A New Method of MHP Analysis for Languages with Dynamic Barriers

Conference Paper · May 2012

DOI: 10.1109/IPDPSW.2012.70

3 authors:

Saurabh Joshi
Indian Institute of Technology Hyderabad
27 PUBLICATIONS 185 CITATIONS
SEE PROFILE

R. K. Shyamasundar
Indian Institute of Technology Bombay
291 PUBLICATIONS 1,587 CITATIONS
SEE PROFILE

Sanjeev Aggarwal
Dow Chemical Company
198 PUBLICATIONS 1,358 CITATIONS
SEE PROFILE

A New Method of MHP Analysis for Languages with Dynamic Barriers

Saurabh Joshi
Department of CSE
IIT Kanpur
Kanpur- 208016, India
sbjoshi@cse.iitk.ac.in

R K Shyamasundar
School of Technology and Computer Science
TIFR
Mumbai - 400005, India
shyam@tcs.tifr.res.in

Sanjeev K Aggarwal
Department of CSE
IIT Kanpur
Kanpur- 208016, India
ska@cse.iitk.ac.in

***Abstract*—May-happen-in-parallel analysis is a very important analysis which enables several optimizations in parallel programs. Most of the work on MHP analysis has used forward flow analysis to compute "parallel(n)" — set of nodes which may execute in parallel to a given node "n" — including those approaches that addressed the issue for dynamic barrier languages. We propose a new approach to MHP analysis called *Phase Interval Analysis* (PIA) which computes phase intervals, corresponding to dynamic barriers, in which a statement may execute. PIA enables us to infer an order between two statements whenever it can establish that they can not execute in parallel. Because the ordering relation is transitive, we may also be able to infer indirect synchronization happening between two statements, even when they do not share a barrier. To the best of our knowledge, the issue of indirect synchronization has not been addressed prior to this work. We also compute condition functions under which different instances of the same statement may not execute in parallel, when the statements are nested within loops.**

***Keywords*-Parallel programming, Data flow computing, Reasoning about programs**

I. Introduction

With the emergence of parallel programming languages like X10 [7] and Habanero-Java [5,6], may-happen-in-parallel(MHP) analysis becomes very important, as precise MHP analysis opens up doors for a plethora of optimization opportunities and leads to more effective debugging techniques for parallel programs.

The goal of MHP analysis is to compute a set M of pairs of program statements, that may run in parallel during some execution of the program. As MHP analysis is undecidable in general [18], we would like to calculate the smallest set M' such that $M \subseteq M'$. Calculating the largest set Q, of pairs of program statements which can not happen in parallel achieves the same because $M \subseteq \bar{Q}$, where $\bar{Q}$ denotes the complement set of Q. The more we infer that some pair of nodes may not execute in parallel, the more optimization opportunity it provides.

Earlier work [20] has focused on computing a set $parallel(n)$ for a node (or statement) n containing all the nodes m which may execute in parallel with n. From now on "$m \parallel n$" will denote the may-happen-in-parallel relationship between m and n, whereas "$m \nparallel n$" will denote the can-not-execute-in-parallel relationship between m and n. Problem with establishing $m \nparallel n$ using techniques which relies on computing a set $parallel(n)$ has a drawback that even when $m \nparallel n$ is established, we can not establish $m < n$ or $n < m$, denoting that m will always precede or succeed n respectively. In addition to having information that $m \nparallel n$, if we can infer the order between two nodes m and n it would open up opportunities for a plethora of optimizations.

Let us take a motivating example given in Fig. 1. Here, `async(c){S}` denotes that a new activity (or thread) is spawned asynchronously, which will execute the statement `S`. The clock `c` indicates the barrier(s) the new activity would inherit from its parent activity. A `next` statement indicates a barrier synchronization point. An activity registered with a set of barriers, will not progress beyond `next` until all the activities registered with any of these barriers has also arrived at their corresponding `next`. These constructs will be explained in detail later on.

