On the fly MHP Analysis

Sonali Saha
IIT Madras, India
sonali.saha.hs@gmail.com

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

Abstract

May-Happen-in-Parallel (MHP) analysis forms the basis for many problems of program analysis and program understanding. MHP analysis can also be used by IDEs (integrated-development-environments) to help programmers to refactor parallel-programs, identify racy programs, understand which parts of the program run in parallel, and so on. Since the code keeps changing in the IDE, re-computing the MHP information after every change can be an expensive affair. In this manuscript, we propose a novel scheme to perform incremental MHP analysis (on the fly) of programs written in task parallel languages like X10 to keep the MHP information up to date, in an IDE environment.

The key insight of our proposed approach to maintain the MHP information up to date is that we need not rebuild (from scratch) every data structure related to MHP information, after each modification (addition or deletion of statements) in the source code. The idea is to reuse the old MHP information as much as possible and incrementally recompute the MHP information (of a small set of statements) which depends on the statement added/removed. We introduce two new algorithms that deal with addition and removal of parallel constructs like `finish`, `async`, `atomic`, and sequential constructs like `loop`, `if`, `if-else` and other sequential statements, on the fly. Our evaluation shows that our algorithms run much faster than the repeated invocations of the fastest known MHP analysis for X10 programs [Sankar et al. 2016].

CCS Concepts • **Software and its engineering Incremental compilers**; *Parallel programming languages*; • **Theory of computation Program analysis**;

Keywords Concurrent programs, may happen in parallel analysis, incremental analysis

ACM Reference Format:
Sonali Saha and V. Krishna Nandivada. 2020. On the fly MHP Analysis. In *25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '20), February 22–26, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3332466.3374541

PPoPP '20, February 22–26, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-6818-6/20/02…$15.00
https://doi.org/10.1145/3332466.3374541

1 Introduction

In parallel languages, May-Happen-in-Parallel (MHP) analysis identifies what all statements may run in parallel with a given statement. MHP analysis forms the basis for many optimizations and program-analysis problems like data race detection, deadlock detection, debugging, and so on [Flores-Montoya et al. 2013; Masticola and Ryder 1991; Nandivada et al. 2013; Naumovich et al. 1999b]. In addition, MHP information can be used as a program understanding tool wherein the programmer (or a semi-automated tool) can learn what all statements may run in parallel with a given statement. Such a feature is especially useful in the context of IDEs (Integrated Development Environments) such as Eclipse [2017], X10DT [2017], IntelliJ [2017] and so on, where besides responding to the programmers queries on which statements may run in parallel with a given statement, the IDEs can use the MHP information to automatically mark statements that may lead to deadlocks and data-races [Vojdani and Vene 2009; Zhan and Huang 2016]). A popular MHP related question and its natural extension, typically asked in the above contexts are given below:

Key question. Given a statement s in a program P, compute $MHP(s)$ that returns the statements in P that may run in parallel with s.

Auxiliary question. Compute the MHP map for each statement in the program.

Many researchers [Agarwal et al. 2007; Barik 2005; Duesterwald and Soffa 1991; Lin and Verbrugge 2004; Masticola and Ryder 1993; Naumovich and Avrunin 1998; Naumovich et al. 1999a,b; Sankar et al. 2016] have studied the problem of May-Happen-in-Parallel (MHP) analysis in the context of many different parallel programming languages. Computing MHP analysis for all pairs of statements in a program has been shown to be undecidable by Taylor [1983] in his influential paper. Taylor also showed the problem to be NP-complete (assuming all the control flow paths are executable) for programs that admit synchronization primitives such as the *rendezvous* construct of Ada. However, the problem is more tractable, in the absence of such low-level synchronization primitives. Most of the research on MHP algorithms (including our paper) is based on these two conservative assumptions. Thus, if two statements S_1 and S_2 are marked as "may-not-run-in-parallel", then it is guaranteed that during the actual execution, S_1 and S_2 will never run in parallel.

```
S0:finish {
 S1:async {S2:...}
 S3:finish {
  S4:async{S5:...}
   S6:.../*Added*/
 }
}
```

Figure 1. Sample code

However, if the analysis answers that S_1 and S_2 "may-run-in-parallel", then during the actual execution, S_1 and S_2 may or may not run in parallel.

Naumovich et al. [1999a,b] answer the key question in the context of Java. Many prior works [Agarwal et al. 2007; Chen et al. 2012; Lee and Palsberg 2010; Lee et al. 2012] present techniques to compute the MHP information for pairs of statements (see the exposition of Sankar et al. [2016], for a detailed discussion on the two types of analyses).

The best-known solution to answer these questions (including the key question and auxiliary question) is given by Sankar et al. [2016]; their approach takes a worst-case time complexity of $O(N^2)$ – significantly less than the other existing techniques (for example, using the technique of Agarwal et al. [2007]). For scenarios where the program continuously changes (for example, in IDEs), the naive approach to maintain up to date MHP information is to repeatedly invoke a fast MHP analysis (such as the `iMHP` algorithm of Sankar et al. [2016]). However, as the code keeps changing, continuously re-building the MHP information (from scratch) after every change can be quite expensive, especially when the code size is not too small. We explain the same and an intuition for the scope for improvement using an example.

Figure 1 shows a sample synthetic X10 [Saraswat et al. 2015] code. In X10, an `async` construct is used to create an asynchronous task. The `finish` construct waits for all the tasks created in its body to terminate (see Section 2 for details on X10 syntax). The *MHP* map before adding statement `S6` is as shown below.

Stmt s	S0	S1	S2	S3	S4	S5
$MHP(s)$	{}	{}	{S3,S4,S5}	{S2}	{S2}	{S2}

After adding `S6`, to recompute the MHP information, we could invoke the `iMHP` algorithm, which will recompute the MHP information for all the statements from scratch. It can be seen that doing such a full re-computation for each possible change in the source code can be expensive and is avoidable. We observe that a single change in the source code does not always drastically modify the *MHP relation* between every pair of statements. And importantly, we can reuse the old MHP information and compute MHP information of specific statements that depends on the statement added or removed. For example, after adding the statement `S6`, the existing MHP relationship among the existing statements does not change; hence, we can reuse that part of the MHP information. In addition to that, we have to compute the MHP information for `S6` and add `S6` to the MHP information of those statements that may run in parallel with statement `S6`; these modifications lead to the following *MHP* map:

Stmt s	S0	S1	S2	S3
$MHP(s)$	{}	{}	{S3,S4,S5,S6}	{S2}
Stmt s	S4	S5	S6	
$MHP(s)$	{S2}	{S2,S6}	{S2,S5}	

As can be seen, the changes are mostly incremental except for MHP(S6). Thus, if we can come up with an approach to update data structures incrementally, after each code change, it may save a lot of time. The exact changes to the MHP map will naturally vary depending on the type of the construct/statement being added/removed. In this manuscript, we present a scheme that only needs to incrementally update the MHP information for a subset of the program statements on the fly, without having to fully recompute the MHP information after each code change, while ensuring that the worst-case cost of update is not more than the cost of the fast iMHP algorithm proposed by Sankar et al. [2016]. In practice, our on-the-fly algorithm runs much faster. Such an on-the-fly MHP analysis can improve the performance of IDEs that need to maintain up-to-date MHP information. To the best of our knowledge, there have been no prior works that compute incremental MHP information on-the-fly, especially in the context of task parallel languages like X10.

In this manuscript, we present our on-the-fly MHP computation algorithms in the context of X10 programs, by covering the complete set of parallel constructs supported by Sankar et al. [2016], namely `async`, `finish`, and `atomic`. We believe that the presented results can be extended to other task-parallel languages like HJ [Habanero 2009], Chapel [Chamberlain et al. 2007], and so on. These techniques can also be used for the subsets of programs written in languages like Java or Cilk [Leiserson 2009], where the thread/task creation and termination follow a "tree" structure like X10.

