May-Happen-in-Parallel Analysis of X10 Programs

Shivali Agarwal
Tata Institute of Fundamental Research, Mumbai, India
shivali@tcs.tifr.res.in

Rajkishore Barik
IBM India Research Lab, New Delhi, India
rajbarik@in.ibm.com

Vivek Sarkar
IBM T.J. Watson Research Center
vsarkar@us.ibm.com

Rudrapatna K Shyamasundar
IBM India Research Lab, New Delhi, India
rshyamas@in.ibm.com

Abstract

X10 is a modern object-oriented programming language designed for high performance, high productivity programming of parallel and multi-core computer systems. Compared to the lower-level thread-based concurrency model in the Java$^{\text{TM}}$ language, X10 has higher-level concurrency constructs such as `async`, `atomic` and `finish` built into the language to simplify creation, analysis and optimization of parallel programs. In this paper, we introduce a new algorithm for May-Happen-in-Parallel (*MHP*) analysis of X10 programs. The analysis algorithm is based on simple path traversals in the Program Structure Tree, and does not rely on pointer alias analysis of thread objects as in *MHP* analysis for Java programs. We introduce a more precise definition of the *MHP* relation than in past work by adding *condition vectors* that identify execution instances for which the *MHP* relation holds, instead of just returning a single true/false value for all pairs of executing instances. Further, *MHP* analysis is refined in our approach by using the observation that two statement instances which occur in atomic sections that execute at the same X10 place must have *MHP* = false. We expect that our *MHP* analysis algorithm will be applicable to any language that adopts the core concepts of places, async, finish, and atomic sections from the X10 programming model. We also believe that this approach offers the best of two worlds to programmers and parallel programming tools — higher-level abstractions of concurrency coupled with simple and efficient analysis algorithms.

Categories and Subject Descriptors D.1.3 [*Programming Technique*]: Concurrent Programming—Parallel Programming

General Terms Languages, Verification

Keywords X10, May-Happen-in-Parallel, Place, Concurrent, Atomic, Activity, Parallel Program Analysis

1. Introduction

The current trends towards multi-core processors is expanding the demand for languages and tools that simplify parallel programming. X10 [4] is a modern object-oriented programming language designed for high performance, high productivity programming of parallel and multi-core computer systems. X10 offers various concurrency control constructs to the programmers: multiple parallel activities can be created using the `async` construct, their termination can be coordinated using the `finish` construct, mutual exclusion can be enforced using `atomic` blocks, and barrier based synchronization among activities can be performed using the `clock` construct. X10 also supports partitioning of data and activities across `places` through the use of *distributions*.

The goal of May-Happen-in-Parallel (*MHP*) analysis is to statically determine if it is possible for execution instances of two given statements (or the same statement) to execute in parallel. *MHP* analysis serves as a key foundation for concurrent static and dynamic debugging tools including tools for data race detection [5, 10, 14, 16]. Past research on *MHP* analysis has been conducted for parallel programming languages ranging from Ada [6, 15, 17] to Java [2, 13, 18]. In general, the problem of precise *MHP* analysis for all pairs of statements in a given program is undecidable. If all control flow paths in all threads are assumed to be executable, Taylor [23] has proved that, under certain assumptions, computation of *MHP* information is an NP-complete problem. Given the complexity of the problem, a scalable and efficient *MHP* analysis algorithm remains a significant challenge in the area of analysis of parallel programs.

In this paper, we focus on computing *MHP* information for concurrent X10 programs. The main contributions of this work compared to past *MHP* analysis algorithms are as follows:

1. We introduce a more precise definition of the *MHP* relation than in past work by adding *condition vectors* that identify execution instances for which the *MHP* relation holds, instead of just returning a single true/false value for all pairs of executing instances.
2. Compared to past work, the availability of basic concurrency control constructs in X10 such as `async` and `finish` enable the use of more efficient and precise analysis algorithms based on simple path traversals in the Program Structure Tree, and does not rely on interprocedural pointer alias analysis of thread objects as in *MHP* analysis for the Java language. Note that Taylor's NP-hardness results [23] for *MHP* analysis applies to programs that use lower-level synchronization primitives such

as Ada's rendezvous, and are not applicable to the X10 constructs considered in this paper.

3. Finally, *MHP* analysis is refined in our approach by using the observation that two statement instances which occur in atomic sections that execute at the same X10 place must have *MHP* = false.

2. X10 Overview

This section provides a brief summary of a core subset of v0.41 of the X10 programming language [4]. The goal of X10 is to introduce a core set of new language constructs that address the fundamental requirements for high productivity programming of parallel systems at all scales — multi-core processors, symmetric shared-memory multiprocessors (SMPs), commodity clusters, high end supercomputers, and even embedded processors like Cell. The key features of X10 include:

- Explicit reification of locality in the form of *places*, with support for a *partitioned global address space* (PGAS) across places
- Lightweight *activities* embodied in *async*, *foreach*, and *ateach* constructs which subsume communication and multithreading operations in other languages
- A *finish* construct for termination detection and rooted exception handling of descendant activities
- Support for lock-free synchronization with *atomic* blocks

With a view to mainstream adoption, X10 uses a serial subset of the Java language as its foundation, but replaces the Java language's current support for concurrency by new constructs that are motivated by high-productivity high-performance parallel programming.

The remainder of this section briefly describes the three core constructs of X10 — async, atomic, finish. An important safety result in X10 is that any program written with async, finish, and atomic can never deadlock. Section 2.1 outlines how these three constructs are used to write single-place parallel programs. Section 2.2 then discusses how the async and finish constructs extend to the multi-place case.

2.1 Single-Place Programming in X10 using async, atomic, finish

2.1.1 async ⟨stmt⟩

Async is the X10 construct for creating or forking a new asynchronous activity. The statement, *async* ⟨*stmt*⟩, causes the parent activity to create a new child activity to execute ⟨*stmt*⟩. Execution of the async statement returns immediately i.e., the parent activity can proceed immediately to its next statement.

Consider the following X10 code example of an async construct. The goal of this example is to use two activities to compute in parallel the sums of the odd and even numbers in the range $1 \ldots n$. This is accomplished by having the main program activity use the *async* statement to create a child activity to execute the for-i loop and print oddSum, while the main program activity proceeds in parallel to execute the for-j loop and print evenSum.

```
public static void main(String[] args) {
  final int n = 100;
  async { // Compute oddSum in child activity
    double oddSum = 0;
    for (int i = 1 ; i <= n ; i += 2 ) oddSum += i;
    System.out.println("oddSum = " + oddSum);
  }
  // Compute evenSum in parent activity
  double evenSum = 0;
  for (int j = 2 ; j <= n ; j += 2 ) evenSum += j;
  System.out.println("evenSum = " + evenSum);
} // main()
```

2.1.2 finish ⟨stmt⟩

The X10 statement, *finish* ⟨*stmt*⟩, causes the parent activity to execute ⟨*stmt*⟩ and then wait till all sub-activities created within ⟨*stmt*⟩ have terminated globally. If async is viewed as a fork construct, then finish can be viewed as a join construct restricted to only the activities created within the scope of the finish. X10 distinguishes between *local termination* and *global termination* of a statement. The execution of a statement by an activity is said to terminate locally when the activity has completed all the computation related to that statement. For example, the creation of an asynchronous activity terminates locally when the activity has been created. A statement is said to terminate globally when it has terminated locally and all activities that it may have spawned (if any) have, recursively, terminated globally.

