# Improved MHP Analyses

Aravind Sankar
Dept. of CSE, IIT Madras, India
aravindsankar28@gmail.com

Soham Chakraborty
MPI-SWS, Germany
sohachak@mpi-sws.org

V. Krishna Nandivada
Dept. of CSE, IIT Madras, India
nvk@iitm.ac.in

## Abstract

May-Happen-in-Parallel (MHP) analysis is becoming the backbone of many of the parallel analyses and optimizations. In this paper, we present new approaches to do MHP analysis for X10-like languages that support async-finish-atomic parallelism. We present a fast incremental MHP algorithm to derive all the statements that may run in parallel with a given statement. We also extend the MHP algorithm of Agarwal et al. (answers if two given X10 statements may run in parallel, and under what condition) to improve the computational complexity, without compromising on the precision.

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

***Keywords*** MHP Analysis, Incremental Analysis

## 1. Introduction

May-Happen-in-Parallel (MHP) analysis helps determine if the execution instances of two given statements (or the same statement) may execute in parallel. MHP analysis works as a basis for many static and dynamic program optimizations and program analysis techniques [2, 8, 13, 17, 20, 21]. Naturally, the speed of the MHP analysis plays a key role in the speed and effectiveness of these dependent optimizations and analyses.

In this paper, we present new approaches to efficiently perform MHP analysis, for programs written in task parallel languages, such as X10 [24] and HJ [11] that support `async-finish-atomic` parallelism. The `async` statement is used to create lightweight tasks; `finish` is used as a task join/termination construct; and `atomic` is used to realize mutual exclusion.

Typically, the MHP analysis is used to answer one of the following key questions.

KEY QUESTION 1. *Given a statement $s_1$ in a program $P$, which statements in $P$ may run in parallel with $s_1$?*

KEY QUESTION 2. *Given two statements $s_1$ and $s_2$ in a program, may $s_1$ and $s_2$ run in parallel and under what condition?*

We use an overloaded map (named MHP) to denote both the variations of the analyses; the number of arguments to the function (for example, $\mathsf{MHP}(s_1)$, or $\mathsf{MHP}(s_1, s_2)$) can be used to make the distinction. These two key questions naturally lead to two corresponding auxiliary challenges.

AUXILIARY CHALLENGE 1. *Compute the* MHP *map for each statement in the program (based on the key question 1).*

AUXILIARY CHALLENGE 2. *Compute the* MHP *map for each pair of statements in the program (based on the key question 2).*

The best known algorithm to answer the key question 2 (for X10-like languages that support async-finish-atomic parallelism) is by Agarwal et al. [1], and it has a worst-case complexity of $O(N^2)$, where $N$ is the program size. The key question 1 and the two auxiliary challenges can also be answered using this algorithm, which incurs higher costs: $O(N^3)$ for the key question 1, and $O(N^4)$ for both the auxiliary challenges 1 and 2. These complexities are significantly high.

Though $\mathsf{MHP}(s_1)$ can be computed by iteratively invoking $\mathsf{MHP}(s_1, s_2)$ and varying $s_2$ over all the statements in the program, we show that from the efficiency point of view it is beneficial to treat the two key questions separately. We present a fast incremental MHP (iMHP) algorithm to answer the key question 1 in amortized $O(N^2)$ time. To answer the key question 2 efficiently, we present an algorithm $\mathsf{aMHP}_{new}$. This algorithm extends the MHP algorithm of Agarwal et al. [1] in a way that it lowers the computational complexity to answer the key question 2 (from worst case quadratic in program size to linear) without compromising on the precision of the original algorithm. The improvements in our proposed algorithms stem mainly from two design choices we make: (i) In case of the iMHP algorithm, we start with a serial version of the input program and compute the MHP information by iteratively introducing one parallelism related construct at a time and in the process reuse the information gathered in one iteration in the other. (ii) In case of $\mathsf{aMHP}_{new}$, we propose a new representation called *compact conditional vector sets* (CCS) to represent the conditional vector sets [1], and reuse the prior computed information to reduce the overall complexity. The CCS representation consumes significantly less space and correspondingly requires much less time to operate on it. We show that the impact of these design choices is further felt when we proceed to address the two auxiliary challenges.

Traditionally May-Happen-in-Parallel (MHP) analysis has been explored in the context of several parallel programming languages [1, 4, 7, 16, 18, 22, 23]. In his seminal paper, Taylor [26] shows that the problem of MHP analysis for all pairs of statements in a program is undecidable in general, and is NP-complete (if we assume that all the control flow paths are executable) for programs that use low level synchronization primitives like Ada *rendezvous*. The problem is more tractable, if we additionally assume that such low level synchronization primitives are not present. Many of the proposed MHP algorithms in research (including our paper) are based on these two assumptions and are conservative in nature: if the analysis concludes that two statements may run in parallel, then during actual execution the statements may or not run in parallel. But if the

analysis concludes otherwise, then during actual execution the two statements are guaranteed to not run in parallel.

In the context of Java, Naumovich et al. [23] answer the key question 1 and Barik [4] answers the key question 2. Chen et al. [5] present some promising initial results on computing MHP analysis (answering key question 2) for Java programs in $O(N + E)$ time, where $N$ is the program size and $E$ is the number of control edges in the parallel reachability graph.

A *simpler* version of the key question 2 answers the boolean question "Can two given statements run in parallel (oblivious to the conditions under which the MHP analysis may hold)?" Similarly, the corresponding auxiliary question 2 can also be framed. Recently, for a *restricted* task parallel language like Featherweight X10 [14], Lee et al. [15] present two interesting procedures to answer this simpler key question (in time linear in the program size) and the auxiliary challenge 1 (in time cubic in the program size, in the worst case – though the authors claim that in practice it will be mostly linear). In contrast, our aMHP$_{new}$ algorithm (that answers the key question 2) is much more precise than that of Lee et al; similar to the analysis of Agarwal et al. [1], we compute conditional MHP information for pairs of statements nested inside serial loops. Further, we work in a more general setting: we handle atomic constructs and admit multi-place programs (see Section 2) that are not allowed in Featherweight X10. Considering the non local nature of the interaction between the atomic sections of different `async`s in X10 (atomic sections inside two parallel asyncs may run in parallel, only if they are running on two different places), it is not clear, how the results of Lee et al. can be extended in a straightforward manner to handle such extensions to Featherweight X10. Note: multiple places and atomics are one of the key reasons for the increased complexity of the original algorithm of Agarwal et al. [1]. Even for us, handling them carefully is critical to keep the complexity in check.

**Contributions:**

- To answer the two key questions discussed above, we present two algorithms (one new and one as an extension to that of Agarwal et al. [1]). Compared to the scheme of using the algorithm of Agarwal et al. for answering the key questions and auxiliary challenges, our scheme improves the computational complexity by a factor ranging between $O(N)$ to $O(N^2)$, where $N$ is the program size.
- We extend our proposed analysis techniques to efficiently handle multi-place programs [24].
- To estimate the impact of the improvements in the computational complexity of our proposed algorithms on the execution time, we present a two-tiered evaluation scheme: (i) evaluation on publicly available IMSuite [10] benchmark kernels, and (ii) a novel empirical evaluation on synthetically generated benchmarks. We designed a parameterized tool to generate a wide variety of representative program structure trees (PST: a program representation proposed by Agarwal et al. [1]), with varying number of nodes, varying percentages of serial constructs (for example, conditional statements, loops) and varying percentages of parallel constructs (for example, task creation, join and atomic constructs). We show that in both the cases our proposed techniques clearly outperform the algorithm of Agarwal et al. [1].

The paper is organized as follows: Section 2 presents a quick introduction to KX10 language (a subset of X10, over which we describe our analyses), and a brief overview of the MHP analysis technique of Agarwal et al. [1]. We present our extensions to the MHP analysis for uni-place programs in Section 3. Section 4 presents the changes to our analyses to handle multi-place programs. We discuss our evaluation in Section 5 and conclude in Section 6.