Contributions.

- We propose new algorithms to incrementally compute MHP information, on the fly, after each change to an X10 program.
- We have implemented these algorithms as a part of X10DT as a plug-in to perform on the fly MHP analysis.
- We present an evaluation of our on-the-fly MHP computation algorithms that shows their effectiveness over the state of the art `iMHP` algorithm [Sankar et al. 2016].

2 Background

Now we present a brief discussion on the relevant background about X10 language and the program-structure-tree over which the proposed algorithms are defined.

2.1 X10 Language

We describe a subset of X10 language over which we define our on-the-fly analysis. The interested reader may go over the X10 language manual [Saraswat et al. 2015] for details. The statements in this subset can be derived from the following grammar covering the three parallel constructs `async`, `finish`, and `atomic`; and the serial constructs such as assignments, declarations, expressions, method invocations, conditional-statements, loops, and so on.

`S` ::= `async S` | `atomic S` | `finish S` | *seq*(`S`)

Here, *seq*(X) is used to denote the programs formed from X, by closing under the sequential constructs. The statement `async S`, creates an asynchronous task that may run in parallel with the parent task. The statement `finish S`, acts as a join point and waits for all the tasks created in the body `S` of the `finish` statement. At runtime, each X10 instruction has a unique associated task, and each task has a unique *immediately enclosing finish* (in short, IEF). Each X10 program contains an implicit *main* task and this task is enclosed inside an implicit outermost finish. The statement `atomic S` realizes a global critical section. An `atomic` block may not invoke an `async` or `finish` statement.

Guo et al. [2009] define an `async` statement S' as an escaping-async (*e-async*) of a statement S, if S' is contained within S, and S' is not enclosed in a finish statement within S. Thus, the immediately-enclosing-finish (IEF) of S' is not present within S.

2.2 Program Structure Tree

We use an extension of the abstraction of Program Structure Tree (PST) proposed by [Agarwal et al. 2007]. PST is a compressed representation of Abstract Syntax Tree (AST). The nodes in our PST correspond to three parallel constructs `async`, `finish`, and `atomic`; and four serial constructs `loop`, `if`, `if-else`, and `seq-stmt`. The `seq-stmt` nodes represent all the sequential statements, except loops, and conditionals. As an example, Fig. 2 shows the extended PST (hereafter, simply referred to as PST) for the program shown in Fig. 1.

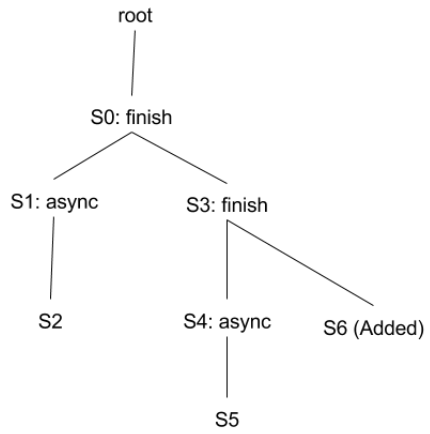


Figure 2. PST for the code shown in Fig. 1

```
1  Function driver(PST P, Node L)
2  begin
3      switch type of L do
4          case finish-node do addFinish(P, L);
5          case async-node do addAsync(P, L);
6          case atomic-node do addAtomic(P, L);
7          case loop-node do addLoop(P, L);
8          case if-else-node do addIfElse(P, L);
9          case if-node do addIf(P, L);
10         case Otherwise do addSeqStmt(P, L);
```

Figure 3. On-the-fly MHP-driver; adding PST nodes

3 On-the-fly MHP Analysis

In this section, we discuss our proposed techniques to compute (and maintain) MHP information, as the program changes as part of the program development life cycle, in IDE type of tools that need to maintain MHP information (after every significant code change). In contrast to the scheme of Sankar et al. [2016] that recomputes the MHP information for all the nodes from scratch (after each change), our scheme reuses the current MHP information to incrementally compute the MHP information after each change.

For the ease of presentation, we divide our proposed techniques into two parts depending on how the PST nodes (constructs, statements) change: addition of nodes (Section 3.1) and deletion of nodes (Section 3.2). Modification of nodes can be handled by considering them as a sequence of deletion(s) followed by addition(s).

3.1 On-the-fly MHP analysis: addition of nodes

We now present our scheme to handle the addition of PST nodes. Depending on the specific node being added, the impact on the MHP information of the existing PST nodes and the added PST nodes varies. For the ease of presentation, in this Section, we only focus on the program updates that add new nodes to the PST; in Section 3.2, we handle the deletion of PST nodes.

Figure 3 presents the driver algorithm that is invoked after each significant change performed by the programmer; see

```
1  Function addFinish(PST P, Node F)
2  begin
3      Node fc = firstChild(F);// has only one child.
4      Set T = MHP(fc);// Initial MHP info of F
5      if !hasLoopAncestorFollowedByAsync(F) then
           // MHP information of the descendants
              of F may change
6          Stmt E = F.parent;
7          while E ≠ root do
8              if E is an async node OR a finish node
                 then break ;
9              E = E.parent();
10         Set M = descendants of E − descendants of
             e-async of E after which F may run;
11         Set S = ∅;
12         for l ∈ descendants(F) do
13             if l is a descendent of an e-async of F then
14                 MHP(l) = MHP(l) − M; S = S ∪ {l};
15         for m ∈ M do
16             MHP(m) = MHP(m) − S;
17         if hasLoopAncestor(F) then
18             for x ∈ async descendants of F do
                   // MHP info of the async node x
                      & its children may be corrupt
19                 addAsync(P, x); // Recompute.
20         T =T − S;
21     computeSelfMHP(fc, F, T);
```

Figure 4. On-the-fly MHP: add `finish`-node

Section 4 on how we handle cases involving multiple significant changes. The driver algorithm takes two arguments (the new PST P and the node L in P that being added) and calls the appropriate routines based on the specifics of L, which in turn incrementally update the MHP information on the fly. We now explain each of these routines independently.

3.1.1 On-the-fly MHP analysis: adding `finish` node

Figure 4 presents the algorithm that is invoked when a `finish` node is added. It is based an observation that if the added `finish` node (F) has an `async` node (say, AA) as an ancestor, which in turn has a `loop` ancestor (say, LA), such that there is no `finish` node in the path from LA to AA, then the insertion of the `finish` node does not alter the MHP information of the nodes of P, except the ones with which the `finish` node F may run in parallel. The negation of this condition is checked at Line 5, and if it succeeds, we find the immediately enclosing `finish` or `async` (E) (Lines 6-9). If F does not have any loop ancestor, the descendants of the

```
1  Function computeSelfMHP(Node fc, Node L, Set T)
2  begin
3      MHP(L) = MHP(L) ∪ T;
4      for x ∈ T do MHP(x) = MHP(x) ∪ {L} ;
5      if fc ∈ MHP(fc) then MHP(L) = MHP(L) ∪ {L} ;
```

Figure 5. MHP information update of the added node

`e-asyncs` of F cannot run in parallel with any of the descendants of E that may run after F, except for the children of `e-asyncs` of E, after which F may run. We update the MHP information of all these nodes accordingly (Lines 10-16).

However, when F has one or more loop-ancestors then MHP information of the statements inside any possible `async`-nodes below F may have to be re-computed. For example, say F has a `loop` ancestor, and the body of F has two `e-asyncs` *async*1 and *async*2. Say, *async*1 has a child $S1$ and *async*2 has a child $S2$. Now because of the code in Lines 12-16, *async*2 and $S2$, which are reachable from E, are removed from the MHP information of $S1$; this needs to be corrected. So, we recompute the MHP information for the descendants of each `async` inside F (Lines 17-19).