Consider a variant of the previous example in which the main program waits for its child activity to finish so that it can print the result obtained by adding oddSum and EvenSum:

```
public static void main(String[] args) {
  final int n = 100;
  final BoxedDouble oddSum = new BoxedDouble();
  double evenSum = 0;
  finish {
    async { // Compute oddSum in child activity
      for (int i = 1 ; i <= n ; i += 2 )
        oddSum.val += i;
    }
    // Compute evenSum in parent activity
    for (int j = 2 ; j <= n ; j += 2 ) evenSum += j;
  } // finish
  System.out.println("Sum = " + (oddSum.val + evenSum));
} // main()
```

The finish statement guarantees that the child activity terminates globally before the print statement is executed. Note that the result of the child activity is communicated to the parent in a shared object, oddSum, since X10 does not permit a child activity to update a local variable in its parent activity. In this case, the local variable oddSum contains a pointer to an object with a val field, thereby enabling oddSum.val to be updatable even though oddSum is a constant pointer. It is also worth noting that the X10 memory model is weak enough to allow oddSum.val to be allocated to a register during the execution of the entire for-i loop.

2.1.3 atomic ⟨stmt⟩

The atomic construct in X10 is used to coordinate accesses by multiple activities to shared data located at the same place. The X10 statement, *atomic* ⟨*stmt*⟩, causes ⟨stmt⟩ to be executed atomically i.e., its execution occurs as if in a single step during which ⟨stmt⟩ executes and terminates locally while all other concurrent activities in the same place are suspended. Compared to user-managed locking, the X10 user only needs to specify that a collection of statements should execute atomically and leaves the responsibility of lock management and alternative mechanisms for enforcing atomicity to the language implementation. Commutative operations, such as updates to histogram tables and insertions in a shared data structure, are a natural fit for atomic blocks when performed by multiple activities. An atomic block may include method calls, conditionals, and other forms of sequential control flow. For scalability reasons, blocking operations like finish and force are not permitted in an atomic block. An async statement is not permitted either because the atomicity guarantee would only apply to local (not global) termination of the child async activity.

2.2 Multi-Place Programming in X10

Current programming models use two separate levels of abstraction for shared-memory thread-level parallelism (e.g., Java threads, OpenMP, pthreads) and distributed-memory communication (e.g., Java messaging, RMI, MPI, UPC) resulting in significant complexity when trying to combine the two. In this section, we show how the three core X10 constructs introduced earlier can be extended to multiple places. A *place* is a collection of resident (non-migrating) mutable data objects and the activities that operate on the data. Every X10 activity runs in a place; the activity may obtain a reference to this place by evaluating the constant *here*.

X10 v0.41 takes the conservative decision that the number of places is fixed at the time an X10 program is launched. Thus, there is no construct to create new places. This is consistent with current programming models, such as MPI, UPC, and OpenMP, that require the number of processes to be specified when an application is launched. This design decision may be revisited in future versions of the language as more experience is gained with adaptive computations which may naturally require a hierarchical, dynamically varying notion of places.

Places are virtual — the mapping of places to physical locations is performed by a deployment step that is separate from the X10 program. Though objects and activities do not migrate across places in an X10 program, an X10 deployment is free to migrate places across physical locations based on affinity and load balance considerations. While an activity executes at the same place throughout its lifetime, it may dynamically spawn activities in remote places.

X10 supports a partitioned global address space (PGAS) that is partitioned across places. Each mutable location and each activity is associated with exactly one place. A scalar object in X10 is allocated completely at a single place. In contrast, the elements of an array, may be distributed across multiple places. We now discuss how the async and finish constructs discussed earlier in a single-place context, extend directly to the multi-place case. A key constraint for atomic constructs is that an atomic block is only permitted to access place-local data (Locality Rule).

The statement, *async* (⟨*place-expr*⟩) ⟨*stmt*⟩, causes the parent activity to create a new child activity to execute ⟨*stmt*⟩ at the place designated by ⟨*place-expr*⟩. The async is local if the destination place is same as the place where the parent is executing, and remote if the destination is different. Local async's are like lightweight threads, as discussed earlier in the single-place scenario. A remote async can be viewed as an active message, since it involves communication of input values as well as remote execution of the computation specified by ⟨*stmt*⟩. The semantics of the X10 *finish* operator is identical for local and remote async's *i.e.,* a finish ensures global termination of all asyncs (local and remote) created in the scope of the finish.

3. Program Structure Tree and MHP Analysis of X10 Programs

The May-Happen-In-Parallel analysis problem addressed in this paper deals with analysis of multi-place X10 programs that use *async*, *finish*, and *atomic* constructs, which are higher-level forms of the thread-based *start*, *join*, and *synchronized* constructs in the Java language. In this section, we introduce the Program Structure Tree (*PST*) representation for X10 procedures, which will be used in later sections as the foundation for performing *MHP* analysis:

DEFINITION 3.1. *A Program Structure Tree PST(N,E) for a procedure is a rooted tree where*

1. *N is a set of nodes such that each node $n \in N$ has one of the following types:* `root`, `statement`, `loop`, `async`, `finish`, `atomic`. *The* `root` *node designates the start of the procedure. Each* `async` *node is annotated with a* place expression *that designates the* X10 *place executing the* `async`.
2. *E is set of tree edges obtained by collapsing the abstract syntax tree representation of the procedure into the six node types listed above. PST.$\mathit{parent}(N)$ returns the parent of node N as defined by E.*

The X10 *language semantics ensures that an* `atomic` *node will not be an ancestor of* `finish` *or* `async` *node. In addition, all* `statement` *nodes must be leaf nodes in the PST.* □

Figure 1 contains a simple example of an X10 code fragment and its *PST*.

Having obtained the *PST* from the abstract syntax tree, the high level steps involved in *MHP* analysis for X10 programs are outlined below:

1. First, Never-Execute-In-Parallel (*NEP*) analysis is performed as described in Section 4. This analysis considers the occurrences of `finish` and `async` nodes in the *PST* and determines statements that can never execute in parallel. For soundness, the *NEP* analysis conservatively errs on the side of returning *NEP* = *false* when it is unable to perform a precise analysis of the input X10 program. In the case of `loop` nodes in the *PST*, we use *condition vectors* (defined in Section 4) to qualify the instances of execution that can never happen in parallel. Note that X10's `foreach` and `ateach` constructs for parallel loops can be represented in the *PST* by an equivalent pair of `loop` and `async` nodes.

2. Next, a Place-Equivalence (*PE*) analysis is performed as described in Section 5. The output of this analysis is a predicate, *PE*($S1$,$S2$), which is set to *true* if all instances of $S1$ and $S2$ are guaranteed to execute at the same place. For soundness, the *PE* analysis conservatively errs on the side of returning *PE* = *false* when it is unable to perform a precise analysis of the input X10 program. Similar to *NEP* analysis, we use *condition vectors* to qualify the instances of execution of statements that are place equivalent.

3. In the final step of *MHP* analysis as defined in Section 6, we combine the *NEP* and *PE* analysis to obtain the *MHP* information for `atomic` constructs. The basic intuition is that for all those instances of execution of pair of statements where *NEP* is *true*, *MHP* is assigned *false*. In addition, if the statements are executed atomically, then *MHP* is assigned *false* for all those instances of execution which happen at same place.