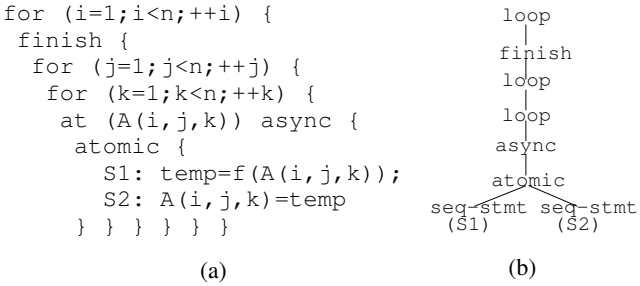



**Figure 1:** A Loop nest and its PST – ported from Agarwal et al. [1]

## 2. Background

### 2.1 Language

We first briefly describe a strict subset of X10 [24], (we call it KX10), over which we define our analyses. This subset is similar to the subset used by Agarwal et al [1] to describe their analysis. KX10 only admits the following concurrency constructs: `async`, `finish`, `atomic`, and places. The statements in KX10 can be derived in the following manner.

```
S ::= // Statements
      async S
    | at (pexp) async S
    | atomic S
    | finish S
    | seq(S)
```

Here, for a non-terminal `A`, *seq(*`A`*)* is used to denote programs formed from `A` by closing under sequential constructs. These constructs include assignments, declarations, expressions, method invocations, loops etc.

The statement `async S` causes the current task to create a new asynchronous child task to execute the statement `S`. The 'parent' task can run in parallel with the child task.

A *place* in KX10 is an abstraction of a computational unit (capable of running many threads), along with some finite memory. Or in other words, a place is a collection of data and tasks operating on that data. Each KX10 task executes in a place. At runtime, a KX10 program may execute over one or more places. The statement `at(pexp) async S`, creates a task at place `pexp` to execute `S`. The expression `pexp` (called place expression) is evaluated at runtime to compute the place value. If the place-value of an `async` that is being created matches that of the current place (or `at(pexp)` is omitted), then the task is considered local. Since each data item is local to the place of execution, KX10 enforces intra-place atomicity.

`atomic S` realizes a global critical section. `atomic` sections (at the same place) execute as if they are executed in a mutually exclusive way. No `async` or `finish` can be executed inside an atomic region.

`finish S` is a join statement. The body `S` is executed with a surrounding join such that all activities created inside `S` have to terminate before the task executing the join can proceed. At runtime, each instruction executed in KX10 program has a unique associated task, which in turn has a unique *immediately enclosing finish* (IEF). Each KX10 program has an implicit task (corresponding to the main task) and an implicit outermost finish enclosing the implicit task.

An `async` statement $A_L$ contained in a statement $S$ is called an *escaping* `async` [9], if it is not enclosed in a `finish` statement within $S$. The IEF of $A_L$ is not contained within $S$.

Using the constructs described here, we can write a parallel uni-place loop (for example: '`for (p=m;p<n;++p) async S`'), or parallel loop distributed across multiple places (for example: '`for(p=m;p<n;++p) at (pexp) async S`').

Figure 1(a) shows a sample KX10 program (ported from the example snippet used by Agarwal et al. [1]) Note: `A(i,j,k)` accesses the three dimensional array, and is also used (as the first argument to the `at` statement) to refer to the place where the array element is distributed. Later in the paper, we use the same example to illustrate the comparative output as generated by our proposed algorithm.

### 2.2 May-Happen-in-Parallel Analysis

We give a brief overview of the intra-procedural MHP analysis technique proposed by Agarwal et al. [1] (we will name it as `AgarwalMHP`) that also works on KX10, and answers the key question 2. We present some insights into the underlying working of their algorithm and discuss some scopes for improvement.

The MHP algorithm `AgarwalMHP` is an intra-procedural algorithm that works on a program representation called *program structure tree* (PST) that compresses an abstract syntax tree (of a procedure) to consider only nodes of the following types: `root`, `seq-stmt`, `loop`, `async`, `finish` and `atomic`. The `root` type corresponds to the start of the procedure, and the `seq-stmt` type corresponds to all other statements except `loop`, `async`, `finish` and `atomic`. In this paper, we will be using the words *statement* and *node* interchangeably. Figure 1(b) shows the PST generated for the loop nest shown in Figure 1(a).

Assume that nodes $S_1$ and $S_2$ have exactly $k \geq 0$ loop nodes as common ancestors in the PST. Let $S_1[i_1, \ldots, i_k]$ and $S_2[j_1, \cdots, j_k]$ denote execution instances of $S_1$ in the iteration vector $\langle i_1, \ldots, i_k \rangle$, and $S_2$ in the iteration vector $\langle j_1, \cdots, j_k \rangle$ of the common loops. The algorithm `AgarwalMHP` takes as input two statements and the PST for the procedure in which the two statements occur. The invocation `AgarwalMHP`($S_1$, $S_2$, PST) updates two output variables: (i) a boolean variable $mhpStatus$ to reflect if $S_1$ and $S_2$ may run in parallel, and (ii) a set $CS$ of *condition vectors* under which they may not run in parallel. Each condition vector in $CS$ contains $k$ elements each. Each such element of a vector can be one of the binary functions $=, \neq,$ and $*$, defined below:

$$= (i,\ j) \quad : \begin{cases} \texttt{true}, \text{ if } i \text{ EQ } j, \\ \texttt{false}, \text{ otherwise.} \end{cases}$$

$$\neq (i,\ j) \quad : \begin{cases} \texttt{true}, \text{ if } i \text{ NEQ } j, \\ \texttt{false}, \text{ otherwise.} \end{cases}$$

$$* \ (i,\ j) \quad : \quad \texttt{true}$$

If $mhpStatus$ = `false` and there exists $\langle C_1, \cdots C_k \rangle \in CS$ such that $C_x(i_x, j_x) =$ `true`, $1 \leq x \leq k$, then the instances $S_1[i_1, \ldots, i_k]$ and $S_2[j_1, \cdots, j_k]$ are guaranteed to not run in parallel. If $mhpStatus$ = `true`, the statements may run in parallel. For example, for the code shown in Figure 1(a), `AgarwalMHP(S1, S2)` returns (`false`, $\{\langle =, =, = \rangle, \langle \neq, *, * \rangle\}$), indicating that `S1` and `S2` will not run in parallel if they have the same `i-j-k` iteration vector, or if their iteration vectors differ on the value of `i`.

Figure 2(a) shows a simple extension to the MHP algorithm of Agarwal et al. [1] to answer the key question 1. The function `computeMHP` invokes `AgarwalMHP` repeatedly, for each node in the procedure. Similarly, as shown in Figures 2(b) and 2(c), we can repeatedly invoke `computeMHP` and `AgarwalMHP` to answer the auxiliary challenges 1 and 2, respectively.

The `AgarwalMHP` algorithm includes two sub-analyses: uni-place MHP analysis (that considers KX10 programs running on a single place) and multi-place MHP analysis (that analyzes multi-place KX10 programs, using the place equivalence analysis [1]). Both of the these versions have the same computational complexity. The same point holds for the three extensions presented in Figure 2.

```
1 Function computeMHP(Stmt s1, PST pst)
2 begin
3     foreach Stmt s2 in the Program do
4         AgarwalMHP(s1, s2, pst, mhp, CS);
5         if mhp then
6             MHP(s1) = MHP(s1) ∪ {s2};
7 end
```

**(a)** All statements running in parallel with a given statement.

```
1 Function computeMHP-allStmts(PST pst)
2 begin
3     foreach node s1 in pst do
4         computeMHP(s1, pst);
5 end
```

**(b)** MHP for all statements in the PST.

```
1 Function computeMHP-allPairs(PST pst)
2 begin
3     foreach node s1 in pst do
4         foreach node s2 in pst do
5             AgarwalMHP(s1, s2, pst, mhp, CSo);
6             MHP(s1, s2) = mhp; CS(s1, s2) = CSo;
7 end
```

**(c)** MHP for all pairs of nodes in the PST.

**Figure 2:** Answering the key question 1, and auxiliary challenges using `AgarwalMHP`.

***Complexity of AgarwalMHP and the presented extensions :*** The main source of the complexity of `AgarwalMHP` is a loop that traverses over the partial height of the PST (say, bound by $H$) and conditionally builds and inserts a vector of size $O(H)$ to the vector set CS (Step 6(b), Figure 2, Agarwal et al. [1]). This leads to a worst case complexity of $O(H^2)$ (this is despite the use of efficient union-find data structures [6], for set union) – which, in the worst case, is quadratic in the PST size. This is in contrast to the complexity of $O(H)$ as claimed by the authors, who seem to ignore the cost involved in building and adding a condition vector of size $O(H)$ to an existing set.

Assuming the size of PST to be $N$, we can now derive the complexities of the `computeMHP`, `computeMHP-allStmts`, and `computeMHP-allPairs` to be $O(H^2 \times N)$, $O(H^2 \times N^2)$, and $O(H^2 \times N^2)$, respectively.