Finally, the MHP information of F has to be updated: F can run in parallel with the statements running in parallel with the first child of F and vice versa. Further, if the first child of F is running in parallel with itself, then F can also run in parallel with itself. These updates are done by calling the function `computeSelfMHP` (Figure 5).

3.1.2 On-the-fly MHP analysis: adding `async` node

Figure 6 presents the algorithm that is invoked when an `async` node is added. First, we find the immediately enclosing `finish` (F, Line 4). We use the logic behind the iMHP-addAsync routine of Sankar et al. [2016]: after adding A, any of the descendants of A (in the set D_A) may run in parallel with any of the descendants of F that may execute after A. We extend the routine further to handle the possible presence of `atomic` blocks in the code.

As per the X10 language semantics, no two `atomic` blocks executing at the same place [Saraswat et al. 2015], may run in parallel. We update the MHP information of members of D_F and D_A accordingly (Lines 10-15).

3.1.3 On-the-fly MHP analysis: adding `atomic` node

Given a PST, the iMHP scheme of Sankar et al. [2016] first removes all the concurrency related constructs (`finish`, `async`, and `atomic`) and computes the complete MHP information by re-introducing the concurrency related constructs in a fixed order: `finish`, `async`, and finally the `atomic` nodes. Since their scheme adds atomics at the end (using the routine iMHP-addAtomic), we can use the same scheme to compute on-the-fly MHP analysis, on adding an `atomic` statement.

```
1  Function addAsync(PST P, Node A) begin
2      Node fc = fc(A);
3      Set T = MHP(fc);
4      F = IEF(A);
5      Set M = ∅, P = ∅, Q = ∅, S = ∅;
6      D_F = Descendants(F) reachable from A;
       // does not have the descendants of A
7      M = Members of D_F not inside any atomic block;
8      D_A = Descendants(A);
9      S = Members of D_A not inside any atomic block;
10     for m ∈ M do MHP(m) = MHP(m) ∪ D_A;
11     for p ∈ (D_F − M) do // p is in Atomic
12         MHP(p) = MHP(p) ∪ S
13     for s ∈ S do MHP(s) = MHP(s) ∪ D_F;
14     for q ∈ (D_A − S) do
           // q: A's descendants in Atomic
15         MHP(q) = MHP(q) ∪ M
16     computeSelfMHP(fc, A, T);
```

Figure 6. On-the-fly MHP algorithm: add async node

3.1.4 On-the-fly MHP analysis: adding a `loop` node

Figure 7 presents the algorithm that is invoked when a loop node L is added. When we add L, the descendants of the `e-asyncs` of L can run in parallel with all the descendants of L. Note: (i) If the loop L is inside an `atomic` block, then it is enough to invoke `computeSelfMHP`, (ii) the `atomic` blocks present inside L are handled in a manner similar to that done for `addAsync`.

3.1.5 On-the-fly MHP analysis: adding `if` nodes

Figure 8 presents the algorithm that is invoked when an `if` node L is added. Since statically we do not know value of the if-condition, we will assume the addition of the if-node does not change the reachability of its body. Hence, it is sufficient to just invoke `computeSelfMHP`.

Figure 9 presents the algorithm that is invoked when an `if-else` node L is added. If the first-child of L (the then-statement) may run in parallel with itself (indicating that this statement has a loop-ancestor whose iterations may run in parallel with each other) then it is enough to just invoke `computeSelfMHP` to update the MHP information for L. Similarly, if the first-child of L is not reachable from itself (indicating that L is not inside any loop), or if the IEF of L is a descendant of the immediately-enclosing-loop of L then none of the descendants of the `e-asyncs` inside the then-part of L can run in parallel with any node inside the else-part of L. A similar argument can be made about the else-part of L. We update the MHP information of all these nodes accordingly (Lines 5-11). Note that in the condition at Line 4, if fc is not reachable from itself $\Rightarrow$ IEL(L) $\neq$ NULL.

```
1  Function addLoop(PST P, Node L)
2  begin
3      Node fc = firstChild(L); // loop-body
4      Set T = MHP(fc);
5      if !inAtomic(L) then
6          D_L = Descendants (L);
7          D_ea = members of D_L that are descendants of
            e-async nodes;
8          D_nea = D_L − D_ea; // non descendants of
               e-async nodes in D_L
9          D_at = descendants atomic nodes of L;
10         D_nat = D_L − D_at; // non-atomic
               descendants of L
11         for a ∈ (D_ea − D_at) do // non-atomic +
            escaping descendants
12             MHP(a) = MHP(a) ∪ D_L;
13         for a ∈ (D_ea − D_nat) do // atomic +
            escaping descendants
14             MHP(a) = MHP(a) ∪ (D_L − D_at);
15         for a ∈ (D_nea − D_at) do //non-atomic +
            non-escaping descendants
16             MHP(a) = MHP(a) ∪ D_ea;
17         for a ∈ (D_nea − D_nat) do // atomic +
            non-escaping descendants
18             MHP(a) = MHP(a) ∪ (D_ea − D_at);
19     computeSelfMHP(fc, L, T);
```

Figure 7. On-the-fly MHP: add loop node

```
1  Function addIf(PST P, Node L)
2  begin
3      Node fc = firstChild(L);
4      T = MHP(fc); // then-branch
5      computeSelfMHP(fc, L, T);
```

Figure 8. On-the-fly MHP analysis: add `if` node

3.1.6 On-the-fly MHP analysis: adding seq stmt nodes

Figure 10 presents an algorithm that is invoked when a sequential statement node L is added. If L has a successor (*next*), then L may run in parallel with all the statements that are running in parallel with *next*. Else, if L has a predecessor (*prev*), then L may run in parallel with all the nodes that are running in parallel with *prev*, along with the descendants of any `e-async` nodes of *prev* (Lines 7-9). Note that if a statement has both successor(s) and predecessor(s) then both the codes in Lines 3-5 and Lines 7-9 lead to the same answer.

```
1  Function addIfElse(PST p, Node L)
2  begin
3      Node fc = firstChild(L);
4      T = MHP(fc);
5      if (fc ∉ MHP(fc)) AND (fc is not reachable from fc OR
         IEF(L) is a descendant of IEL (L)) then
           // IEL:immediately-enclosing-loop
6          Node L1 = firstChild(L); // then-branch
7          Node L2 = secondChild(L); // else-branch
8          Set S1 = Descendants of the e-asyncs of L1;
9          Set S2 = Descendants of the e-asyncs of L2;
10         for s ∈ S1 do MHP(s) = MHP(s) − S2 ;
11         for s ∈ S2 do MHP(s) = MHP(s) − S1 ;
12     computeSelfMHP(fc, L, T);
```

Figure 9. On-the-fly MHP analysis: add `if-else` node

```
1  Function addSeqStmt(PST P, Node L) begin
2      if (next(L) != null) then
3          Node next = next(L);
4          C = MHP(next);
5          computeSelfMHP(next, L, C);
6      else if (prev(L) != null) then
7          Node prev = prev(L);
8          C = MHP(prev) ∪ descendants of the e-asyncs
             of prev;
9          computeSelfMHP(prev, L, C);
10     else MHP(L) = ∅ ; // First stmt in the proc
```

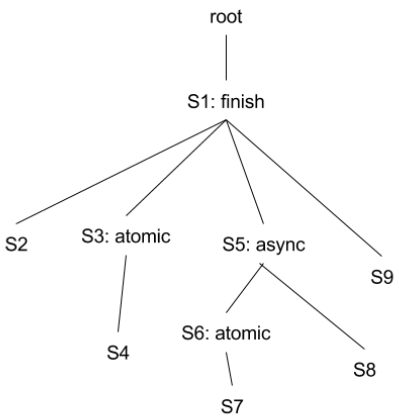
Figure 10. On-the-fly MHP analysis: add seq-stmt

