433–447, New York, NY, USA, 1991. ACM.

[22] Lin L. and Clarke V. A practical mhp information analysis for concurrent java programs. In *Proceedings of LCPC*, 2004.
[23] Z. Li. Array privatization for parallel execution of loops. In *ICS '92: Proceedings of the 6th international conference on Supercomputing*, pages 313–322, New York, NY, USA, 1992. ACM.
[24] S-W. Liao, A. Diwan, R. P. Bosch, Jr., A. Ghuloum, and M. S. Lam. Suif explorer: an interactive and interprocedural parallelizer. In *PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 37–48, New York, NY, USA, 1999. ACM.
[25] S. Markstrum, R. M. Fuhrer, and T. D. Millstein. Towards concurrency refactoring for x10. In *PPOPP, poster*, pages 303–304, 2009.
[26] S. P. Masticola and B. G. Ryder. Non-concurrency analysis. In *Proceedings of the symposium on Principles and practice of parallel programming*, pages 129–138, New York, NY, USA, 1993. ACM.
[27] G. Naumovich and G. S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *Proceedings of the International symposium on Foundations of software engineering*, pages 24–34, 1998.
[28] G. Naumovich, G. S. Avrunin, and L. A. Clarke. An efficient algorithm for computing mhp information for concurrent java programs. In *Proceedings of the European software engineering conference*, pages 338–354, London, UK, 1999. Springer-Verlag.
[29] Robert H. B. Netzer and Barton P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, 1992.
[30] P.Luszczek, J.Dongarra, and J.Kepner. Design and implementation of the HPCC benchmark suite. *CT Watch Quarterly*, 2(4), November 2006.
[31] L. Ricci. Automatic loop parallelization: An abstract interpretation approach. In *Proceedings of the International Conference on Parallel Computing in Electrical Engineering*. IEEE Computer Society, 2002.
[32] N. S. Sundar, S. Jayanthi, P. Sadayappan, and M. Visbal. An incremental methodology for parallelizing legacy stencil codes on message-passing computers. In *Proceedings of the International Conference on Parallel Processing*, page 302. IEEE Computer Society, 1999.
[33] Richard N. T. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, pages 57–84, 1983.
[34] P. Tu and D. A. Padua. Automatic array privatization. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, pages 500–521. Springer-Verlag, 1994.