## 3. Improvements to MHP Analysis

We present two different algorithms to answer the two key questions and the auxiliary challenges presented in Section 1. We first present an incremental MHP algorithm to answer the key question 1 and then present an extension to the MHP algorithm of Agarwal et al. [1] to efficiently answer the key question 2. For the ease of presentation, we first present our extensions assuming programs to be running on a single place. We extend our presented analyses for multi-place programs in Section 4.

### 3.1 List of Statements that May Run in Parallel With a Statement

We now present our incremental MHP (iMHP) algorithm to answer the key question 1. The core idea behind our iMHP algorithm is that we can incrementally compute the MHP information by considering each parallel element in the program (such as `finish`, `async`, and `atomic`) one after the other as an update. The advantage of

```
1  Function iMHP-driver(PST pst)
2  begin
3      (SeqP, Lst) = Replace all parallel elements in pst
       with dummy nodes and store the updates to be performed;
       // Lst contains all the updates.
4      PST CP = SeqP; // Current Program.
5      MHP = {};
6
7      LstFinish = Entries in Lst of the form 'replace
       L:dummy by L:finish ';
8      foreach r ∈ LstFinish do
9          iMHP-addFinish(CP, L);
10         CP = r(CP)
11
12     LstAsync = Entries in Lst of the form 'replace
       L:dummy by L:async ';
13     foreach r ∈ LstAsync do
14         iMHP-addAsync(CP, L);
15         CP = r(CP)
16
17     LstAtomic = Entries in Lst of the form 'replace
       L:dummy by L:atomic ';
18     foreach r ∈ LstAtomic do
19         iMHP-addAtomic(CP, L);
20         CP = r(CP)
21 end
```

**Figure 3:** Driver for the incremental MHP analysis. Process the `finish`, `async` and `atomic` statements in that order.

such a scheme is that for each such update, the MHP maps of only a small localized subset of nodes need to be modified.

Figure 3 shows the driver for our incremental MHP algorithm. To compute the MHP information for each statement in a given task parallel program, we start with the serial version of the PST, obtained by replacing each of the PST nodes corresponding to the parallel elements (`asyncs`, `finish` and `atomic`) with a corresponding dummy node. Each such dummy node (for example, one corresponding to `L:async`) has an entry in `Lst`, as an update to be applied (for example, 'replace `L:dummy` by `L:async`').

We re-introduce the parallel elements in a particular order (`finish` nodes, `async` nodes, and then `atomic` nodes). We choose this order to improve the efficiency of our algorithm. For each re-introduction, we invoke the corresponding `iMHP-add` method (shown in Figure 4) that takes as input the current PST and the node corresponding to the chosen update. Our analysis updates the MHP map to include the MHP information for the modified PST. We also apply the chosen update on the PST to derive the modified PST. We now discuss, how we compute the MHP information on different types of updates.

`finish` update (replace `L:dummy` by `L:finish`): We introduce the `finish` nodes before introducing any `async` statement. Thus the addition of `finish` nodes does not affect the MHP maps of any other statement. The `iMHP-addFinish` routine (shown in Figure 4(a)) sets the MHP map corresponding to the `finish` node. As discussed in Section 2, no `async` escapes a `finish` node. For each statement, EscAsyncs keeps the information on the `asyncs` that may escape the statement; the EscAsyncs map is used later in the paper.

`async` update (replace `L:dummy` by `L:async`): Say the IEF of $L$ is given by the singleton set $A$. After the update, $L$ additionally

```
1  Function iMHP-addFinish(PST pst, Node L)
2  begin
3      MHP(L) = {}; // Initialize the MHP map.
4      EscAsyncs (L) = {}; // No task escapes L
5  end
```

**(a)** Replacing `S` by `finish S`.

```
1  Function iMHP-addAsync(PST pst, Node L)
2  begin
3      A = IEF(L);// Tasks that may contain L
4      m = {}; // to contain MHP(L)
       // Add to the MHP map of each stmt
          inside the proposed task, all the
          stmts within the common IEF that
          may start after the task.
5      foreach s ∈ Descendents(A) do
6          if s is reachable from L then m = m ∪ {s};
7      D = Descendents(L);
8      foreach l ∈ D do
9          MHP(l) = MHP(l) ∪ m;
       // Update the MHP maps of the
          statements in MHP(L)=m
10     foreach a ∈ m do
11         MHP(a) = MHP(a) ∪ D;
       // Update the EscAsyncs maps.
12     P = L;
13     repeat
14         EscAsyncs(P) = EscAsyncs(P) ∪ {L};
15         P = Parent of P in the PST;
16     until P is a finish node ;
17 end
```

**(b)** Replacing `S` by `async S`.

```
1  Function iMHP-addAtomic(PST pst, Node L1)
2  begin
3      foreach L2 ∈ MHP(L1) do
4          if inAtomic(L2) AND L1 and L2 access the same
           memory location and one of them is a write then
5              MHP(L1) = MHP(L1) − {L2};
6      foreach L2 ∈ Nodes do
7          if inAtomic(L2) AND L1 and L2 access the same
           memory location and one of them is a write then
8              if L1 ∈ MHP(L2) then
9                  MHP(L2) = MHP(L2) − {L1};
10     foreach s ∈ Descendents(L1) do
11         inAtomic (s) = true;
12 end
```

**(c)** Replacing `S` by `atomic S`.

**Figure 4:** Components of the Proposed incremental MHP analysis – updates to the MHP maps on introducing a new parallel construct.

runs in parallel with all the descendants of $A$ that are *reachable* from $L$ during program execution. Further, the lifetime of $L$ does not exceed that of $A$. Figure 4(b) presents our incremental MHP algorithm that updates the MHP for each `async` update.

`atomic` update (replace `L:dummy` by `L:atomic`): Agarwal et al. [1] observe that instances of two statements $S1$ and $S2$ occurring in atomic sections will have MHP$(S1, S2)$ = `false`.

We implement the idea (see Figure 4(c)) in two phases: (i) For each statement in the program, we maintain a map `inAtomic`: $Stmts \rightarrow boolean$; `inAtomic`($S1$) returns `true`, if $S1$ is inside an atomic block. Upon introducing a new atomic block, we update the `inAtomic` map of all the statements inside the atomic block to `true`. (ii) We modify the MHP map of $S1$ and all the MHP maps of all those tasks that contain $S1$.

***Example:*** To illustrate the working of our incremental MHP algorithm, we use Figures 5(a) and 5(b) that present the example code and the corresponding PST that was used by Agarwal et al. [1]. After the initial pre-pass, we get a modified PST (Figure 5(c)) and `Lst` contains the list of updates to be applied. Figure 6 shows how the MHP maps change with each update of our incremental MHP algorithm. We start with the `finish` updates and then proceed onto the `async` updates. The order in which we choose the individual `finish` updates or `async` updates does not have a bearing on the final MHP information. The `root` node does not run in parallel with any of the nodes and we avoid showing the same in the MHP maps in Figure 6, for brevity. A sample update S6:`async` indicates that we replace S6:`dummy` by S6:`async` in the PST. The associated `iMHP-addAsync` call updates the MHP maps of only three nodes `S7`, `S11`, and `S12`.

### 3.2 MHP Relation Between Two Statements

We now present our extensions to the MHP algorithm of Agarwal et al. [1] to answer the key question 2 presented in Section 1, for uni-place programs. Instead of computing the MHP relation as an inverse of NEP (Never Execute in Parallel) relation, we directly compute the MHP relation. The main source of our efficiency stems from the design choice we make to represent conditional vector sets using a compact representation called CCS.

#### 3.2.1 Compact Conditional Vector Sets (CCS)

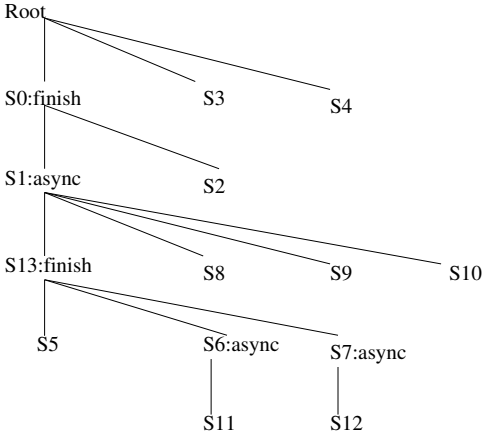
An interesting aspect of the conditional vectors used in the uni-place analysis of `AgarwalMHP` is that in any vector there is at most one $\neq$ function. All the functions to the left of this function can only be $=$ and functions to the right of $\neq$ can only be $*$. If there is no $\neq$ function in the vector, then all the entries are $=$ functions. Thus, it is enough to note the index of the $\neq$ function to derive the rest of the vector (we will call it the *key* index); thus CS can be seen as a set of key indices. Assuming that the original conditional vectors stored the elements at index 1 onwards, the special value '0' for the key index can be used to denote the vector where all the entries are "=" functions.