4. Never-Execute-in-Parallel Analysis

In this section, we describe our approach for determining if two statements will *never execute in parallel* (*NEP*). The *NEP* relation is the complement of the *May-Happen-in-Parallel* (*MHP*) relation that has been introduced in past work for Java and other concurrent programming languages [2, 18]. *NEP* is used instead of *MHP* in this section for the sake of convenience in presentation. In addition, the *NEP* relation will be used to compute a refined *MHP* relation later in Section 6.

The significant differences between the *NEP* analysis presented in this paper and past work on *MHP* analysis are as follows:

1. The availability of basic concurrency control constructs in X10 such as `async` and `finish` enables a more efficient and precise *NEP* analysis algorithm compared to past work on *MHP* analysis for Java. Our algorithm is based on simple path traversals in the *PST*.

2. Past work on *MHP* analysis resulted in a simple true/false value for a given pair of statements. Our work makes the *NEP*

```
         for ( i = 1 ; i <= n ; i++ )

            finish

               for ( j = 1 ; j <= n ; j++ )

                  for ( k = 1 ; k <= n ; k++ )

                     async (A.distribution[i,j,k])

                        atomic  {

S1:                        temp  = f(A[i,j,k]);

S2:                        A[i,j,k] = temp;
                        }
```

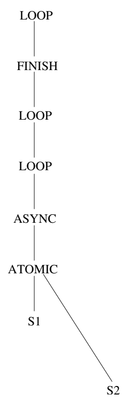


Figure 1. Example X10 Program and its *PST*

relation more precise by adding *condition vectors* that are able to identify execution instances for which the *NEP* relations hold.

3. As discussed later in Section 6, we show how the *NEP* information can be further refined by using the atomicity properties of atomic sections in X10.

DEFINITION 4.1. *Two statements S_1 and S_2 are said to* never execute in parallel, *written as NEP(S_1,S_2) = true, with* condition vector set *CS if the following conditions hold:*

1. *S_1 and S_2 have exactly k loop nodes, $L_1, \dots L_k$ as common ancestors in the PST (where $k \geq 0$).*
2. *Each element $\langle C_1, \dots C_k \rangle$ in CS is a vector of k functions of type* `int` $\times$ `int` $\rightarrow$ `boolean`*. In this paper, we will restrict our attention to three possible functions — "=", "$\neq$", and "$*$". The symbol $*$ denotes the function that returns true for all inputs*[1].
3. *Let $S_1[i_1, \dots i_k]$ denote any execution instance of S_1 in iteration $i_1, \dots i_k$ of loops $L_1, \dots, L_k$, and likewise for $S_2[j_1, \dots j_k]$. If $C_x(i_x, j_x)$ =* true *$\forall 1 \leq x \leq k$ for some condition vector $\langle C_1, \dots C_k \rangle$ in CS, then it is guaranteed that statement instances $S_1[i_1, \dots i_k]$ and $S_2[j_1, \dots j_k]$ cannot execute in parallel.* □

To summarize Definition 4.1, if *NEP*(S_1,S_2) = *false* then there are no pairs of instances of S_1 and S_2 that can be guaranteed to not execute in parallel. If *NEP*(S_1,S_2) = *true* then the instances of S_1 and S_2 that can be guaranteed to not execute in parallel are determined by the condition vectors in *CS*.

The algorithm for computing the *NEP* relation is given in Figure 2. The algorithm takes two inputs: PST for the X10 procedure being analyzed, and two statements, S_1 and S_2, for which we want to compute whether *NEP* is *true* or *false*. Note that the algorithm also accepts the case where $S_1 = S_2$. The first step is to find the least common ancestor of the two statements, denoted by A = *LCA*(S_1,S_2). This gives us the common scope of execution of the two statements. In Steps 2 and 3, it is established for S_1 and S_2 respectively whether they execute within an "unfinished" `async` created within A. Depending on this information, there are 4 cases that arise for *NEP* analysis for the ⟨"=", . . . , "="⟩ condition vector, as described in Steps 5d - 5g:

- *Case 1 (Step 5d)*: If both S_1 and S_2 do not execute in an `async` construct under A then we can conclude they will never execute in parallel.
- *Case 2 and 3 (Steps 5e and 5f)*: If exactly one of S_1 or S_2 executes in an `async` scope, then the dominator relation can be used to compute the *NEP*(S_1,S_2) relation. In the algorithm, the dominator relation is checked on ancestors of S_1 and S_2 (AS_1 and AS_2 respectively) that are immediate children of *LCA*(S_1, S_2). If the *PST* path from S_1 upto *LCA*(S_1, S_2) contains an `async` node which is not followed by any `finish` node and AS_1 dominates AS_2, then S_1 and S_2 will never execute in parallel.
- *Case 4 (Step 5g)*: If both S_1 and S_2 execute in a `async` scope, then we have to conservatively assume that *NEP* = *false*.

Finally, we come to Step 6 which is performed in the case when S_1 and S_2 have $k \geq 1$ common loops. This step examines all nodes in the *PST* starting from A, the least common ancestor of S_1 and S_2, and ending at L_1, the outermost common loop that encloses S_1 and S_2. Note that the algorithm uses the fact whether a loop contains a `finish` or `async` node in the *PST* to restrict the set of iterations for which *NEP* = *true*. If (say) loop L_x contains a `finish` node that is an ancestor of both S_1 and S_2 statements and there is no intervening `async` node in *PST* path from the `finish` node to L_x, we observe that instances of S_1 and S_2 from two distinct iterations of L_x (but created in the same iteration of outer loops $L_1, \dots, L_{x-1}$) can never execute in parallel. This property is captured by a *condition vector* in which C_x is set to "$\neq$", $C_1, \dots, C_{x-1}$ are set to "=", and $C_{x+1}, \dots, C_k$ are set to "$*$".

The algorithm in Figure 2 assumes that the *PST* has already been constructed, which is a one-time $O(N)$ cost. In addition, Steps 5e and 5f use the *dominator* relation on the original control flow graph, which can be computed using algorithms that vary in execution time complexity from $O(NlogN)$ [12] to $O(N)$ [8] as a one-time cost. We observe that the *NEP* algorithm takes $O(H)$ time for determining if a given pair of nodes, S_1 and S_2, satisfy *NEP*(S_1,S_2) = *true*, where H is the *height* of the *PST*. Note that the condition vector set *CS*, can contain at most $L + 1$ condition vectors — one contributed by Step 5h and L by Step 6(b)iiiB – each of which has $O(L)$ size, where $L \leq H$ is the maximum nesting of *loops* in the PST. Step 6(b)iii can be considered a constant time operation. If used to compute the *NEP* relation for all pairs of statements, the total execution time will be $O(N^2H)$, which is more efficient than the $O(N^3)$ time of the *MHP* algorithm in [18]. However, we

[1] These three operators have been also used in past work on *direction vectors* [24]. We may choose to extend C_x in the future to represent *distance vectors* or other more general boolean functions.

Inputs:

1. A Program Structure Tree (*PST*) for the procedure being analyzed
2. Two statement nodes S_1 and S_2 in the *PST* with $k \geq 0$ common loop node ancestors in the *PST*, $L_1, \ldots, L_k$.