3.1.7 Complexity and Efficiency

We now discuss the time complexity of add* procedures discussed earlier. We use N to denote the total number of nodes in the PST, α to denote the inverse Ackermann function [Cormen et al. 2001], A to denote the total number of async nodes in the PST, C to denote the total number of concurrency-related nodes ($C \geq A$) in the PST. The add* procedures invoke set operations like union, find an element in a set, delete an element from a set, and so on. Each of these set operations can be done in near constant time $O(\alpha(N))$ [Alstrup et al. 2014; Cormen et al. 2001; Kaplan et al. 2002]. The procedures addAsync, addLoop, addIf, addIfElse, and addSeqStmt invoke these set operations at most N times. Thus, the time complexity for each these procedures is $O(N \times \alpha(N)) \approx O(N)$. Note that the addFinish procedure may call addAsync procedure at most A times; hence the time complexity of addFinish is $O(A \times N)$.

```
S1: finish{
 S2: . . .
 S3: atomic{
  S4: . . .
 }
 S5: async{
  S6: atomic{
   S7: . . .
  }
  S8: . . .
 }
 S9: . . .
}
```

(a)



(b)

Stmt	MHP	Stmt	MHP	Stmt	MHP	Stmt	MHP
S1	{}	S2	{}	S3	{}	S4	{}
S5	{}	S6	{S9}	S7	{S9}	S8	{S9}
S9	{S6, S7, S8}						

(c)

Figure 11. On-the-fly MHP analysis example. (a) initial code, (b) initial PST, (c) initial MHP.

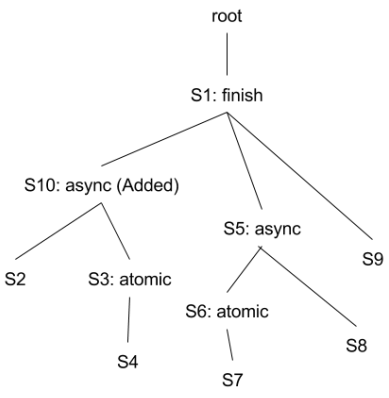
We can see that in contrast to the scheme of invoking iMHP routines [Sankar et al. 2016] (complexity $O(C \times N)$), after every update, there is a linear time improvement for most of the updates (addition of async, loop, if, if-else, and sequence nodes). And in case of addition of finish nodes, in the worst-case addFinish has the same complexity as iMHP [Sankar et al. 2016], but in practice, it takes much less time, as $A \ll C$, and we avoid recomputing of MHP information for a large portion of the PST.

3.1.8 Example

To explain the above algorithms we use the code in Fig. 11a as the initial program. Fig. 11b shows the corresponding PST, and Fig. 11c shows the initial MHP map. Say, we first introduce an async construct S10 around statements S2 and S3. The resulting code is shown in Fig. 12a, and Figure 12b represents the corresponding PST. The first child of S10 = S2, and MHP(S2) = {}. Following the steps in Fig. 6, before Line 10, F = IEF(S10) = {S1}, M = {S5, S6, S8, S9}, D_A(S10) = {S2, S3, S4}, S = {S2, S3}. After processing loop at line 10, we update MHP(S5) = {S2, S3, S4}, MHP(S6) = MHP(S8) = {S2, S3, S4, S9}, and MHP(S9) = {S2, S3, S4, S6, S7, S8}. After processing the loop at 11, we update MHP(S7) = {S2, S3, S9}. After processing the loop at 13, we update MHP(S2) = MHP(S3) = {S5, S6, S7, S8, S9}. After processing the loop at 14, we update MHP(S4) = {S5, S6, S8, S9}.

```
S1: finish{
 S10: async{
  S2: ...
  S3: atomic{
   S4: ...
 } }
 S5: async{
  S6: atomic{
   S7: ...}
  S8: ...}
 S9: ...}
```

(a)



(b)

Stmt	MHP	Stmt	MHP
S1	{}	S2	{S5, S6, S7, S8, S9}
S3	{S5, S6, S7, S8, S9}	S4	{S5, S6, S8, S9}
S5	{S2, S3, S4}	S6	{S2, S3, S4, S9}
S7	{S2, S3, S9}	S8	{S2, S3, S4, S9}
S10	{}	S9	{S2, S3, S4, S6, S7, S8}

(c)

Figure 12. Example after adding S10. (a) modified code, (b) modified PST, (c) updated MHP.

After that, say we introduce (i) an `if-else` statement S11, with S10 in the then-part and S5 in the else-part, (ii) a `loop` construct S12 around S11 and S9, and (iii) an `if` statement S13 around S12 – in that order. The resulting PSTs are shown in Fig. 13, and the updated MHP maps after each statement are shown in Fig. 14. As it can be seen, after each code insertion, the number of updates to the MHP map are few. The final MHP-map is shown in Figure 14d.

3.2 On-the-fly MHP analysis: on removal of nodes

In Section 3.1, we discussed how the MHP information is incrementally computed on the fly, after each addition to the PST. Now, we discuss how to incrementally compute the MHP information on-the-fly, on the deletion of PST nodes. When a PST node n is removed, all its children nodes get attached to the parent of n in the PST.

The driver algorithm (skipped for space) on removing a statement is similar to that shown in Fig. 3, except that this driver invokes the routines for handling the statement removal. We now explain each of these routines independently.

3.2.1 On-the-fly MHP analysis on removing a simple sequential statement or an if-statement

Figure 15 shows the algorithm to be invoked when a sequential-statement node is removed; handling of the removal of `if`-nodes is exactly similar. When we remove a simple sequential-statement (for example, an assignment statement), we just need to remove the entry of L from the MHP map and remove L from the MHP information of the statements that may run in parallel with L, if L would not have been removed.

3.2.2 On-the-fly MHP analysis on removing `finish`

Figure 16 presents the algorithm that is invoked when a `finish` node (F) is removed. If E is the IEF (immediately-enclosing-finish) of F, then after removing F, any descendant of the `e-async` nodes of F may run in parallel with any of the descendants of E that are reachable from F (and vice-versa). The `atomic` blocks present inside L are handled in a manner similar to that done for `addAsync`. Finally, we remove the MHP maps related to the `finish` node.

3.2.3 On-the-fly MHP analysis on removing `async`

Figure 17 presents the algorithm that is invoked when an `async` node (A) is removed from the PST. If the first-child of A may run in parallel with itself (indicating that this statement has a loop-ancestor whose iterations may run in parallel) then it is enough to just invoke `remSeqStmt` to update the MHP information for A. Otherwise, we first identify R, the immediately enclosing `finish` or immediately enclosing `async` of A. Note: an `async` node cannot have an `atomic` node as parent. The only other parents of interest are Loop, `finish` and `async` nodes. We handle all the three. Since the MHP recomputation related to statements inside the `finish` parent or `async` parent is the same (the impact is limited to the children of that parent, and the MHP nodes thereof) these two cases are clubbed together.