We store the conditional vector set as a zero based array (CCS) containing $M+1$ boolean elements, where $M$ is the number of loops in the program. We index each loop with a unique positive index and the $i^{th}$ element in the CCS for any node indicates if the condition vector with key value $i$ is a member of the conditional vector set for that node. For example, CCS$[i]$ = `true` indicates that the conditional vector with key value $i$ is a member of the conditional vector set; we call it the compact conditional vector set representation. For example, the conditional vector set $CS = \{\langle C_1 = \text{“}=\text{”}, C_2 = \text{“}\neq\text{”}, C_3 = \text{“}*\text{”}\rangle, \langle C_1 = \text{“}=\text{”}, C_2 = \text{“}=\text{”}, C_3 = \text{“}\neq\text{”}\rangle\}$, is represented as CCS = [`false`, `false`, `true`, `true`]. The zeroth element is set to `false`, to indicate that this set does not contain the vector containing all "=" functions.

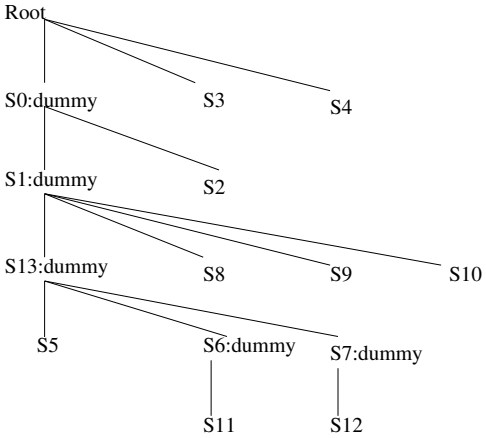
We also note that any two pairs of statements that have the same least common ancestor in the PST will have identical values in the positive indices of CCS (the value at zeroth index may differ). Our goal is to store the CCS information at each of the ancestor (non-leaf) nodes.

```
S0: finish {
  S1: async {
   S13: finish {
      S5: ...
      S6: async S11
      S7: async S12
    }
    S8: ...
    S9: ...
    S10: ... } // end async
  S2: ... } // end finish
S3: ...
S4: ...
```

**(a)** Input program.




**(b)** Generated PST.




**(c)** PST after initialization.

**Figure 5:** Input program used by Agarwal et al. [1], the corresponding PST and the resulting PST after our initialization pre-pass.

#### 3.2.2 MHP Relation for a Pair of Statements

For a given parallel program, the MHP related queries for a pair of statements (say, $s_1$ and $s_2$) in the input program are answered by invoking the function aMHP$_{new}(s_1, s_2)$, shown in Figure 7.

aMHP$_{new}$ invokes a procedure `computeCCS` (shown in Figure 7(a)) that computes and stores the CCS information for a given node in the PST. Note that the index ($id$) of any loop node is a positive integer. The computational complexity of this procedure is $O(H)$, where $H$ is the maximum height of the PST.

**Initialization:**

```
Lst = {S13:finish, S0:finish, S6:async,
       S7:async, S1:async}
```

**Process** `finish` **updates:**

```
S13:finish: MHP(s) = {},
    where s∈{S0, S1, S2, S3, S4, S5, S6,
         S7, S8, S9, S10, S11, S12, S13}
S0:finish: MHP(s) = {},
     where s∈{S0, S1, S2, S3, S4, S5, S6,
          S7, S8, S9, S10, S11, S12, S13}
```

**Process** `async` **updates:**

```
S6:async: MHP(S11) = {S12, S7}
MHP(S7) = MHP(S12) = {S11}
MHP(s) = {} where s∉{S7, S11, S12}

S7:async: MHP(S11) = {S12, S7}
MHP(S7) = MHP(S12) = {S11}
MHP(s) = {} where s∉{S7, S11, S12}

S1:async: MHP(S11) = {S2, S7, S12}
MHP(S2) = {S5, S6, S7, S8, S9, S10, S11,
           S12, S13},
MHP(S7) = MHP(S12) = {S2, S11},
MHP(s) = {S2},
     where s∈{S5, S6, S8, S9, S10, S13},
MHP(S0) = MHP(S1) = MHP(S3) = MHP(S4)={}
```

**Figure 6:** MHP sets computation, for the example shown in Figure 5.

The algorithm $\mathsf{aMHP}_{new}$ is presented in Figure 7(b). Most part of this algorithm is a faithful recasting of the NEP algorithm and the main MHP computation algorithm of Agarwal et al. [1], for uni-place programs, in our modified settings. One interesting point to note is that we reuse the CCS information from the least-common-ancestor (`LCA`) for all the non-zero indices. We set $CCS[0]$ for a given pair of statements, if the condition vector containing all "=" functions can be part of the conditional vector set.

### 3.3 Complexity and Efficiency

We now discuss the computational complexity of our presented MHP algorithms to answer the key questions and auxiliary challenges discussed in Section 1. We will use $N$ to denote the program size (or the number of nodes in the PST), $H$ to denote the maximum height of the PST, $C$ to denote the maximum number of concurrency constructs (typically a very small number), and $\alpha$ to denote the inverse Ackermann function [6], a slow growing function. These results are summarized in Figure 8.

**Key Question 1**: Given a task parallel program, the incremental MHP is computed by repeatedly invoking the `iMHP-add` routines. While the `iMHP-addFinish` is an $O(1)$ operation, the other two `iMHP-add` routines may execute a few 'set union', 'find element' or 'delete element' operations (at most $N$ times). Use of efficient disjoint-set union-find algorithms can help in near constant time ($O(\alpha(N))$) union, find [6] and delete element [3, 12] operations. Thus the amortized complexity of invoking the `iMHP-add` routines is $O(N \times \alpha(N))$. Since these `iMHP-add` routines are invoked at most $C$ times, it results in an overall complexity of $O(C \times N \times \alpha(N)) \approx O(C \times N)$. Compared to `computeMHP`, our algorithm improves the complexity by a factor of $O(H^2/C)$. In the worst case, where $H = O(N)$ and $C = O(N)$, the resulting improvement is a factor of $O(N)$.

**Auxiliary challenge 1**: The cost of computing the MHP information for all the statements in the program, using our algorithm (same as the one used to answer Key Question 1), is

```
1  Function computeCCS(Stmt L)
2  begin
3      if L.CCS is already available then
4          return L.CCS;
5      L.CCS = new boolean[maxHeight+1];
6      boolean seqLoop = true;
7      Stmt P = L;
8      while P is ≠ root do
9          if P is an async node then
10             seqLoop = false;
11         if P is a finish node then
12             seqLoop = true;
13         if P is a loop node then
14             L.CCS[P.id] =seqLoop;
15         P = P.parent();
16     return L.CCS;
17 end
```

**(a)** Compute CCS for the given node.

```
1  Function aMHP_new(Stmt S1, Stmt S2)
2  begin
3      Stmt A = LCA(S1, S2);
4      boolean asyncS1 = false;
5      for (N = S1; N ≠ A; N = N.Parent()) do
6          if N is an async node then asyncS1 = true;
7          if N is a finish node then asyncS1 = false;
8      boolean asyncS2 = false;
9      for (N = S2; N ≠ A; N = N.Parent()) do
10         if N is an async node then asyncS2 = true;
11         if N is a finish node then asyncS2 = false;
12     boolean [ ] retCCS =computeCCS(A);
13     retCCS[0] = false;
14     if S1 ≠ S2 then
15         AS1 = (S1.Parent() eq A)?S1 : S1.Parent();
16         AS2 = (S2.Parent() eq A)?S2 : S2.Parent();
17         switch (asyncS1, asyncS2) do
18             case (false, false)
19                 retCCS[0] = true;
20             case (false, true)
21                 retCCS[0] = ¬(AS2 dominates AS1?);
22             case (true, false)
23                 retCCS[0] = ¬(AS1 dominates AS2?);
24             case (true, true)
25                 retCCS[0] = false;
26
27     if inAtomic(S1) AND inAtomic(S2) AND S1 and S2
       must access the same memory location and one of them is
       a write then
28         return (false, retCCS);
29     if every element of retCCS has value false then
30         return (true, retCCS);
31     return (false, retCCS);
32 end
```

**(b)** Algorithm to answer key question 2.

**Figure 7:** Compute MHP information for a pair of statements.

$O(C \times N \times \alpha(N)) \approx O(C \times N)$. Compared to the algorithm `computeMHP-allStmts`, our algorithm improves the complexity by a factor of $O(H^2 \times N/C)$. In the worst case, where $H = O(N)$ and $C = O(N)$, the resulting improvement is a factor of $O(N^2)$.