Outputs:

1. *NEP*(S_1, S_2), a boolean value that indicates if instances of S_1 and S_2 never execute in parallel.
2. *CS*, a set of condition vectors that is used only if *NEP*(S_1, S_2) = true. Given statement instances $S_1[i_1, \ldots i_k]$ and $S_2[j_1, \ldots j_k]$, if $C_x(i_x, j_x)$ = *true* $\forall 1 \leq x \leq k$ for some condition vector $\langle C_1, \ldots C_k \rangle$ in *CS* then it is guaranteed that the two statement instances cannot execute in parallel.

Algorithm:

```
1. $A :=$ LCA$(S_1, S_2)$, the Lowest Common Ancestor of $S_1$ and $S_2$ in the PST
2. /* Determine if an instance of $S_1$ can be executed in a new async activity that escapes a given execution instance of $A$ */
   async_$S_1$ := false
   for ( $N := S_1$ ; $N \neq A$ ; $N :=$PST.$parent(N)$ ) do
   (a) if $N$ is an async node then async_$S_1$ := true end if
   (b) if $N$ is a finish node then async_$S_1$ := false end if
   end for
3. /* Repeat the previous step for $S_2$ */
   async_$S_2$ := false
   for ( $N := S_2$ ; $N \neq A$ ; $N :=$PST.$parent(N)$ ) do
   (a) if $N$ is an async node then async_$S_2$ := true end if
   (b) if $N$ is a finish node then async_$S_2$ := false end if
   end for
4. CS$:= \emptyset$ /* Initialize CS to an empty set */
5. if ( $S_1 \neq S_2$ ) then
   /* Analyze four cases for async_$S_1$ and async_$S_2$ */
   (a) $AS_1 :=$ PST ancestor of $S_1$ that is a child of $A$; Note that $AS_1 := S_1$ if $S_1$ is a child of $A$;
   (b) $AS_2 :=$ PST ancestor of $S_2$ that is a child of $A$; Note that $AS_2 := S_2$ if $S_2$ is a child of $A$;
   (c) $flag := false$ /* $\langle =, \ldots, = \rangle$ will be added to CS if $flag$ = true */
   (d) if ( $\neg$ async_$S_1$ $\wedge$ $\neg$ async_$S_2$ ) then $flag := true$ end if /* Case 1 */
   (e) if ( $\neg$ async_$S_1$ $\wedge$ async_$S_2$ ) then $flag$ := ($AS_2$ does not dominate $AS_1$) end if /* Case 2 */
   (f) if ( async_$S_1$ $\wedge$ $\neg$ async_$S_2$ ) then $flag$ := ($AS_1$ does not dominate $AS_2$) end if /* Case 3 */
   (g) if ( async_$S_1$ $\wedge$ async_$S_2$ ) then $flag$ := false end if /* Case 4 */
   (h) if ( flag ) then CS:= CS$\cup\{ \langle$"=", ..., "="$\rangle \}$ end if
   end if
6. if ( $k \geq 1$ ) then /* $S_1$ and $S_2$ have at least one common loop */
   (a) $seqloop$ := true ; $x := k + 1$
   (b) for ( $N := A$ ; $N \neq L_1$ ; $N :=$PST.$parent(N)$ ) do
       i. if $N$ is an async node then $seqloop$ := false end if
       ii. if $N$ is a finish node then $seqloop$ := true end if
       iii. if $N$ is a loop node then
            A. $x := x - 1$ ;
            B. if ( $seqloop$ ) then CS:= CS$\cup\{\langle C_1 =$ "=", $\ldots, C_x =$ "$\neq$", $C_{x+1} =$ "*", $\ldots, C_k =$ "*"$\rangle\}$ end if
          end if
       end for
   end if
7. NEP$(S_1, S_2)$ := ( CS$\neq \emptyset$ ) /* Return NEP = true if CS is non-empty */
```

Figure 2. Algorithm for computing Never-Execute-in-Parallel (*NEP*) relations

```
Main thread:
------------
S1: ExternalHelper1.start();
S2: ...
S3: ExternalHelper1.join();
S4: ... // MHP algorithm concludes that
        // S11 and S12 may happen in parallel with S4

ExternalHelper1 thread:
-----------------------
S5: ...
S6: InternalHelper1_1.start();
S7: InternalHelper1_2.start();

S8: InternalHelper1_1.join();
S9: InternalHelper1_2.join();
S10: ... // MHP algorithm concludes that
         // S11 and S12 cannot happen in parallel with S10

InternalHelper1_1 thread:
-------------------------
S11: ...

InternalHelper1_2 thread:
-------------------------
S12: ...
```

Figure 3. Java example program to illustrate *MHP* algorithm

expect that the execution time overhead of the *NEP* algorithm will be much smaller than $O(N^2H)$ in practice, since it can be used in a demand-driven fashion to only compute the *NEP* relation for pairs of statements that are of interest to an interactive tool or compiler transformation.

We will first use the example program in Figure 1 to illustrate the algorithm. The example was intentionally chosen to be as simple as possible to illustrate the core ideas in this paper. In this example, we are interested in determining which pairs of execution instances of statements S_1 and S_2 will never execute in parallel with each other, so the algorithm in Figure 2 will be invoked to compute *NEP*(S_1,S_2). The output of this algorithm will be *NEP*(S_1,S_2) = true, with condition vector set *CS*= $\{\langle =,=,= \rangle, \langle \neq,*,* \rangle\}$. This implies that two instances of S_1 and S_2 are guaranteed to never execute in parallel if they belong to the same `i-j-k` iteration, or if they come from iterations with distinct values of `i`.

Finally, we use the following X10 code fragment

```
{ S1 ; async S2 ; S3 ; async S4 ; }
```

to illustrate the four cases in Step 5 in the *NEP* algorithm as follows:

Case 1 *NEP*(S_1, S_3) = true, in accordance with Step 5d.

Case 2 *NEP*(S_1, S_2) = true, in accordance with Step 5e.

Case 3 *NEP*(S_2, S_3) = false, in accordance with Step 5f.

Case 4 *NEP*(S_2, S_4) = false, in accordance with Step 5g.

4.1 Comparison with *MHP* Analysis of Java programs

In this section, we briefly compare the *NEP* algorithm from Figure 2 with the *MHP* data flow analysis algorithm developed by Naumovich et al [18]. The later algorithm was designed to address all concurrency features in Java threads, including wait/notify/notifyall operations in synchronized blocks. In this comparison, we will restrict our attention to the *MHP* algorithm's handling of the *start*, *join*, and *synchronized* constructs in Java threads, which are comparable, but not equivalent, to *async*, *finish*, and *atomic* in X10.