In the given code fragment, activity P spawns three new activities named A, B and C which will execute the code segments given in their corresponding braces. In addition, A is registered with barriers `c1` and `c2`, B is registered with `c1` and C is registered with `c2`. In Fig. 1, it is easy to see that the statement labelled `p1` will always precede the statement labelled `s3`. Even though activities B and C are registered with different sets of barriers, the synchronization happens because they both share a barrier with activity A. Because A and B share the barrier `c1`, `p1` will always precede `t2`. Similarly, `t2` will precede `s3` due to a common barrier `c2` between activities A and C. None of the earlier work [1,20] detects such an indirect synchronization, because the set $parallel(n)$ only consists of those nodes which share at least one barrier with n. Hence, even when one establishes `t2` $\nparallel$ `p1` and `t2` $\nparallel$ `s3`, it fails to establish that `p1` $\nparallel$ `s3` in the absence of an inferred order. If we could infer an order between statements which can not happen in parallel, it would lead to additional optimizations in programs. For example, in Fig. 1, if we can infer `t1` $<$ `p2` and `p2` $<$ `t3`, we can use copy propagation to establish `z` $=$ `u` at `t3`.

We propose a flow analysis called *Phase Interval Analysis* (PIA) which will always be able to infer the order between two statements m and n, whenever it can establish $m \nparallel n$.

```
//activity P
async(c1,c2) { //activity A
     t1: x=y;
     next;
     t2: ...
     next;
     t3: z=x;
}
async(c1) { //activity B
     p1: ...
     next;
     p2:x=u;
     next;
     p3: ...
}
async(c2) { //activity C
     s1: ...
     next;
     s2: ...
     next;
     s3: ...
}
```

Figure 1. A motivating example

We do this by labelling each node with a *phase interval*. This interval denotes the minimum phase and the maximum phase of the barrier during which a node can execute. Hence, a phase interval of $\langle a, b\rangle_c$ denotes that a statement will execute anywhere between the phase a and b of the barrier c. Each time a barrier synchronization point `next` is encountered, the barrier will have said to move to the *next* phase.

A statement in the program has multiple instances if it is nested within loops. If m and n are nested within a common set of loops $L_1, \ldots, L_l$ then we need to infer whether $m[i_1, \ldots, i_l] < n[i'_1, \ldots, i'_l]$. To deal with such instances, we compute a condition function $\phi_{m<n}$ in terms of instance variables $i_1, \ldots, i_l, i'_1, \ldots, i'_l$, such that whenever $\phi_{m<n}$ evaluates to $true$, m will precede n for the corresponding instance. This gives us additional opportunities when $m < n$ does not hold for all instances but only for some of the instances.

A. Our Contributions

Having looked at the motivating example, we enlist the contributions of this paper as follows :

- We propose a data flow analysis called *Phase Interval Analysis*(PIA), which computes the phase interval of a barrier within which a statement can execute. This analysis is especially useful for languages with dynamic barriers where activities can **join or leave the barrier any time**.
- In addition to inferring $m \nparallel n$, we also infer an order $m < n$ or $n < m$ indicating which node would execute first, giving an opportunity for more optimization.
- We compute condition functions $\phi_{m<n}$ under which two nodes m and n or instances thereof, will precede one another.
- The most important contribution is that PIA enables us to infer indirect synchronization between two nodes even when they do not share a barrier.

The rest of the paper is organized as follows. Section II mentions the related work. Section III-A describes a language having dynamic barriers as a synchronization construct. Section III-B introduces program structure tree, statement instances within nested loops and condition functions. Section III-C describes the clocked control flow graph that is used by phase interval analysis described in section IV. Finally, section V summarizes the paper.