**Key Question 2**: To compute the MHP relation between two given statements in the program, we invoke the function $\mathsf{aMHP}_{new}$. Considering that (i) the time to compute the Least Common Ancestor (LCA) can be done in constant time [25], (ii) the dominator information being a standard analysis can be considered to be pre-computed, and (iii) `computeCCS` takes $O(H)$ time, our algorithm $\mathsf{aMHP}_{new}$ takes $O(H)$ time. Compared to `AgarwalMHP` that takes $O(H^2)$ time, our algorithm improves the complexity by a factor of $O(H)$. In the worst case, where $H = O(N)$, the resulting improvement is a factor of $O(N)$.

**Auxiliary challenge 2**: Now we compute MHP map for each pair, by repeatedly invoking the function $\mathsf{aMHP}_{new}$; this takes $O(H \times N^2)$ time. Compared to `computeMHP-allPairs` (Figure 2(c)) that takes $O(H^2 \times N^2)$ time, our algorithm improves the complexity by a factor of $O(H)$. In the worst case, where $H = O(N)$, the resulting improvement is a factor of $O(N)$.

### 3.4 Discussion

- Consider the code snippet shown in Figure 1(a). The call $\mathsf{aMHP}_{new}$ (`S1`, `S2`) returns (`false`, [`true`, `true`, `false`, `false`]). This implies that two instances of `S1` and `S2` may not run in parallel, (a) when they have the same iteration vector – given by the first `true` value in the CCS, or (b) when they have different values of `i` in their iteration vector – represented by the second value of `true` in the CCS. Or in other words, two instances of `S1` and `S2` may run in parallel, when they are running in the same iteration of `i` and different iterations of `j` and `k` (corresponding to the two `false` entries in the condition vector). In Section 4, we further refine this result by taking into consideration the `atomic` construct and the multi-place code.
- The traditional approach (`computeMHP`) does not reuse information computed in prior iterations of the algorithm. In contrast, we reuse (in `computeCCS`) the CCS information computed for different pairs of statements having the same least-common-ancestor. Though this does not change the computational complexity of our algorithm, in practice, this improves the execution time of our response to the auxiliary challenge 2.
- iMHP: Our analysis starts with the serial version of the program and computes MHP information incrementally by reintroducing the parallelism related annotations in a particular order("finish" followed by "async" and "atomics"). The order is important: if "finish" annotations are introduced after or during the introduction of the "async" annotations, then after each introduction of the "finish" annotation, we have to recalculate the MHP information of all the statements under the "finish" node. But when we introduce the "finish" annotations in a serial program, it does not change the MHP maps. Further, say we reintroduce an "async" annotation on a node $s_1$. The MHP information of the descendant nodes of $s_1$ can be incrementally updated (not recomputed from scratch) using the information about the tasks that may be executed after $s_1$ and are under the same immediately-enclosing-finish (IEF), and vice versa.
- Summary: The improvements realized in the iMHP algorithm is mainly because of the localized updates resulting from the incremental introduction of the parallel constructs. And the improvements obtained in $\mathsf{aMHP}_{new}$ are mainly because of compact representation of CCS. Further, reusing the CCS information computed in prior invocations improves the analysis time, in practice.

## 4. MHP for Multi-Place Programs

We now extend our proposed analysis techniques presented in Section 3, to handle multi-place programs. Our extension is based on the observation of Agarwal et al. [1] that two statements inside two different atomic regions are considered to not run in parallel, only if they are guaranteed to be executed in the same place. We will assume that a global value numbering algorithm has already been run to identify the places at which different activities are invoked (an assumption similar to that by Agarwal et al. [1]). For each PST node $S$, we assume that the map $V(S)$ gives the value number of the place at which $S$ will execute.

### 4.1 Computing MHP(L1) for Multi-Place Programs

We can extend our incremental MHP (iMHP) algorithm to answer the key question 1, for multi-place programs, in a straightforward way. This is done by updating the predicates at Line 4 and 7 in the `iMHP-addAtomic` routine (Figure 4(c)) to the following:
**if** (`inAtomic`$(L_2)$ *AND* $(V(L_1) == V(L_2))$ *AND* $L_1$ *and* $L_2$ *access the same memory location and one of them is a write*)

### 4.2 Computing MHP(L1, L2) for Multi-Place Programs

We now present an extension to our algorithm to answer the key question 2, for the multi-place KX10 programs. The basis of the multi-place MHP analysis of Agarwal et al. [1] is the Place Equivalence (PE) analysis that tells if two given statements (both present inside atomic regions) run at the same place. We present an extension to their PE analysis. We then extend the $\mathsf{aMHP}_{new}$ algorithm (using the modified PE analysis) to compute the MHP information for multi-place programs.

#### 4.2.1 Compact Conditional Vector Sets for PE Analysis

Similar to the approach of Section 3.2, we modify the conditional vector representation to derive a second type of compact representation called CCSp, to suit the results of PE analysis.

An interesting aspect of the conditional vectors used in the PE analysis of `AgarwalMHP` is that in any conditional vector, all the functions to the left of $*$ can only be $=$, and functions to the right of $*$ can only be $*$. If there is no $*$ function in the vector, then all the entries are $=$ functions. Thus, it is enough to note the index of the $*$ function to derive the rest of the vector (we will call it the *key* index); and conditional vector sets can be stored as a set of key indices. Similar to Section 3.2, we will assume that the original conditional vectors stored the elements at index 1 onwards, and the special value '0' for the key index is used to denote the vector where all the entries are "=" functions. Further, we will store CCSp as a zero based boolean array of size $M + 1$, where $M$ is the number of loops in the program.

#### 4.2.2 PE Analysis

Figure 9 gives a sketch of our scheme to perform PE analysis. Given two statements $S_1$ and $S_2$, we invoke the `computeCCSp` function on the least common ancestor of $S_1$ and $S_2$. This function requires information about loops for which the place-expression of any async is place-invariant. Similar to the scheme of the original PE analysis [1], we compute the `placeLocalLoops` information, in a global loop-invariant pre-pass, and use it in the `computeCCSp` function. This function sets the $\mathsf{CCS}_p$ entry corresponding to each place local loop to `true`. Since the cost of the set-find operation is nearly $O(1)$, the computational complexity of the `computeCCSp` routine is $O(H)$.

Given two nodes $S_1$ and $S_2$, the algorithm $\mathsf{PE}_{new}$ returns a pair $(pe, Cp)$. If $pe =$ `true`, then it implies that $S_1$ and $S_2$ will always run at the same place. $Cp$ contains the conditional vectors represented in CCSp form, under which $S_1$ and $S_2$ may run at

| | Agarwal et al. [1] | | | Our approach | | | Impro-vment |
|---|---|---|---|---|---|---|---|
| | Complexity | Worst-case | Algorithm | Complexity | Worst-case | Algorithm | |
| MHP(S1) | $O(H^2 \times N)$ | $O(N^3)$ | `computeMHP` | $O(C \times N)$ | $O(N^2)$ | iMHP | $O(N)$ |
| $\forall$ S1: MHP(S1) | $O(H^2 \times N^2)$ | $O(N^4)$ | `computeMHP-allStmts` | $O(C \times N)$ | $O(N^2)$ | iMHP | $O(N^2)$ |
| MHP(S1, S2) | $O(H^2)$ | $O(N^2)$ | `AgarwalMHP` | $O(H)$ | $O(N)$ | aMHP$_{new}$ | $O(N)$ |
| $\forall$ S1, S2: MHP(S1, S2) | $O(H^2 \times N^2)$ | $(ON^4)$ | `computeMHP-allPairs` | $O(H \times N^2)$ | $O(N^3)$ | aMHP$_{new}$ | $O(N)$ |

**Figure 8:** Comparison of time complexity between iMHP and the MHP algorithm of Agarwal et al. [1].

```
1  Function computeCCSp(Stmt L)
2  begin
3      if L.CCS is already available then
4          return L.CCS;
5      L.CCS_p = new boolean[maxHeight+1];
6      Stmt P = L;
7      Set pLocalLoops = φ;
8      while P is ≠ root-loop-node do
9          if P is an async node with place expression e then
10             pLocalLoops = placeLocalLoops(e);
11         if P is a loop node then
12             if P ∈ pLocalLoops then
13                 L.CCS_p[P.id] =true;
14         P = P.parent();
15     return L.CCS_p;
16 end
```