Figure 3 contains the skeleton of a Java program that represents the parallel control flow in the `SplitRendererNested` example

```
S0: finish {
  S1: async { // ExternalHelperThread1.start()
    finish {
      S5: ...
      S6: async S11   // InternalHelperThread1_1.start()
      S7: async S12   // InternalHelperThread1_2.start()
      ...
    }
    S8: ...  // finish subsumes InternalHelper1_1.join()
    S9: ...  // finish subsumes InternalHelper1_2.join()
    S10: ... // NEP algorithm concludes that
             // NEP(S10,S11) = NEP(S10,S12) = false
  }
  S2: ...
}
S3: ... // S0's finish subsumes ExternalHelperThread1.join()
S4: ... // NEP algorithm concludes that
        // NEP(S4,S11) = NEP(S4,S12) = false
```

Figure 4. X10 example program to illustrate *NEP* algorithm

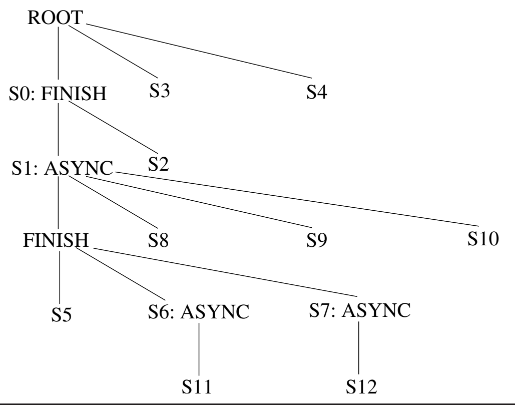


Figure 5. *PST* for example program in Fig 4

used in [18]. As discussed in [18], the *MHP* algorithm is conservative in its analysis of nested parallelism and concludes that $S11$ and $S12$ may happen in parallel with $S4$, even though it is able to conclude that $S11$ and $S12$ cannot happen in parallel with $S10$.

As a comparison, Figure 4 contains the skeleton of an X10 program that is equivalent to the Java program in Figure 3. If the *NEP* algorithm from Figure 2 is invoked to compute *NEP*($S4$,$S11$), it will perform the following steps to conclude that *NEP*($S4$,$S11$) = *true*:

- Step 1: A := *LCA*($S4$,$S11$) = *ROOT*
- Step 2: $async\_S4$:= *false*
- Step 3: $async\_S11$:= *false*
- Step 5(a): $AS4 := S4$
- Step 5(b): $AS11 := S0$
- Step 5(d): *flag* := *true*
- Step 5(h): *CS* := $\{ \langle =, \ldots, = \rangle \}$
- Step 7: *NEP*($S4$,$S11$):= *true*

Thus, the *NEP* algorithm is able to establish that $S11$ and $S12$ cannot happen in parallel with $S4$, while the *MHP* algorithm from Figure 3 conservatively concludes that $S11$ and $S12$ may happen in parallel with $S4$.

The above discussion focused on the conservativeness in analysis of nested parallelism in the *MHP* algorithm. As mentioned earlier, another dimension of conservativeness in *MHP* analysis of Java programs is the necessity to perform interprocedural pointer alias analysis of thread objects to establish accurate parallel control flow relationships among threads. For example, the *MHP* analysis must establish that all thread objects (*e.g.,* `ExternalHelper1`, `InternalHelper1_1`, `InternalHelper1_2`) are distinct before it can even conclude that $S11$ and $S12$ cannot happen in parallel with $S10$ in Figure 3. As observed in past work on static data race detection, interprocedural alias analysis of thread objects can pose a significant challenge in practice. In contrast, the analysis of X10's *async*, *finish*, and *atomic* constructs is simpler because it does not rely on alias analysis of thread objects.

5. Place Equivalence Analysis

In this section, we describe our approach for determining if two statements are *place equivalent* (*PE*) *i.e.,* if they will definitely execute at the same place. Most parallel programming models that are currently used for distributed-memory multiprocessors follow a Single Program Multiple Data (*SPMD*) model in which one thread is executed per place. However, the X10 programming model is more general since it integrates thread-level parallelism and cluster-level parallelism by allowing multiple activities to be created at different *places*. Place equivalence analysis therefore becomes important for more general parallel programming models such as X10.

DEFINITION 5.1. *Two statements S_1 and S_2 are said to be* place equivalent, *written as PE*(S_1, S_2) *= true, with* condition vector set *CS if the following conditions hold:*

1. *S_1 and S_2 have exactly k loop nodes, $L_1, \ldots L_k$ as common ancestors in the PST (where $k \geq 0$)*
2. *Let $S_1[i_1, \ldots i_k]$ denote any execution instance of S_1 in iteration $i_1, \ldots i_k$ of loops $L_1, \ldots, L_k$, and likewise for $S_2[j_1, \ldots j_k]$. If $C_x(i_x, j_x)$ =* true *$\forall 1 \leq x \leq k$ for some condition vector $\langle C_1, \ldots C_k \rangle$ in CS, then it is guaranteed that statement instances $S_1[i_1, \ldots i_k]$ and $S_2[j_1, \ldots j_k]$ must execute at the same place.* □

To summarize Definition 5.1, if *PE*(S_1,S_2) = *false* then there are no pairs of instance of S_1 and S_2 for which place equivalence is guaranteed. If *PE*(S_1,S_2) = *true* then the instances of S_1 and S_2 that can be guaranteed to execute at the same place are determined by the condition vectors in *CS*.

The algorithm for computing the *PE* relation is given in Figure 6. The algorithm needs two additional pre-passes as inputs along with the *PST*. First, a global place-value numbering pre-pass for place expressions to determine place local information for statements. Second, a global loop-invariant analysis (also known as $LoopSet$ analysis) pre-pass to determine loops for which a given place expression is place variant. Global value numbering can be performed using an SSA-based algorithm as in [1]. Let us describe the how $LoopSet$ information is computed.

Consider the following code fragment as an example:

```
for ( i = 1 ; i < = n ; i++)      // L1
  for ( j = 1 ; j < = n ; j++)    // L2
    for ( k = 1 ; k < = n ; k++) // L3
      async (A.distribution[f(i,j),k]) S;
```

To compute $LoopSet$ information for the place expression, `A.distribution[f(i,j),k]`, in the `async` statement in the above code fragment, we need to know the data distribution of array A. In X10 [4], array A can be distributed using a wide range of standard and user-defined distributions such as `UNIQUE`, `RANDOM`, `CYCLIC`, and `BLOCK`. As an example, let us assume that A is distributed in (`BLOCK`, $*$) fashion so that $A[p, *]$ is guaranteed to reside at the same place, where p is the index of the first dimension. The `async` activity in the above code fragment with distribution (`BLOCK`,$*$) will be mapped to different places based on indices i and j, but not k i.e., place-variant with respect to loops $L1$ and $L2$. Hence, $LoopSet(A.distribution[f(i,j),k])$:= $\{L1,L2\}$.

As shown in Step 4a of Figure 6, a pair of statements S_1 and S_2 associated with same global place-value numbers i.e., $V(S_1) = V(S_2)$ are always going to execute at the same place. If $V(S_1) \neq V(S_2)$ and there are no intervening `async` nodes within the inner-most common scope of S_1 and S_2, then these statements are also bound to execute at the same place.

Step 5 traverses the common ancestors (only `loop` and `async` *PST* nodes) to compute *condition vector* using $LoopSet$. For `loops` that are $placeLocalLoops$, the condition vector entries are set to $*$. Note that $LoopSet$ keeps track of the place-variant loops and $placeLocalLoops$ keeps track of place-invariant loops.