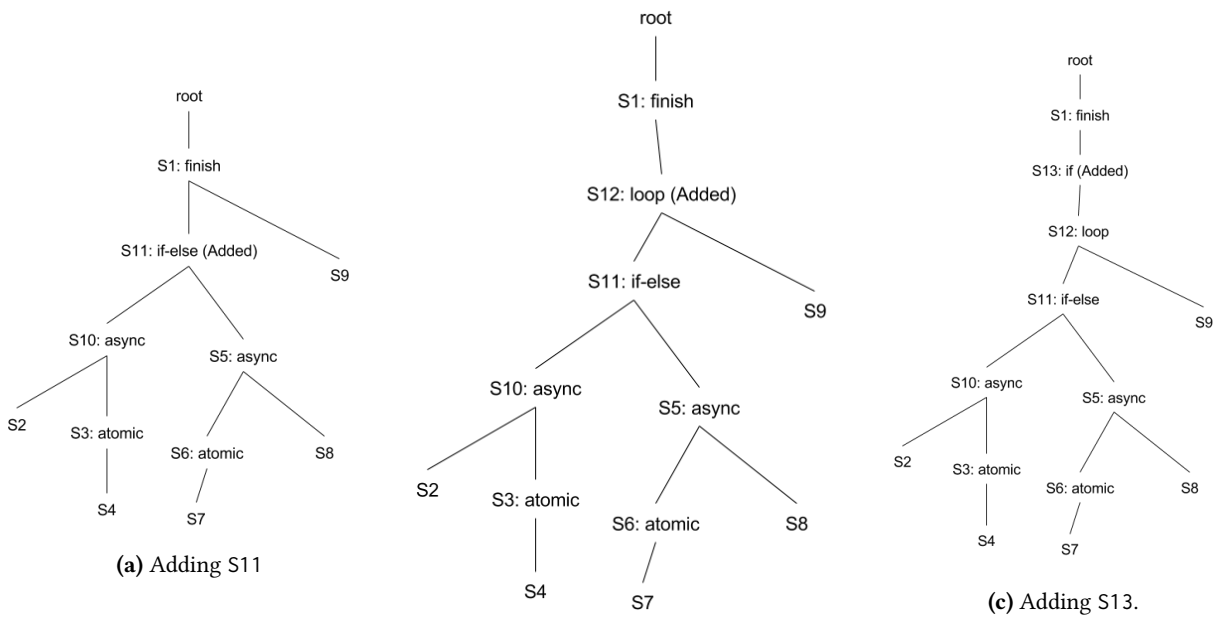
If A does not have any loop ancestor, none of the non escaping descendants of A can run in parallel with any of the descendants of R that may execute after A. However, when A has one or more loop-ancestors then the MHP information of the statements inside any `async`-nodes below L, which would have been removed because of the updates discussed in this procedure, has to be reverted back; we do so by invoking `addAsync` algorithm on these `async`-nodes.

3.2.4 On-the-fly MHP analysis on removing `atomic`

Figure 18 shows the algorithm that is invoked when an `atomic` node (L) is removed. In addition to removing the MHP information related to L, we have to update the MHP information related to the descendants of L: if A contains the set of nodes that may run in parallel with L, then the descendants of L can run in parallel with the nodes in A.

3.2.5 On-the-fly MHP analysis on removing `loop`

Figure 19 presents the algorithm that is invoked when a `loop` node (L) is removed. If the first-child of L may run in parallel with itself (indicating that this statement has a loop-ancestor whose iterations may run in parallel) or if L has an IEL (immediately-enclosing-loop) and IEF(L) is a descendant of IEL(L) then it is enough to just invoke `remSeqStmt` to update the MHP information for L. Otherwise, we first



(a) Adding S11

(b) Adding S12.

(c) Adding S13.

Figure 13. PSTs after adding (a) S11, (b) S12, and (c) S13.

Stmt	MHP
S2, S3, S4, S6, S7, S8	{S9}
S5, S11	{}

(a) Updates after adding S11

Stmt	MHP
S2, S3, S6, S8	{S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12}
S4, S7	{S2, S3, S5, S6, S8, S9, S10, S11, S12}
S5, S9, S10, S11, S12	{S2, S3, S4, S6, S7, S8}

(b) Updates after adding S12.

Stmt	MHP
S13	{}

(c) Updates after adding S13.

Stmt	MHP
S1, S13	{}
S2, S3, S6, S8	{S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12}
S4, S7	{S2, S3, S5, S6, S8, S9, S10, S11, S12}
S5,S9,S10,S11,S12	{S2, S3, S4, S6, S7, S8}

(d) Final MHP map after adding S10, S11, S12, and S13.

Figure 14. Updates to MHP after each insertion, shown in the PSTs in Fig. 13 and the final MHP maps.

aggressively update the MHP information of the descendants of L, by removing all the descendants of L from the escaping descendants of L, and all the escaping descendants of L from the non-escaping descendants of L. We then iterate over the async descendants of L and invoke *addAsync* on each one of them to take into consideration their impact on the MHP information of the descendants of L.

```
1 Function remSeqStmt(PST P, Node L)
2 begin
3     Set A = MHP(L);
4     for a ∈ A do MHP(a)=MHP(a) − {L};;
5     MHP.remove(L);// Removes L from MHP
```

Figure 15. On-the-fly MHP analysis: remove seq-stmt.

3.2.6 On-the-fly MHP analysis on removing `if-else`

Figure 20 presents the algorithm that is invoked when an `if-else` node is removed. Descendants of all escaping `async` nodes inside the `then` part of L can run in parallel with nodes inside the `else` part of L; note: the `e-asyncs` inside the `else` block do not have a similar impact on the statements of the `then` block. The `atomic` blocks present inside L are handled in a manner similar to that done for `addAsync`.

3.2.7 Time complexity and Efficiency

In this section, we discuss the time complexity of the `remove*` routines discussed earlier in this section. We use N to denote the total number of nodes in the PST, α to denote the inverse Ackermann function [Cormen et al. 2001], A to denote

```
1  Function remFinish(PST P, Node F)
2  begin
3      E = IEF of F;
4      M = ∅; // {a | a ∈ Descendants of E AND
           reachable from F AND descendants of
           e-asyncs in F}
5      B = ∅; // {a | a ∈ Descendants of E AND
           reachable from F AND ¬descendant of
           e-asyncs in F}
6      for a ∈ Descendants(E) do
7          if a is reachable from F then
8              if a is a descendant of an e-async of F then M =
                 M ∪ { a };
9              else B = B ∪ { a };
10     S = ∅; // Descendants of e-asyncs of F inside
           atomics
11     C = ∅; // Descendants of e-asyncs of F not
           inside atomics
12     for l ∈ descendants(e-asyncs of F) do
13         if !inAtomic(l) then S = S ∪ { l };
14         else C = C ∪ { l };
15     for m ∈ M do MHP(m) = MHP(m) ∪ S ∪ C;
16     for b ∈ B do MHP(b) = MHP(b) ∪ S;
17     for s ∈ S do MHP(s) = MHP(s) ∪ M ∪ B;
18     for c ∈ C do MHP(c) = MHP(c) ∪ M;
19     remSeqStmt (P, F)
```

Figure 16. On-the-fly MHP analysis: remove `finish`.

the total number of async nodes in the PST. Procedures `remFinish`, `remAtomic`, `remSeqStmt`, `remIfElse` perform some set operations like union, find an element in a set, and delete an element from a set. Each of these can be done in near constant time $O(\alpha(N))$ [Alstrup et al. 2014], [Cormen et al. 2001], [Kaplan et al. 2002]. These set operations are performed at most N times. So time complexity for these procedures is $O(N \times \alpha(N)) \approx O(N)$. The procedures `remLoop` and `remAsync` can call `addAsync` procedure at most A ($\ll N$) times, and hence the time complexity for these two procedures is $O(A \times N)$. Thus, we can see there is a linear time improvement over the iMHP algorithms of [Sankar et al. 2016] ($O(N^2)$)) for each of `remFinish`, `remAtomic`, `remSeqStmt`, `remIfElse`. But `remAsync` and `remLoop`, in the worst case, have the same complexity as iMHP [Sankar et al. 2016]; in practice, they take significantly less time, as we are reusing the MHP information and typically $A \ll N$.