**(a)** compute CCSp for all the nodes.

```
1  Function PE_new(Stmt S1, Stmt S2)
2  begin
3      Stmt A = LCA(S1, S2);
4      boolean asyncS1 = ···; // Similar to Fig.7
5      boolean asyncS2 = ···; // Similar to Fig.7
6      boolean [ ] retCCSp = computeCCSp(A);
       retCCSp[0] = false;
7      if S1 ≠ S2 then
8          if V(S1) = V(S2) then // global place numbers
           match
9              retCCSp[1] = true;
10             return (true, retCCSp);
11         else if ¬asyncS1 ∧ ¬asyncS2 then
12             retCCSp[0] = true;
13     if every element of retCCSp has value false then
14         return (true, retCCSp);
15     return (false, retCCSp);
16 end
```

**(b)** Compute the place equivalence relation for a pair of statements.

**Figure 9:** Compute place equivalence information.

the same place. After invoking the `computeCCSp` function, the PE$_{new}$ function checks if the global place numbers of `S1` and `S2` match and if so, returns a tuple with its first element set to `true`. If instances of `S1` and `S2` may not run at the same place conditionally, then PE$_{new}$ returns a tuple with its first element set to `false`, and the vector $retCCSp$ contains the corresponding conditions.

### 4.3 Multi-Place aMHP$_{new}$

To answer the MHP related queries in multi-place programs, we modify the aMHP$_{new}$ algorithm to return a triplet (instead of a pair); the third field in the triplet indicates a conditional vector set in CCSp form. The modifications are as follows:

- Replace the code in Figure 7, line 28 with the following:

  Say $(pe, Cp) =$ PE$_{new}(S_1, S_2)$;
  **return** $(\neg pe, retCCS, Cp)$;

- Each of the other return statements, in Figure 7, of the form **return** $(b, val)$ is replaced with **return** $(b, val, emptyCp)$, where $emptyCp$ is a boolean array with all the entries set to `false`.

**Example:** For the example shown in Figure 1, let us assume that the array `A` is distributed using (`BLOCK`, `BLOCK`, *) distribution. It indicates that for any fixed value of `i` and `j`, all the elements `A(i,j,k)`, for varying values of `k`, are mapped on to the same place. Invocation of PE$_{new}$(`S1`, `S2`), first invokes `computeCCSp`, with the loop identifier of the `k` loop as the argument. Which, in turn, finds that the `k` loop is a place local loop and returns the vector [`false`, `false`, `false`, `true`]. The `computeCCSp` function, when invoked identifies that the place local loops for the `async` under consideration is the `k` loop. PE$_{new}$(`S1`, `S2`) returns (`true`, [`true`, `false`, `false`, `true`]). This indicates that instances of `S1` and `S2` will execute in the same place for varying values of `k`, with same values of `i` and `j`.

Now incorporating the place equivalence analysis (given by PE$_{new}$) in the MHP analysis (given by aMHP$_{new}$), we get aMHP$_{new}$(`S1`, `S2`) = (`false`, [`true`, `true`, `false`, `false`], [`true`, `false`, `false`, `true`]). That is, instances of `S1` and `S2` may not run in parallel, if (i) their iteration vectors have the same values for `i`, `j`, and `k`, or (ii) if their iterations vectors have different values of `i`, or (iii) if their iteration vectors have the same values of `i` and `j`.

### 4.4 Complexity of MHP analyses for Multi-Place Programs

As it can be trivially seen, the complexity of the iMHP algorithm does not change because of the changes suggested in Section 4.1.

Let us now consider the MHP analysis discussed in Section 4.2. The cost of PE$_{new}$ algorithm is bounded by the cost of `computeCCSp`. And the latter is bound by $O(H)$. Thus the cost of the modified aMHP$_{new}$ algorithm for multi-place programs is still bound by $O(H)$. Thus, in terms of precision, our analysis is as precise as that of Agarwal et al. [1], but in terms of complexity it is clearly better.

## 5. Evaluation

The goal of this section is to study the empirical gains resulting from the improvements in computational complexities (summarized in Figure 8). We implemented our proposed algorithms and that of Agarwal et al. [1] and studied the execution behavior of these algorithms. The study is divided into two parts : (i) evaluation over real benchmarks, and (ii) evaluation over synthetic benchmarks.

| | Name | #LOC | #PST | #finish | #async | #atomic |
|---|---|---|---|---|---|---|
| 1. | BF | 374 | 27 | 3 | 3 | 0 |
| 2. | DST | 578 | 64 | 8 | 8 | 3 |
| 3. | BY | 555 | 53 | 5 | 5 | 1 |
| 4. | DR | 407 | 29 | 2 | 2 | 0 |
| 5. | DS | 718 | 119 | 3 | 3 | 0 |
| 6. | KC | 579 | 108 | 12 | 12 | 2 |
| 7. | DP | 495 | 44 | 5 | 5 | 1 |
| 8. | HS | 511 | 46 | 6 | 6 | 0 |
| 9. | LCR | 326 | 26 | 4 | 4 | 0 |
| 10. | MIS | 443 | 48 | 6 | 6 | 2 |
| 11. | MST | 981 | 149 | 15 | 15 | 6 |
| 12. | VC | 476 | 45 | 6 | 6 | 0 |

**Figure 10:** Some characteristics of the IMSuite kernels.

### 5.1 Evaluation on Real Benchmarks

We evaluated our proposed algorithms on the publicly available IMSuite [10] kernels that implement twelve classical distributed algorithms : breadth first search (BF – computes the distance of every node from the root and DST – computes the BFS tree), byzantine consensus (BY), routing table creation (DR), dominating set (DS), maximal independent set (MIS), committee creation (KC), leader election (DP – for general network, HS – for bidirectional ring network, and LCR – for unidirectional ring network), spanning tree (MST), and vertex coloring (VC). Figure 10 shows some of the characteristics of these kernels, including the number of lines of code, number of `finish`, `async`, and `atomic` constructs. This also lists the number of PST nodes in the main parallel section of the program (not the whole program). Note: the number of finish constructs matches the number of `async`s here as each `async` is inside a for-loop which is embedded inside a `finish`.

We use our proposed algorithms to answer the two auxiliary challenges (1 and 2). All the MHP queries were based on the PST nodes of the main parallel section of the code. Figure 11 shows a comparison between iMHP and `computeMHP-allStmts`, and aMHP$_{new}$ and `computeMHP-allPairs`. For each of the benchmarks, we use the average analyses time over five invocations each. Overall, it can be seen that iMHP runs 80–96% faster (compared to the baseline `computeMHP-allStmts`), and aMHP$_{new}$ runs 68–88% faster (compared to the baseline `computeMHP-allPairs`). This is in accordance to the improved complexity of the proposed algorithms (Figure 8).

### 5.2 Evaluation on Synthetic Benchmarks

To further analyze the impact of our proposed analyses in the context of different types of programs (that may be written), we first designed a parameterized tool to generate a wide variety of representative PSTs. These parameters can be used to vary the number of nodes, percentage of serial constructs (for example, conditional statements and loops) and percentage of parallel constructs (for example, task-creation, join and atomic constructs) in the PSTs. These generated PSTs represent many varieties of X10 programs, both existing and the ones that may be written in future. We first discuss some details of our PST generation scheme and then present an evaluation of our proposed techniques using the generated PSTs.

#### 5.2.1 PST Generator

We first make a minor extension to the structure of PST (Section 2) : besides `loop`, `async`, `finish`, `atomic`, and `seq-stmt`, we additionally expose `If` and `IfElse` nodes. Further, each of the `seq-stmt` nodes is classified into one of the following five categories: *Assignment*, *Break*, *Continue*, *FuncCall*, and *Return*. The addition of the new node types and classification of the `seq-stmt` node helps expose the control flow information (e.g., dominators) of the underlying program.

The PST generation algorithm takes as input the total number of nodes and a distribution of various types of nodes (which is specified as a percentage). The PST generation algorithm generates a PST with the following additional natural constraints : a `seq-stmt` node can only be a leaf node; an `atomic` node cannot be an ancestor of a `finish` or `async` node; a *Break*/*Continue* node should be inside a sequential loop; and nodes such as `async`, `finish`, `atomic`, and `loop` should have at least one child. After the PST is generated we use the standard algorithms [19] to compute the successors, predecessors and dominators.