The algorithm in Figure 6 assumes that the *PST* is constructed in $O(N)$ time. The pre-passes for the other inputs to the algorithm, Global Value Numbering[2] and $LoopSet$ analysis, can also be computed in linear time. We observe that Step 2 takes $O(H)$ time. The condition vector set *CS*, can at most have two entries – one obtained from Step 4a and another from Step 5c – each of which has $O(L)$ size, where $L \leq H$. For all pairs of statements in the X10 program, the overall complexity of *PE* analysis is bounded by $O(N^2H)$, which is the same complexity as that of *NEP* analysis.

Let us now see how the algorithm works for the example program in Figure 1, assuming that array `A` has a (`BLOCK,BLOCK,*`) distribution. This means that elements $A[i, j, *]$ of array `A` are guaranteed to be mapped to the same place, and the `async` statement in the example will follow the same distribution. Hence, the *PE* algorithm will compute *placeLocalLoops* = $\{L_3\}$, which in turn results in a place condition vector set of *CS*= $\{\langle =,=,= \rangle, \langle =,=,* \rangle\}$. This implies that S_1 and S_2 with same values of `i` and `j` are guaranteed to execute in the same place. Note that, in general, the algorithm in Figure 6 does not require that the number of dimensions in an array reference match the number of loops in the loop nest or that the index ordering for the array access match the ordering of the loop nesting.

6. May-Happen-In-Parallel Analysis using Atomic Sections

In this section, we show how the Never-Execute-in-Parallel (*NEP*) analysis from Section 4 can be combined with the Place-Equivalence (*PE*) information analysis from Section 5 to obtain a more precise May-Happen-in-Parallel (*MHP*) analysis from X10 programs by using atomic sections. The simple approach to computing *MHP* would be to simply invert the *NEP* relation *i.e.,* to return *MHP*(S_1,S_2) = false when *NEP*(S_1,S_2) = true. The key insight leveraged in this section is that two execution instances of statements S_1 and S_2 in an X10 program are guaranteed to not happen in parallel if they both occur in atomic sections that are executed at the same place. This enables us to broaden the number of executions for which we can assert that *MHP* = false. Note that instances of S_1 and S_2 can indeed happen in parallel if they occur in atomic sections that execute at different places.

The algorithm for computing the *MHP* relation is given in Figure 7. For a pair of statements S_1 and S_2, Steps 2 and 3 check if they are nested in `atomic` blocks. In case both S_1 and S_2 are nested in atomic blocks, *MHP*(S_1,S_2) is computed by combining the *NEP* and *PE* results in Step 4. Otherwise, the *MHP* relation is computed directly from the *NEP* relation.

[2] For an SSA-based algorithm such as [1], the complexity is technically linear in the size of the SSA form, which in turn is observed to be linear in the size of the input program in practice.

Inputs:

1. A Program Structure Tree (*PST*) for the procedure being analyzed.
2. Two statement nodes S_1 and S_2 in the *PST* with $k \geq 0$ common loop node ancestors in the *PST*, $L_1, \ldots, L_k$.
3. A *value number* $V(e)$, for each place expression e that is the target of an `async` (e) statement. For convenience, we also assume the availability of $V(N)$ for each `async` node N in the *PST*, where $V(N)$ denotes the value of $here$ for the activity executing S. $V(e)$ and $V(S)$ can be computed by a global value numbering analysis [1] pre-pass on place expressions.
4. For each place expression e, $LoopSet(e)$ = subset of loops $\{L_1, \ldots, L_m\}$ for which the value of place expression e is *place-variant*, where $L_1, \ldots, L_m$ are the loops surrounding expression e (counting from outer to inner). $LoopSet(e)$ can be computed by a global loop-invariant analysis pre-pass.

Outputs:

1. $PE(S_1, S_2)$, a boolean value that indicates if instances of S_1 and S_2 must execute at the same place.
2. *CS*, a set of condition vectors that is used only if $PE(S_1, S_2)$ = true. Given statement instances $S_1[i_1, \ldots i_k]$ and $S_2[j_1, \ldots j_k]$, if $C_x(i_x, j_x)$ = *true* $\forall 1 \leq x \leq k$ for some condition vector $\langle C_1, \ldots C_k \rangle$ in *CS* then it is guaranteed that the two statement instances must execute at the same place.

Algorithm:

```
1. $A := LCA(S_1, S_2)$, the Lowest Common Ancestor of $S_1$ and $S_2$ in the PST
2. Compute async_$S_1$, and async_$S_2$ as in Figure 2
3. $CS := \emptyset$ /* Initialize CS to an empty set */
4. if ( $S_1 \neq S_2$ ) then
   (a) if ( $V(S_1) = V(S_2)$ ) then
        i. $CS := CS \cup \{ \langle *, \ldots, * \rangle \}$ /* $S_1$ and $S_2$ always execute at the same place */
       ii. $PE(S_1, S_2) :=$ true
      iii. return
       else if ( $\neg$ async_$S_1$ $\wedge$ $\neg$ async_$S_2$ ) then
       $CS := CS \cup \{ \langle =, \ldots, = \rangle \}$ /* Instances of $S_1$ and $S_2$ that come from the same iteration of $L_1, \ldots, L_k$ must execute in the same activity and hence at the same place. */
       end if
   end if
5. if ( $k \geq 1 \wedge \neg$ async_$S_1$ $\wedge \neg$ async_$S_2$ ) then /* $S_1$ and $S_2$ have at least one common loop */
   (a) $placeLocalLoops := \{L_1, \ldots, L_k\}$ ; $x := k + 1$
   (b) for ( $N := A$ ; $N \neq L_1$ ; $N :=PST.parent(N)$ ) do
        i. if $N$ is an async node with destination place expression $e$ then
           $placeLocalLoops := placeLocalLoops$ - $LoopSet(e)$ end if
       ii. if $N$ is a loop node then
           A. $x := x - 1$ ;
           B. if ( $L_x \in placeLocalLoops$ ) then $C_x$ := "$*$" else $C_x$ := "=" end if
           end if
       end for
   (c) if ( $placeLocalLoops \neq \emptyset$ ) then $CS := CS \cup \{ \langle C_1, \ldots, C_k \rangle \}$ end if
   end if
6. $PE(S_1, S_2)$ := ( $CS \neq \emptyset$ ) /* Return PE = true if CS is non-empty */
```

Figure 6. Algorithm for computing Place Equivalence (*PE*) relations

Inputs:

1. A Program Structure Tree (*PST*) for the procedure being analyzed
2. Two statement nodes S_1 and S_2 in the *PST* with $k \geq 0$ common loop node ancestors in the *PST*, $L_1, \ldots, L_k$.

Outputs:

1. *MHP*(S_1, S_2), a boolean value that indicates if instances of S_1 and S_2 may happen in parallel.
2. *CS*, a set of condition vectors that is used only if *MHP*(S_1, S_2) = false. Given statement instances $S_1[i_1, \ldots i_k]$ and $S_2[j_1, \ldots j_k]$, if $C_x(i_x, j_x)$ = *true* $\forall 1 \leq x \leq k$ for some condition vector $\langle C_1, \ldots C_k \rangle$ in *CS* then it is guaranteed that the two statement instances cannot happen in parallel.