3.2.8 Example

We start with the PST shown in Figure 13c, and remove `S13`, `S12`, `S11`, and `S10` in that order. The resulting updates to the MHP are as follows: (i) `S13`: MHP entry for `S13` gets removed. (ii) `S12`: The MHP entry for `S12` gets removed; the MHP maps of `S2`, `S3`, `S4`, `S6`, `S7`, `S8` get trimmed to contain

```
1  Function remAsync(PST P, Node A)
2      remSeqStmt(P, A)
3      if firstChild(A) ∉ MHP(firstChild(A)) then
4          Node R = A.parent();
5          while R ≠ root do
6              if R is an async node OR R is a finish node
                 then break;
7              R = R.parent();
8          Set M = ∅; S = ∅;
9          for r ∈ descendants(R) do
10             if r is reachable from A then M = M ∪ { r };
11         for l ∈ descendents(A) do
12             if l ∉ descendants of an e-async of A then
13                 MHP(l) = MHP(l) − M; S = S ∪ { l };
14         for m ∈ M do MHP(m) = MHP(m) − S;
15         if hasLoopAncestor(A) then
16             for ad ∈ async descendants of A; do
17                 addAsync(P, ad);
```

Figure 17. On-the-fly MHP analysis: remove `async`.

```
1  Function remAtomic (P: PST, L: Node)
2      Set A = MHP(L);
3      remSeqStmt(P, L)
4      B = Descendents(L);
5      for b ∈ B do MHP(b) = MHP(b) ∪ A;
6      for a ∈ A do MHP(a) = MHP(a) ∪ B;
```

Figure 18. On-the-fly MHP analysis: remove `atomic`.

```
1  Function remLoop(PST P, Node L)
2  remSeqStmt(P, L)
3  fc=firstChild(L);
4  if fc ∉ MHP(fc) AND ((IEL(L) = null) OR IEF (L) not a
     descendant of IEL (L)) then
5      M = escaping descendants of L;
6      S = non-escaping descendants of L;
7      for m ∈ M do
8          MHP(m) = MHP(m) − set of descendants of L;
9      for s ∈ S do MHP(s) = MHP(s) − M;;
10     for x ∈ set of async descendants of L do
11         addAsync(P, x);
```

Figure 19. On-the-fly MHP analysis: remove `loop`.

only `S9` and the MHP maps of `S5`, `S10`, `S11` are set to the empty set (Fig. 14a). (iii) `S11`: The MHP entry for `S11` gets removed; the MHP maps of `S2`, `S3`, `S4`, `S5`, `S6`, `S7`, `S8`, `S9` get updated to those shown in Fig. 12c. (iv) `S10`: The MHP entry for `S10` gets removed; the MHP maps of `S6`, `S7`, `S8` are set to

```
1  Function remIfElse(PST P, Node L)
2  begin
3      remSeqStmt(P, L)
4      Node L1 = firstChild(L); // then-branch
5      Node L2 = secondChild(L); // else-branch
6      Set M = ∅; P = ∅;
7      if L1 is an atomic node then P = P ∪ L1;
8      else M = M ∪ L1;
9      for l ∈ Descendents (L1) do
10         if l is a descendent of an escaping async of L1 then
11             if inAtomic(l) then P = P ∪ { l };
12             else M = M ∪ { l };
13     Q = ∅; S = ∅;
14     for l ∈ Descendents(L2) do
15         if inAtomic(l) then Q = Q ∪ { l };
16         else S = S ∪ { l };
17     for i ∈ M do MHP(i) = MHP(i) ∪ S ∪ Q;
18     for i ∈ P do MHP(i) = MHP(i) ∪ S;
19     for i ∈ S do MHP(i) = MHP(i) ∪ M ∪ P;
20     for i ∈ Q do MHP(i) = MHP(i) ∪ M;
```

Figure 20. On-the-fly MHP analysis: remove `if-else`

S9; the MHP maps of S9 are set to S6, S7, S8 and the MHP maps of S2, S3, S4, S5 are set to the empty set.

4 Discussion

Correctness and Precision of the on-the-fly MHP analysis. The correctness of the proposed on-the-fly MHP analysis is derived from the correctness of each of the individual add/remove routines. We skip the detailed correctness proof for lack of space. The precision of the proposed analysis is same as that of Sankar et al. [2016]. Both of which may lead to false-positives if the conservative assumption (used to keep MHP analysis tractable [Taylor 1983]) that all control paths are executable is violated.

Invoking the on-the-fly MHP analysis. After each addition/deletion of a statement to/from the X10 program, we compare the new AST to the old AST and update the PST incrementally. Based on the node being added or removed, the corresponding function is invoked to update the MHP information on the fly.

MHP analysis for the whole program. The analyses proposed by Sankar et al. [2016] and our extension both are intra-procedural in nature. These can be extended to handle multiple-procedures. For programs without recursion the extension is straight-forward: we can use a program structure graph (PSG [Nandivada et al. 2013]) to represent the program and invoke the routines discussed in the previous sections. For programs with recursion, we can construct an SCC call-graph and compute MHP analysis by conservatively handling the SCC nodes representing recursive calls. Precisely computing inter-procedural MHP analysis, especially for X10 type of OO language with dynamic dispatch, is left as a future work.

On the fly MHP analysis in an IDE We have implemented both our proposed on the fly algorithms and the iMHP algorithm [Sankar et al. 2016] in the X10DT (an extension based on the Eclipse framework). Such a plugin can be used by the programmer or the IDE to identify the code that may run in parallel with a given statement, deadlock detection, identify racy programs, refactor parallel-programs, argue about program correctness, and so on.

Further, since our analysis is intra-procedural in nature, the changes in other parts of the code don't impact the code/analysis in the current window of the IDE. However, if some function is added/modified outside the IDE and loaded, we need to reanalyze that function completely (likely using a technique like iMHP [Sankar et al. 2016]) as the IDE cannot find the list of changes to be supplied to our analysis.

5 Experimental Evaluation

We have implemented the algorithms discussed in Section 3 and the iMHP algorithms of Sankar et al. [2016] as X10DT plug-ins. We then compared the running times of our on-the-fly algorithms against that of the iMHP algorithms. To perform the evaluation, in the absence of large X10 benchmarks, similar to the approach used by Sankar et al. [2016], we used two types of benchmarks: (i) synthetic benchmarks, (ii) existing benchmark kernels. We discuss both the evaluations separately.

5.1 Evaluation Using Synthetic Benchmarks

We modified the PST generator tool as mentioned by Sankar et al. [2016] to generate a wide-variety of X10 programs. The PST generator takes the number of nodes, the percentage of different parallel constructs such as `async`, `finish` and `atomic` as inputs and synthesizes a PST satisfying the input constraints. We fix the percentage of `async`, `atomic` and `finish` node as 5%, 2%, and 5%, respectively – these percentages indicate a more practical mix [Sankar et al. 2016]. We present the comparison for varying number of PST nodes (100 to 1000, in steps of 100). For each input, we generate a sequence of 100 random updates; this sequence of updates consists of adding and removing of PST nodes corresponding to `finish`, `async`, `atomic`, `loop`, `if-then`, `if-then-else` and simple sequential statements (such as assignment statement). We ran both the algorithms on this sequence of updates for five times and recorded the average running time. We also re-enforced the correctness of our proposed algorithms by comparing our generated MHP information with that of iMHP; we found that the MHP information matched.