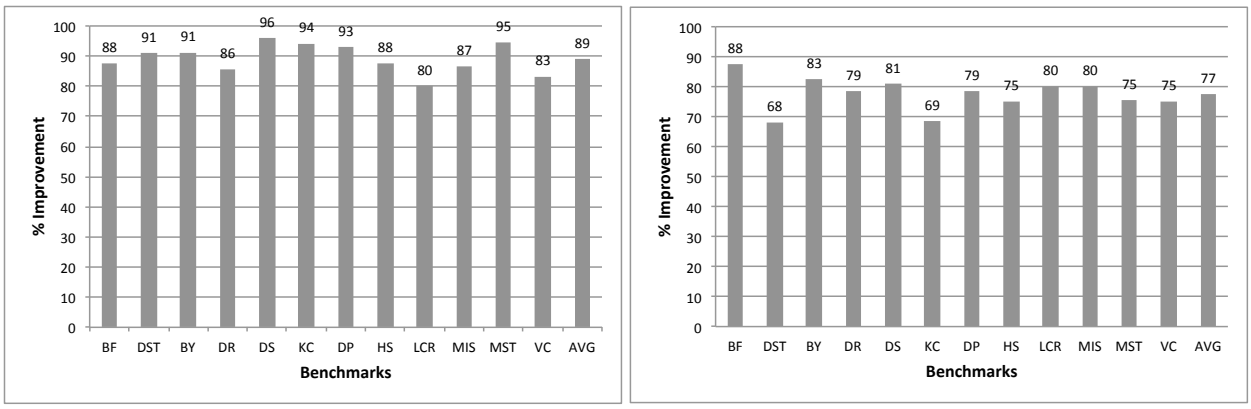
The PST generator can be accessed at: `http://www.cse.iitm.ac.in/~krishna/PSTGen/`

#### 5.2.2 Impact of the Proposed Techniques on PSTs with Varying Amounts Parallelism

We first aim to study the impact of our proposed techniques on PSTs with varying parallelism. We fix the number of nodes of the PST to 10,000 and fix the percentage of leaf nodes to 50%, where all the five category of statements may occur equally likely. We vary the number of `async` nodes in the program (from 1% to 25%, in increments of 1%), and the balance nodes are randomly chosen among `loop`, `finish`, `atomic`, `If` and `IfElse` nodes. Our study of the example programs shipped with X10 distribution and the IMSuite benchmark suite [10] showed that in practice the percentage of `async` nodes are typically small ($\approx 1-5\%$), in simple applications/kernels. PSTs with 20-25% `async` nodes represent programs with rather large number of `async`s. Thus, the chosen range for the percentage of `async`s covers a wide range of programs that may be written.

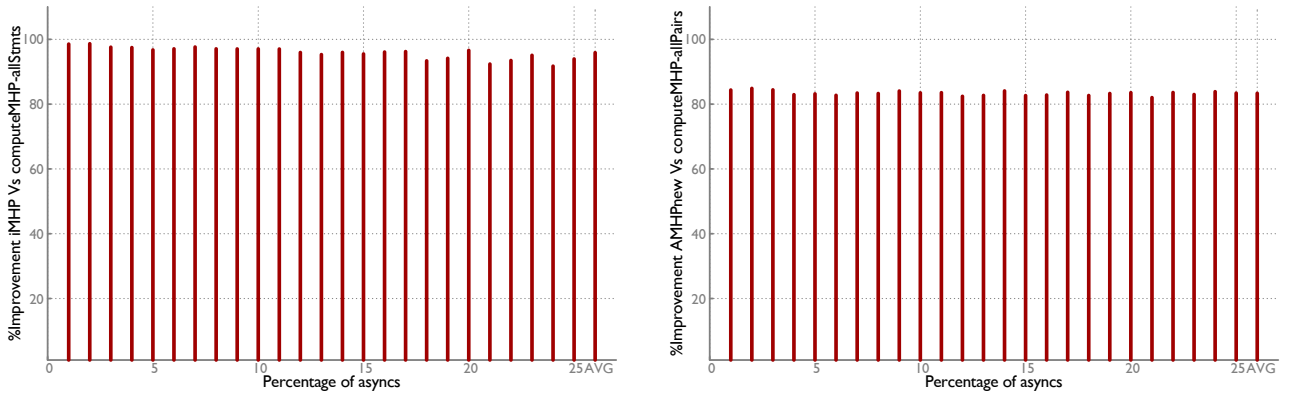
Figure 12 presents a comparative study of our proposed algorithms for varying percentage of `async` nodes. For each such chosen percentage, we ran the analyses five times each (by varying the random seed each time) and report the average execution time over these runs. This is done to amortize the effects of the possible variations in the PST based on the chosen random seed. Figure 12(a) presents a comparison of the iMHP algorithm and the function `computeMHP-allStmts` (Figure 2(b)), in answering the auxiliary challenge 1: $\forall$ S1: MHP(S1) and $\forall$ S1, S2: MHP(S1, S2). It can be seen that our proposed approach leads to significant improvements (between approximately 92% to 99% improvement, and on average 95.9%). The percentage improvement of scheme B Vs scheme A is given by 100×(time to run scheme A − time to run scheme B)/(time to run scheme A). It can be seen that when the number of `async` nodes in the PST is very low ($\leq 2\%$), the improvements are slightly on the higher side ($\approx 99\%$). This percentage comes down marginally to around 94% (on average) when the PST has large number of `async` nodes ($\geq 20\%$). This is partly because of the fact that our proposed iMHP analysis is quite fast (cost nearly zero) in handling constructs other than `async`s. And in case of `async`s, the cost we incur is slightly more (compared to the non-`async` nodes), though the cost is still less than that of Agarwal et al. We are confident that such a fast incremental analysis will be useful in efficient implementation of optimizations for task parallel programs (that may require re-computation of MHP information after each transformation) or refactoring tools that incrementally parallelize a given program, in a semantic preserving manner, by using the up-to-date MHP information.

Figure 12(b) presents a comparison of the aMHP$_{new}$ algorithm and the function `computeMHP-allPairs` (Figure 2(c)), in answering the auxiliary challenge 2. Again, compared to the baseline `computeMHP-allPairs` that uses the algorithm of Agarwal et al (`computeMHP`), our proposed mechanism results in significant savings (between 82% to 84%, on average 83.3%). We observed




**(a)** auxiliary challenge 1: iMHP Vs `computeMHP-allStmts`. **(b)** auxiliary challenge 2: aMHP$_{new}$ Vs `computeMHP-allPairs`.

**Figure 11:** Impact of the proposed algorithms on the IMSuite kernels.




**(a)** auxiliary challenge 1: iMHP Vs `computeMHP-allStmts` **(b)** auxiliary challenge 2: aMHP$_{new}$ Vs `computeMHP-allPairs`

**Figure 12:** Impact of the proposed techniques on PSTs with varying parallelism

that increasing the percentage of `async` nodes did not have much visible effect on the gains registered by our aMHP$_{new}$ analysis.

#### 5.2.3 Impact of the Proposed Techniques on PSTs with Varying Size

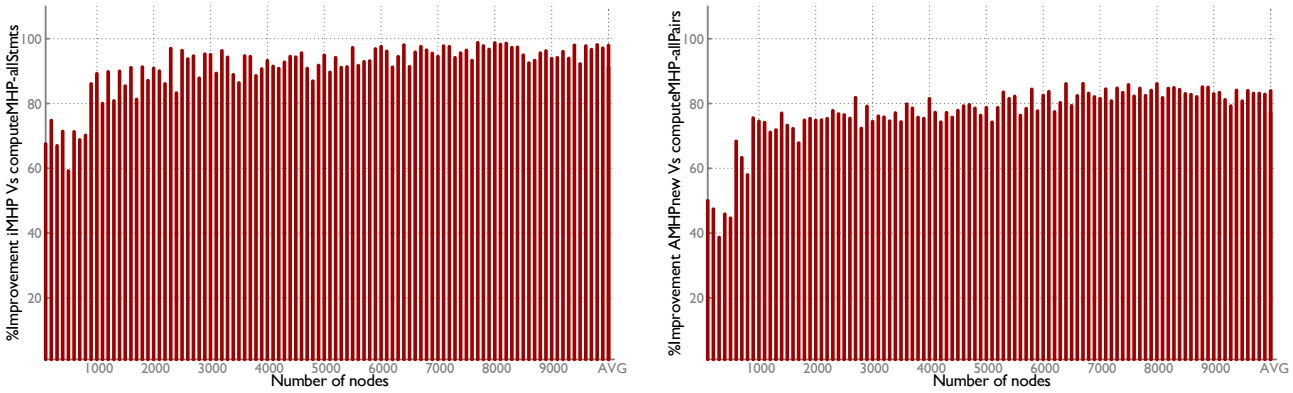
We now discuss the impact of the PST size (or in practice, the program size) on the effectiveness of our proposed algorithms. We fix the percentage number of `asyncs` to 5% (indicating a moderately parallel program), and the rest of the PST non-leaf nodes are equally distributed. We then varied the number of PST nodes from 100 to 10,000 (in increments of 100) and measured the performance. It may be noted that the use of the generated PSTs helps us conduct such a study, which is otherwise not possible using real world X10 programs (due to non-availability of such wide variety of X10 programs in public domain).