Algorithm:

```
1. Compute NEP$(S_1, S_2)$ and its associated condition vectors, $CS_{NEP}$ using the NEP algorithm in Figure 2
2. Set atomic_$S_1$ := true if $S_1$ has an atomic node as an ancestor in the PST
3. Set atomic_$S_2$ := true if $S_2$ has an atomic node as an ancestor in the PST
4. if ( atomic_$S_1$ $\wedge$ atomic_$S_2$ ) then
   /* Combine NEP and PE analysis results */
   (a) Compute PE$(S_1, S_2)$ and its associated condition vectors, $CS_{PE}$ using the PE algorithm in Figure 6
   (b) MHP$(S_1, S_2)$ := $\neg$ ( NEP$(S_1, S_2)$ $\vee$ PE$(S_1, S_2)$ )
   (c) CS := $CS_{NEP} \cup CS_{PE}$
   else
   /* Just return NEP analysis results */
   (a) MHP$(S_1, S_2)$ := $\neg$ NEP$(S_1, S_2)$
   (b) CS := $CS_{NEP}$
   end if
```

Figure 7. Algorithm for computing May-Happen-in-Parallel (*MHP*) relation using place equivalence and atomic sections in X10

The complexity of computing the *MHP* relation is bounded by $O(N^2H)$. This is due to the complexity of computing both *NEP* relation and *PE* relation.

As discussed earlier, the *NEP* solution computed using the analysis described in Section 4 for the example in Figure 1 was *NEP*(S_1,S_2) = *true* with condition vector set $CS_{NEP} = \{\langle =,=,= \rangle, \langle \neq, *, * \rangle\}$. This indicates that for different values of loop index i, S_1 and S_2 can not execute in parallel. However, this information can be refined using the *PE* solution due to the presence of `atomic` *PST* node in the loop body. The *PE* solution computed using the analysis described in Section 5 was *PE*(S_1,S_2) = *true* with condition vector set $CS_{PE} = \{\langle =,=,= \rangle, \langle =,=,* \rangle\}$. This indicates that S_1 and S_2 will execute at the same X10 place for different values of loop index k but with same values for loop indices i and j. Using the above two results, the algorithm in this section is able to determine that *MHP*(S_1,S_2) = *false* with condition vector set *CS* = $\{\langle =,=,= \rangle, \langle \neq, *, * \rangle, \langle =,=,* \rangle\}$. i.e., *MHP*($S_1$,$S_2$) = *false* if they belong to the same *i-j-k* iteration, or if they come from iterations with distinct values of i or with the same values of i and j.

7. Related Work

Several approaches for computing May Happen in Parallel (*MHP*) information for programs have been suggested in the past. Callahan and Sublok [3] proposed a data flow algorithm that computes, for each statement in a concurrent program with post-wait synchronization, the set of statements that must be executed before this statement can be executed (*B4* analysis). Deusterwald and Soffa [6] applied *B4* analysis approach to the Ada rendezvous model and extended *B4* analysis to be interprocedural. Masticola and Ryder [15] proposed an iterative approach of non-concurrent analysis that computes a conservative estimate of the set of pairs of communication statements that can never happen in parallel in a concurrent Ada program (the complement of this set is a conservative approximation of the set of pairs that may occur in parallel). In that work, it is assumed initially that any statement from a given process can happen in parallel with any statement in any other process. This pessimistic estimate is then improved by a series of refinements that are applied iteratively until a fixed point is reached. Naumovich and Avrunin [17] proposed a data flow algorithm for computing the *MHP* information for programs with a rendezvous model of concurrency. Thereafter, Naumovich, Avrunin and Clarke [18] proposed an algorithm for computing *MHP* information for concurrent Java programs. Their algorithm uses a data flow framework to compute a conservative estimate of *MHP* information and is shown to be more efficient than reachability analysis based algorithms that determines 'ideal' static *MHP* information. A practical implementation of the algorithm from [18] is described in [13]. Barik [2] proposed an alternative algorithm to compute *MHP* information for concurrent Java programs based on the thread creation tree. This algorithm works with abstract threads rather than concrete thread instance of threads and is shown to be more efficient than the algorithm in [18].

Program Dependence Graphs (PDGs) [7] have been used to analyze the intrinsic parallelism in a sequential program, but are not designed to represent explicitly parallel programs. Parallel Program Graphs (PPGs) [20, 21, 22] are a generalization of PDGs and CFGs that can be used as a foundation for analyzing flow-sensitive properties of parallel programs that go beyond the PST-based analysis presented in this papers. Srinvasan et al. [9] proposed a Parallel Flow Graph (PFG) for optimizing explicitly parallel programs. They provided data flow equations for the reaching definitions analysis and used a copy-in/copy-out semantics for accessing shared variables in parallel constructs. Concurrent Control Flow Graphs

(CCFGs) [11] are similar to PPGs and PFGs, with the addition of conflict edges in addition to synchronization and control flow edges that can be used for analysis of programs with a sequentially consistent memory model.

8. Conclusions and Future Work

In this paper, we introduced a new algorithm for May-Happen-in-Parallel (*MHP*) analysis that is applicable to any language that adopts the core concepts of places, async, finish, and atomic sections from the X10 programming model. The main contributions of this work compared to past *MHP* analysis algorithms are as follows:

1. We introduced a more precise definition of the *MHP* relation than in past work by adding *condition vectors* that identify execution instances for which the *MHP* relation holds, instead of just returning a single true/false value for all pairs of executing instances.
2. Compared to past work, the availability of basic concurrency control constructs such as `async` and `finish` enabled the use of more efficient and precise analysis algorithms based on simple path traversals in the Program Structure Tree, and did not rely on interprocedural pointer alias analysis of thread objects as in *MHP* analysis for the Java language.
3. We introduced place equivalence (*PE*) analysis to identify execution instances that happen at the same place. The *PE* analysis helps us in leveraging the fact that two statement instances which occur in atomic sections that execute at the same X10 place must have *MHP* = false.

For future work, we plan to implement this *MHP* algorithm in an Eclipse-based X10 Development Toolkit (X10DT) being developed at IBM and included in the X10 release on SourceForge [19], so as to be able to respond to interactive *MHP* queries. As motivation, the X10DT screen shot in Figure 8 shows how closely the *outline view* in X10DT mirrors the Program Structure Tree. We will also investigate how the algorithms in this paper can be extended to support additional concurrency constructs in X10, notably clocks and futures. Finally, we will explore how the algorithms can be enriched using distance vectors and can be applied in an interprocedural context.

Acknowledgments

X10 is being developed in the context of the IBM PERCS (Productive Easy-to-use Reliable Computing Systems) project, which is supported in part by DARPA under contract No. NBCH30390004. We are grateful to the following people for their contributions to the X10 project which motivated this work: Philippe Charles, Christopher Donawa, Kemal Ebcioglu, Robert Fuhrer, Christian Grothoff, Allan Kielstra, Nathaniel Nystrom, Igor Peshansky, Christoph von Praun, Vijay Saraswat, Tong Wen.

References

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11, New York, NY, USA, 1988. ACM Press.

[2] Rajkishore Barik. Efficient computation of may-happen-in-parallel information for concurrent java programs. In *18th International Workshop on Languages and Compilers for Parallel Computing*, October 2005.

[3] David Callahan and Jaspal Sublok. Static analysis of low-level synchronization. In *PADD '88: Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, pages 100–111, New York, NY, USA, 1988. ACM Press.

[4] Philippe Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Christoph von Praun, Vijay Saraswat, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA 2005 Onward! Track*, 2005.