Fig. 21 shows the improvement of running time of our analysis over the `iMHP` algorithm [Sankar et al. 2016] – ranges

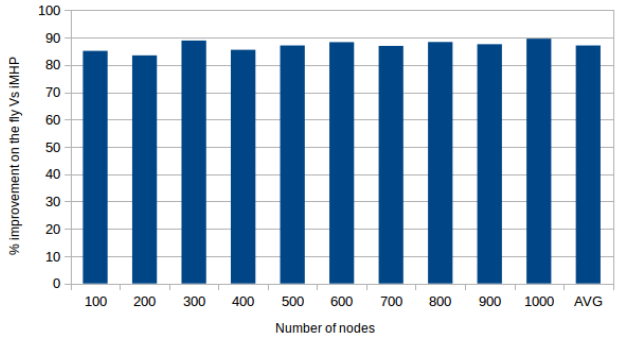


Figure 21. Execution time gains due to the on-the-fly algorithms on PSTs with varying number of nodes

#PST nodes	Total updates	iMHP analysis		on-the-fly analysis	
		naacs	exec-time	naacs	exec-time
100	10	64	261	13	38.8
200	13	109	583	12	96.4
300	24	108	699.2	2	77.6
400	12	270	3320	24	481.6
500	12	273	7949.4	26	1027.2
600	16	502	16956.2	33	1982.2
700	11	414	13519	3	1767
800	13	502	19618.8	59	2284
900	11	498	64362.4	14	8014
1000	10	531	57988.8	62	6050.8

Figure 22. iMHP analysis Vs. on-the-fly analysis, for the synthetic benchmarks. Metrics: number of `addAsync` calls and execution time. Abbreviation used: naacs = number of `addAsync` calls; exec-time = execution time in ms.

from 83% to 90%, which is significantly high; for reference, the actual execution time numbers are shown in Figure 22. These gains are realized from the improvement (up to $O(N)$) in the computational complexity of our proposed algorithm over the iMHP algorithms of Sankar et al. [2016]. Importantly, our execution times are small enough to make our algorithms a practical option in an IDE type of setting.

As discussed in Sections 3.1.7 and 3.2.7, the proposed routines `addFinish`, `remAsync`, and `remLoop` call the routine `addAsync` many times (up to A times, where A is the total number of `async` nodes inside the corresponding `finish`, `async`, and `loop` node, respectively). And this may lead to a worst-case complexity, which is comparable to that of the iMHP routines of Sankar et al. [2016]. To understand this issue further, we calculated the total number of times the `addAsync` procedure is invoked during the `iMHP` based and on-the-fly based analysis. Fig. 22 shows a comparison between these two. We can see that in practice, our on-the-fly analysis takes less time than `iMHP`, as $A \ll C$, where C is the total number of concurrency related nodes in the program.

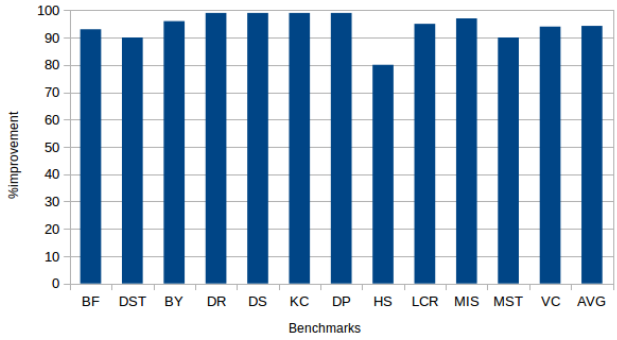


Figure 23. Execution time gains due to the proposed algorithms on the IMSuite Kernels.

5.2 Evaluation on Real Benchmarks

We also evaluated the proposed on the fly MHP algorithms and `iMHP` on the IMSuite kernels [Gupta and Nandivada 2015] that implement twelve classical concurrent algorithms: BF (perform breadth first search and compute the distance of every node from the root), DST (perform breadth first search and compute the BFS tree), BY (realize byzantine consensus), DR (create routing table), DS (compute dominating set), MIS (compute maximal independent set), KC (create k-committee), DP (perform leader election for general network), HS (perform leader election for bidirectional ring network), LCR (perform leader election for unidirectional ring network), MST (create spanning tree), and VC (perform vertex coloring). These kernel are of varying sizes: 380-991 lines of code. In each kernel, we removed 20% of the PST nodes and used them as a sequence of PST updates to be added one by one. Fig. 23 shows the improvement of running time of our analysis over the `iMHP` algorithm: ranges from 80% to 99%, which is quite high; for reference, the actual execution time numbers are shown in Figure 24. Again, our execution times are small enough to make our algorithms a practical option in an IDE type of setting.

Similar to Fig. 22, we also compared (in Figure 24) the number of times `addAsync` is called during the updates done in the context of IMSuite kernels. Similar to our experience with the synthetic inputs, we find that $A \ll C$, where A is the number of `asyncs` under the nodes being added and C is the total number of concurrency related nodes in the program, and hence in practice, on-the-fly analysis takes much less time than `iMHP`

6 Conclusion

In this paper, we show the importance of on-the-fly MHP analysis for task-parallel languages such as X10. We have implemented our proposed on-the-fly MHP analysis of X10 programs as a part of X10DT as a plug-in. We have presented a novel approach to update MHP information incrementally

Bench marks	Total updates	iMHP analysis		on-the-fly analysis	
		naacs	exec-time	naacs	exec-time
BF	6	18	39.6	3	2.4
DST	8	8	37.2	0	3.4
BY	7	19	142	1	5
DR	37	6	463.6	0	3.6
DS	54	6	671.2	1	2.2
KC	80	140	987	10	5.25
DP	88	1	1003	0	1
HS	5	26	130	7	26
LCR	8	8	135	0	6
MIS	15	30	147	1	4
MST	31	167	241	14	22
VC	38	19	258	2	15

Figure 24. iMHP analysis Vs. on-the-fly analysis, for the IMSuite kernels. Metrics: number of addAsync calls and execution time. Abbreviation used: naacs = number of addAsync calls; exec-time = execution time in ms.

when a statement is added/removed to/from the X10 program. We have shown that the cost of our proposed algorithm is not more than the cost of the iMHP algorithm proposed by Sankar et al. [2016]. We have implemented our proposed on-the-fly MHP analysis as part of X10DT (as a plug-in). We demonstrate the performance of our proposed algorithms over a large set of synthetic benchmarks (along with a series of auto-generated updates) and IMSuite benchmark kernels. The results show that our proposed algorithms lead to massive improvements over the iMHP algorithms of Sankar et al. [2016] and are of practical utility in an IDE type of settings.

References

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

S. Alstrup, M. Thorup, T. Gørtz, I. L. Rauhe, and U. Zwick. 2014. Union-find with constant time deletions. *ACM Transactions on Algorithms (TALG)* 11, 1 (2014), 6.

R. Barik. 2005. Efficient Computation of May-Happen-in-Parallel Information for Concurrent Java Programs. In *Proceedings of LCPC*. 152–169.

B L Chamberlain, D Callahan, and H P Zima. 2007. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.* 21, 3 (2007), 291–312.

C. Chen, W. Huo, and X. Feng. 2012. Making it practical and effective: fast and precise may-happen-in-parallel analysis. In *Proceedings of PACT*. 469–470.

T. H. Cormen, C. E. Leiserson, R. Rivest, and C. Stein. 2001. Introduction to Algorithms, the Massachusets Institute of Technology. *Cambridge, MA, USA,* (2001).

E Duesterwald and M.L. Soffa. 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.

Eclipse 2017. Eclipse IDE. http://www.eclipse.org/. (2017).

A. E. Flores-Montoya, E. Albert, and S. Genaim. 2013. May-happen-in-parallel based deadlock analysis for concurrent objects. In *Formal Techniques for Distributed Systems*. Springer, 273–288.

Y. Guo, R. Barik, R. Raman, and V. Sarkar. 2009. Work-first and help-first scheduling policies for async-finish task parallelism. In *Proceedings of IPDPS*. IEEE Computer Society, 1–12.

S. Gupta and V. K. Nandivada. 2015. IMSuite: A benchmark suite for simulating distributed algorithms. *JPDC* 75 (2015), 1–19.

Habanero. 2009. Habanero Java. http://habanero.rice.edu/hj. (Dec 2009).

IntelliJ 2017. IntelliJ IDEA. https://www.jetbrains.com/idea/. (2017).

H. Kaplan, N. Shafrir, and R. E. Tarjan. 2002. Union-find with deletions. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 19–28.

J. K. Lee and J. Palsberg. 2010. Featherweight X10: a core calculus for async-finish parallelism. In *ACM Sigplan Notices*, Vol. 45. ACM, 25–36.