Figure 13 presents an evaluation of our proposed algorithms for varying number of PST nodes. Like in the previous section, we use our proposed algorithms to answer the two auxiliary challenges (1 and 2). Figure 13(a) presents a comparison of the iMHP algorithm and the `computeMHP-allStmts` function, in answering the auxiliary challenge 1. Our proposed approach leads to significant improvements (approximately between 59% to 98% improvement, and on average 91.24%). It can be seen that for very small PSTs, the improvements are slightly low (between 60% to 70%); with increasing the number of nodes the gains increase sharply. This is natural, considering the significant difference (nearly $O(N^2)$) in the complexity (see Figure 8), between the two techniques.

Figure 13(b) presents a comparison of the aMHP$_{new}$ algorithm and the `computeMHP-allPairs` function, in answering the auxiliary challenge 2. We note that improvements due to our proposed approach are between approximately 39% to 86% improvement (on average 77.27%). Like before, it can be seen that for small PSTs ($\leq$ 500 nodes), the improvements are low (between 39% to 60%). And with increasing number of nodes the gains increase to around 75-80%. Note that, compared to the analysis of auxiliary challenge 1, the improvements here are slightly less. This is consistent with the improvements in complexity (over the methods using the algorithm of Agarwal et al.) achieved by iMHP (nearly a factor of $O(N^2)$) and aMHP$_{new}$(a factor of $O(N)$).

## 6. Conclusion and Future Work

In this paper, we present new approaches to do May-Happen-in-Parallel (MHP) analysis for task parallel languages (such as X10 and HJ) that support async-finish-atomic parallelism. We present a fast incremental MHP algorithm to derive all the statements that may run in parallel with an input statement. We also extend the MHP algorithm of Agarwal et al. [1] (that answers if two given statements may run in parallel) to improve the computational complexity (from




**(a)** auxiliary challenge 1: iMHP Vs `computeMHP-allStmts`

**(b)** auxiliary challenge 2: $\mathsf{aMHP}_{new}$ Vs `computeMHP-allPairs`

**Figure 13:** Impact of the proposed techniques on PSTs with varying number of nodes

worst case quadratic to linear in program size), without compromising on the precision. We demonstrate the efficiency of our proposed MHP analysis techniques empirically on (i) actual benchmarks and (ii) a large representative set of automatically generated program-structure-trees (PSTs). Considering the pivotal role played by MHP analysis in many static and dynamic program optimizations/analyses, results shown in this paper will also have a positive effect on the speed and effectiveness of those optimizations/analyses.

Clocks in X10 [24] and Phasers in HJ [11] present newer challenges to compute MHP information and extending our proposed incremental MHP algorithm to take into consideration clocks and phasers is left as an interesting future work.

## Acknowledgements

This work is partially supported by the New Faculty Seed Grant, IIT Madras CSE/11-12/567/NFSC/NANV, DAE research grant 2012/36/54-BRNS/2943 and DST Fasttrack grant SB/TP/ETA-166/2012.

## References

[1] AGARWAL, S., BARIK, R., SARKAR, V., AND SHYAMASUNDAR, R. 2007. May-happen-in-parallel analysis of X10 programs. In *Proceedings of PPoPP*. 183–193.

[2] ALBERT, E., FLORES-MONTOYA, A., GENAIM, S., AND MARTIN-MARTIN, E. 2013. Termination and cost analysis of loops with concurrent interleavings. In *ATVA*. 349–364.

[3] ALSTRUP, S., THORUP, M., GØRTZ, I. L., RAUHE, T., AND ZWICK, U. 2014. Union-find with constant time deletions. *ACM Transactions on Algorithms*, 6.

[4] BARIK, R. 2005. Efficient Computation of May-Happen-in-Parallel Information for Concurrent Java Programs. In *Proceedings of LCPC*. 152–169.

[5] CHEN, C., HUO, W., AND FENG, X. 2012. Making it practical and effective: fast and precise may-happen-in-parallel analysis. In *Proceedings of PACT*. 469–470.

[6] CORMEN, T. T., LEISERSON, C. E., AND RIVEST, R. L. 1990. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA.

[7] DUESTERWALD, E. AND SOFFA, M. 1991. Concurrency analysis in the presence of procedures using a data-flow framework. In *Proceedings of the symposium on Testing, analysis, and verification*. ACM, 36–48.

[8] FLORES-MONTOYA, A., ALBERT, E., AND GENAIM, S. 2013. May-Happen-in-Parallel Based Deadlock Analysis for Concurrent Objects. In *Proceedings of FORTE*. 273–288.

[9] GUO, Y., BARIK, R., RAMAN, R., AND SARKAR, V. 2009. Work-first and help-first scheduling policies for async-finish task parallelism. In *Proceedings of IPDPS*. IEEE Computer Society, 1–12.

[10] GUPTA, S. AND NANDIVADA, V. K. 2015. IMSuite: A Benchmark Suite for Simulating Distributed Algorithms. *Journal of Parallel and Distributed Computing 75,* 0, 1 – 19.

[11] HABANERO. 2009. Habanero Java. http://habanero.rice.edu/hj.

[12] KAPLAN, H., SHAFRIR, N., AND TARJAN, R. E. 2002. Union-find with deletions. In *Proceedings of SODA*. 19–28.

[13] KRINKE, J. 1998. Static slicing of threaded programs. In *Proceedings of PASTE*. ACM, New York, NY, USA, 35–42.

[14] LEE, J. K. AND PALSBERG, J. 2010. Featherweight X10: a core calculus for async-finish parallelism. In *Proceedings of SAS*. 25–36.

[15] LEE, J. K., PALSBERG, J., MAJUMDAR, R., AND HONG, H. 2012. Efficient May Happen in Parallel Analysis for Async-Finish Parallelism. In *Proceedings of SAS*. 5–23.

[16] LIN, L. AND VERBRUGGE, C. 2004. A Practical MHP Information Analysis for Concurrent Java Programs. In *Proceedings of LCPC*. 194–208.

[17] MASTICOLA, S. P. AND RYDER, B. G. 1991. A model of Ada programs for static deadlock detection in polynomial times. In *workshop on Parallel and distributed debugging*. ACM, 97–107.

[18] MASTICOLA, S. P. AND RYDER, B. G. 1993. Non-concurrency analysis. In *Proceedings PPoPP*. ACM, New York, NY, USA, 129–138.

[19] MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

[20] NANDIVADA, V. K., SHIRAKO, J., ZHAO, J., AND SARKAR, V. 2013. A Transformation Framework for Optimizing Task-Parallel Programs. *ACM Trans. Program. Lang. Syst. 35,* 1, 3:1–3:48.

[21] NAUMOVICH, G., AVRUIN, G. S., AND CLARKE, L. A. 1998. Data Flow Analysis for Checking Properties of Concurrent Java Programs. Tech. rep., Amherst, MA, USA.

[22] NAUMOVICH, G. AND AVRUNIN, G. S. 1998. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *Proceedings of FSE*. 24–34.

[23] NAUMOVICH, G., AVRUNIN, G. S., AND CLARKE, L. A. 1999. An efficient algorithm for computing MHP information for concurrent Java programs. In *Proceedings of ESEC/FSE*. 338–354.

[24] SARASWAT, V., BARD, B., IGOR, P., TARDIEU, O., AND GROVE, D. 2012. X10 Language Specification Version 2.3. Tech. rep., IBM.

[25] SCHIEBER, B. AND VISHKIN, U. 1988. On finding lowest common ancestors: simplification and parallelization. *SIAM J. Comput. 17,* 6, 1253–1262.

[26] TAYLOR, R. N. 1983. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica 19,* 1, 57–84.