[5] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 258–269, New York, NY, USA, 2002. ACM Press.

[6] Evelyn Duesterwald and Mary Lou Soffa. Concurrency analysis in the presence of procedures using a data-flow framework. In *TAV4: Proceedings of the symposium on Testing, analysis, and verification*, pages 36–48, New York, NY, USA, 1991. ACM Press.

[7] J. Ferrante, K. Ottenstein, and J. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[8] Dov Harel. A Linear Time Algorithm for Finding Dominators in Flow Graphs and Related Problems. *Symposium on Theory of Computing*, May 1985.

[9] Ferrante J, Grunwald D, and Srinivasan H. Compile-time analysis and optimization of explicitly parallel programs. In *Journal of Parallel algorithms and applications*, 1997.

[10] Jens Krinke. Static slicing of threaded programs. In *Workshop on Program Analysis For Software Tools and Engineering*, pages 35–42, 1998.

[11] Jaejin Lee. *Compilation Techniques for Explicitly Parallel Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

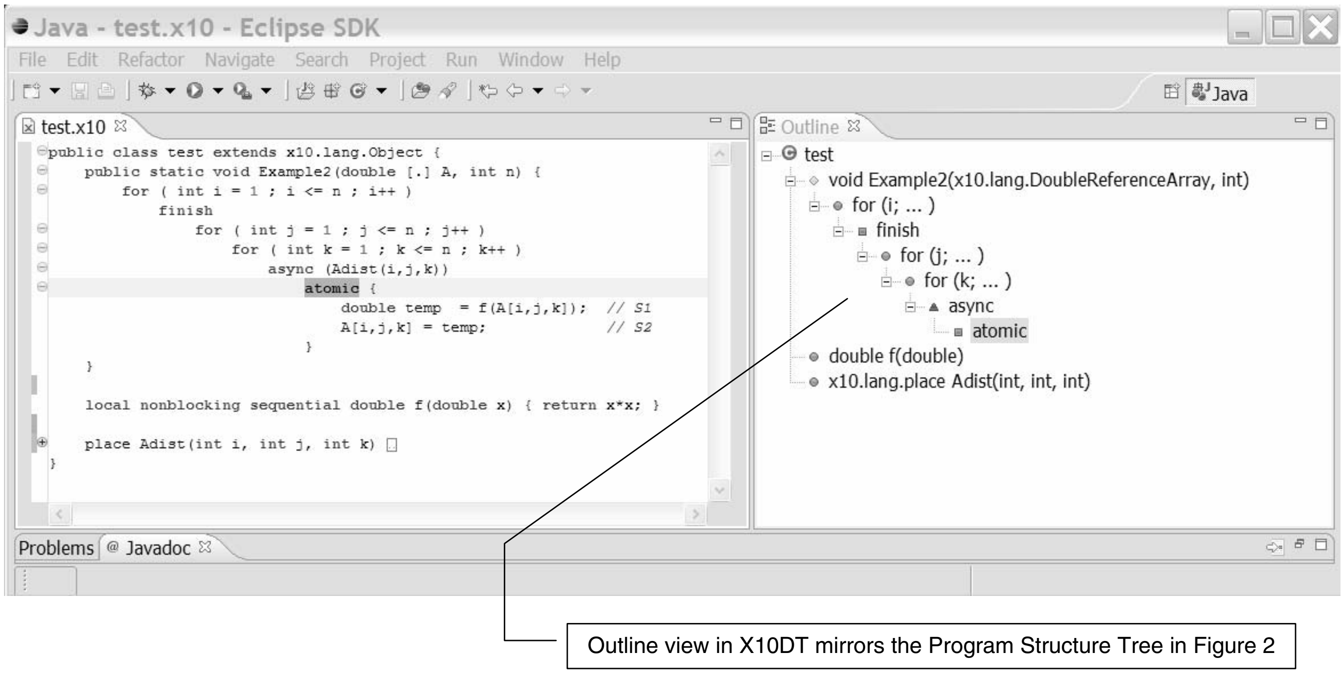
[12] T. Lengauer and Robert Tarjan. A Fast Algorithm for Finding Dominators in a Flowgraph. *TOPLAS*, July 1979.

[13] Lin Li and Clark Verbrugge. A practical mhp information analysis for concurrent java programs. In *The 17th International Workshop on Languages and Compilers for Parallel Computing (LCPC'04)*, 2004.

[14] Stephen P. Masticola and Barbara G. Ryder. A model of ada programs for static deadlock detection in polynomial times. In *PADD '91: Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, pages 97–107, New York, NY, USA, 1991. ACM Press.

[15] Stephen P. Masticola and Barbara G. Ryder. Non-concurrency analysis. In *PPOPP '93: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 129–138, New York, NY, USA, 1993. ACM Press.

[16] G. Naumovich, G. S. Avruin, and L. A. Clarke. Data flow analysis for checking properties of concurrent java programs. Technical Report UM-CS-1998-022, 1998.


Java - test.x10 - Eclipse SDK
File Edit Refactor Navigate Search Project Run Window Help
Java
test.x10
Outline
Problems
Javadoc

```
public class test extends x10.lang.Object {
    public static void Example2(double [.] A, int n) {
        for ( int i = 1 ; i <= n ; i++ )
            finish
                for ( int j = 1 ; j <= n ; j++ )
                    for ( int k = 1 ; k <= n ; k++ )
                        async (Adist(i,j,k))
                            atomic {
                                double temp  = f(A[i,j,k]);  // S1
                                A[i,j,k] = temp;             // S2
                            }
    }

    local nonblocking sequential double f(double x) { return x*x; }

    place Adist(int i, int j, int k) 
}
```

```
test
  void Example2(x10.lang.DoubleReferenceArray, int)
    for (i; ... )
      finish
        for (j; ... )
          for (k; ... )
            async
              atomic
  double f(double)
  x10.lang.place Adist(int, int, int)
```

Outline view in X10DT mirrors the Program Structure Tree in Figure 2

Figure 8. Sample X10DT screenshot for example from Figure 1

[17] Gleb Naumovich and George S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 24–34, New York, NY, USA, 1998. ACM Press.

[18] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. An efficient algorithm for computing MHP information for concurrent Java programs. In *Proceedings of the joint 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 338–354, September 1999.

[19] X10 release on SourceForge. http://x10.sf.net.

[20] Vivek Sarkar. A Concurrent Execution Semantics for Parallel Program Graphs and Program Dependence Graphs (Extended Abstract). *Springer-Verlag Lecture Notes in Computer Science*, 757:16–30, 1992. Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing, Yale University, August 1992.

[21] Vivek Sarkar. Analysis and Optimization of Explicitly Parallel Programs using the Parallel Program Graph Representation. In *Languages and compilers for parallel computing. Proceedings of the 10th international workshop. Held Aug., 1997 in Minneapolis, MN.*, Lecture Notes in Computer Science. Springer-Verlag, New York, 1998.

[22] Vivek Sarkar and Barbara Simons. Parallel Program Graphs and their Classification. *Springer-Verlag Lecture Notes in Computer Science*, 768:633–655, 1993. Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing, Portland, Oregon, August 1993.

[23] Richard N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Inf.*, 19:57–84, 1983.

[24] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers*. Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989. In the series, Research Monographs in Parallel and Distributed Computing.