J. K. Lee, J. Palsberg, R. Majumdar, and H. Hong. 2012. Efficient may happen in parallel analysis for async-finish parallelism. In *International Static Analysis Symposium*. Springer, 5–23.

Charles E. Leiserson. 2009. The Cilk++ Concurrency Platform. In *DAC*. ACM, 522–527.

L. Lin and C Verbrugge. 2004. A Practical MHP Information Analysis for Concurrent Java Programs. In *Proceedings of LCPC*. 194–208.

S. P. Masticola and B. G. Ryder. 1991. A model of Ada programs for static deadlock detection in polynomial times. In *ACM SIGPLAN Notices*, Vol. 26. ACM, 97–107.

S. P. Masticola and B. G. Ryder. 1993. Non-concurrency analysis. In *Proceedings PPoPP*. ACM, New York, NY, USA, 129–138.

V. K. Nandivada, J. Shirako, J. Zhao, and V. Sarkar. 2013. A transformation framework for optimizing task-parallel programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 35, 1 (2013), 3.

G. Naumovich and G. S. Avrunin. 1998. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *Proceedings of FSE*. 24–34.

G. Naumovich, G. S. Avrunin, and L. A. Clarke. 1999a. An efficient algorithm for computing MHP information for concurrent Java programs. In *Proceedings of ESEC/FSE*. 338–354.

G. Naumovich, G. S. Avrunin, and L. A. Clarke. 1999b. Data flow analysis for checking properties of concurrent Java programs. In *Proceedings of the 21st international conference on Software engineering*. ACM, 399–410.

A. Sankar, S. Chakraborty, and V. K. Nandivada. 2016. Improved MHP Analysis. In *Proceedings of the 25th International Conference on Compiler Construction*. ACM, 207–217.

V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. 2015. X10 language specification. (2015). http://x10.sourceforge.net/documentation/languagespec/x10-254.pdf

R. N. Taylor. 1983. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica* 19, 1 (1983), 57–84.

V. Vojdani and V. Vene. 2009. Goblint: Path-sensitive data race analysis. In *Annales Univ. Sci. Budapest., Sect. Comp*, Vol. 30. 141–155.

X10DT 2017. An Eclipse based IDE for X10. http://x10-lang.org/documentation/x10dt-installation.html. (2017).

S. Zhan and J. Huang. 2016. ECHO: Instantaneous in Situ Race Detection in the IDE. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 775–786. https://doi.org/10.1145/2950290.2950332

A | Artifact Setup and Evaluation

A.1 Abstract

Abstact: This appendix describes how to replicate the experiments to evaluate the performance of the proposed on-the-fly MHP analysis as compared to the prior work (iMHP analysis). It details the software requirement, installation instructions, steps to configure and run the experiment, and the expected results.

A.2 Description

A.2.1 Overview

- Google drive link: https://drive.google.com/drive/folders/1Y1b21-oWSKiLKajrgSwgKnLyv_K7rE5A?usp=sharing A detailed version of this artifact with detailed screenshots can be found in a README file there.
- Required software: Java, Eclipse, X10DT.
- Data set: IMSuite Benchmarks (available in the drive)
- Output: Analysis times (along with the % improvement) of all Kernels using (a) on-the-fly MHP analysis, (b) iMHP, over IMSuite kernels and synthetically generated Program Structure Tree (PST)s. In addition, we also output the number of times the addAsync routines are called,
- Experiment workflow: (i) Setup Java and Eclipse, (ii) add the developed plugin and see a test program in action, (iii) Analysis using the IMSuite kernels, (iv) Analysis using the synthetically generated PSTs.
- Experiment customization: Size of the synthetically generated PSTs and the different types of nodes etc.
- Publicly available?: Yes
- Note: The evaluation involves randomly adding and removing nodes in both synthetic and publicly available benchmark kernels. As a result, while the overall % improvement should still remain in the same ballpark figure, the other statistics (the number of calls to asyncAdd) may not exactly match.

A.2.2 Installation

1. Install Java 1.8
2. Install Eclipse on your system (https://www.eclipse.org/downloads/packages/installer).
3. Install eclipse PDE.
   a. In the menu click on Help –> Install New Software.
   b. In the Available Software dialog select the 2019-09 site (http://download.eclipse.org/releases/2019-09) from the "Work with" drop down.
   c. In the search box enter the phrase "Plug-in" this should filter the list so you can see the Eclipse Plug-in Development Environment.
   d. Click the checkbox and next till finish to install.
4. Install X10DT via Eclipse Update Manager. Refer this link: http://x10-lang.org/documentation/x10dt-installation.html
5. Copy `iMHP_1.0.0.201912010346` from artifact folder to '`${ECLIPSE_INSTALLED_FOLDER\}/plugins`' and to '`${ECLIPSE_INSTALLED_FOLDER\}/dropins/plugins`'.
6. Run eclipse from terminal with command 'eclipse.exe -clean' on windows and './eclipse -clean' on linux.

A.2.3 Testing the setup (for both on-the-fly MHP and iMHP) using a toy program

1. Create an X10 project. In the menu click on `File->New ->Project->X10->X10 Project (Java back-end)` named 'test'.
2. Copy 'test.x10' program from 'artifact' folder to 'test/src'. Keep this file open in eclipse.
3. Please keep test.x10 program open in eclipse.
   a. In the menu click on 'MHP Analyses' plugin. In Drop down you will see four options 'Evaluation on Real Benchmarks', 'Evaluation using Synthetic Benchmarks', 'iMHP' and 'on-the-fly'. Please click on 'iMHP' to run iMHP algorithm on this program.
   b. Please enter statement number for which you want to know MHP information.
4. After running the test, the result pop-up will appear showing what are the statements may run in parallel with the given statement.
5. Please add a statement in the program (One update at a time) and wait for few second to load the program in eclipse. Please save the program.
6. In the menu click on 'MHP Analyses' plugin. Please click on 'on-the-fly'.
7. Please enter statement number for which you want to know MHP information.
8. After running the test, the result pop-up will appear showing what are the statements may run in parallel with given statement.
9. Note: The overall execution time you may observer will include 'on-the-fly execution time' + 'delay in running the test automatically in Eclipse'. Hence, for measuring/comparing execution time, only consider the execution time of on-the-fly algorithm and not the delay with it. Here delay refers to loading the x10 program in eclipse after each update. Delay time varies based on the number of statements in an x10 program.

A.2.4 Evaluation using the Synthetic kernels (For Figure 21 and 22)

To perform this evaluation, we use a PST generator (internally) to generate PSTs as per the given spec. We present the comparison for varying number of PST nodes (100 to 1000, in steps of 100).

1. In the 'MHP Analysis' menu click on 'Evaluation using Synthetic Benchmarks'.

2. Please enter number of PST nodes (in the paper, we use 100, 200, . . . up to 1000 nodes).
3. Please enter number of update sequence (use the numbers from Figure 22).
4. Please enter number of times you want to run the above configuration.
5. Please enter delay. (If number of PST nodes is 100, delay of 5000ms is recommended. More the number of nodes more delay should be given.)
6. After running the test, the result will pop-up.

A.3 Evaluation using the IMSuite kernels (For Figures 23 and 24)

To perform this evaluation, we remove 20% of nodes and used them as a sequence of PST updates to be added one by one. To simulate the addition of nodes, we have to add a delay between each update.

1. Copy 'X10Project' from 'artifact' folder to your system. The '*.x10' files in 'X10Project/src/' folder are input to 'Evaluation on Real Benchmarks'.
2. Keep test.x10 program open in eclipse.
3. Click on 'Evaluation on Real Benchmarks' and enter the file location.
4. Enter the delay. (For 'bfsBellmanFord.x10' program delay of 5000ms is recommended.)
5. After running the test, the result will pop-up.