II. Related Work

May happen in parallel analysis has been of interest since the advent of parallel programming. MHP is undecidable in general [18]. Under certain restrictions, MHP analysis for the async-finish parallelism without dynamic barriers has been shown to be in cubic time [13]. MHP analysis has been studied in [9,15,16] for Ada, in [4,14,17] for Java and explored in [1,12,20] for X10. Agarwal et al. used Program Structure Tree (PST) for X10 programs [1] and then their algorithm tries to compute *condition vectors*— which gives equality constraints over instance variables — for which given pair of statement instances[1] can not run in parallel. This paper enriches the notion of condition vectors with condition functions with more expressive power. Context sensitive MHP information is computed using type inference in [12]. However, both [1] and [12] compute MHP in the absence of clocks (dynamic barriers) for the async-finish parallelism model. Computing MHP information for languages with dynamic barriers, where activities can join and leave the barrier any time, becomes a bit more difficult than computing the same for static barriers. To the best of our knowledge, MHP computation for dynamic barrier languages has been addressed only in [20], which uses data flow analysis on Clocked Control Flow Graphs (CCFG). MHP information computed in their work focuses only on statement pairs, therefore it misses on opportunities when two statement instances can not run in parallel. As they do not handle statement instances, they can infer $m \nparallel n$ only if for all possible statement instances of m and n, they do not execute in parallel. Therefore, their approach, though being conservative, misses out on many optimization opportunities. Phase Interval Analysis (PIA) described here computes condition under which two statement instances will not run in parallel. Concept of using intervals to conservatively capture information flow in the program is not new and has been studied in the setting of program verification in [8]. PIA is an adaptation of abstract interpretation described in [8] in the setting of MHP analysis for languages with dynamic barriers. It is worth noting that the work presented here is an attempt to capture some of the **missed opportunities** in [1] and [20].

[1]A statement inside a loop can have multiple instances with different MHP characteristics at different instances.

III. Notations and Background

This section lays the technical background and introduces notations that will be used to describe the phase interval analysis.

A. Abstract Dynamic Barrier Language

This section describes ADBL (Abstract Dynamic Barrier Language), which is powerful enough to capture programming constructs of languages with dynamic barriers such as clocks in X10 [7] and phasers in Habanero-Java [5,6].

SyncStmt	::	Finish | AsyncList | ClockStmt | ClockedAsync
ClockStmt	::	ClockVar := `new clock()` | ClockVar.`drop()` | `next`
AsyncList	::	Async AsyncList | ϵ
Async	::	async{Stmt}
ClockedAsync	::	`async`(ClockVarList) { Stmt }
Finish	::	`finish` { AsyncList }
Stmt	::	`if`(BExpr) Stmt `else` Stmt | `do` $Const$ `times with` Var Stmt `od` | `while` (BExpr) Stmt | SyncStmt
Atomic	::	`atomic` {SeqStmt}

ADBL has all the constructs of a sequential languages denoted by *SeqStmt* like loops, sequence and branch. We introduce two kinds of loop in the language. `while` introduces a general loop, where as a `do` loop represents all those loop for which it is possible to statically figure out the number of times the loop will iterate before exiting. In another words, the loop count is a static constant or a symbolic constant. Var in the `do` loop indicates the loop induction variable which increases by 1 after each iteration and ranges from 0 to $Const - 1$. We will see later that such loops provides an additional opportunity in asserting that two statements can not execute in parallel.

- *Clocks* : Clocks act as dynamic barriers in ADBL. Whenever an activity (thread) declares a new clock using `new clock()`, that activity is said to have created and registered with the clock (barrier). An activity can deregister itself from the clock using a `drop()` on that clock. An activity also implicitly drops all the clocks that it is registered with, when it finishes. Thus, number of activities participating in a clock (dynamic barrier) may keep changing. Note that **an activity can not re-register to a clock which has been dropped by it**. Just like in earlier works, we assume that the set of clocks in the program is given by $\{c_1, \ldots, c_r\}$.
- `async` : An `async` statement spawns an activity which executes the given statement *Stmt*. Once this new activity is spawned, it is completely independent from its parent activity and can outlive it. A *ClockedAync* inherits a set of clocks from its parent activity. This increases the number of activities participating on an existing clock (dynamic barrier). Note that apart from creating a new clock this is the only other way to register with a clock. At this point of time, the parent activity has to be registered with the clock which it passes on to the child activity.
- `finish` : A `finish` statement is said to terminate only when all the activities transitively spawned from its scope terminates. In some sense this construct acts in a similar fashion to *join()* in Java. A `finish` can not spawn a *ClockedAsync* within its immediate scope. This prevents deadlock from happening due to cyclic dependence between a child waiting for a clocked parent, and parent waiting for the child to terminate so that its `finish` can terminate.
- `next` : A `next` indicates a barrier synchronization point. An activity registered on a set of clocks C, will wait for all other activities registered on any subset of C, to reach the corresponding synchronization point. After this synchronization, a clock is said to have **advanced by one phase**. Since, `next` advances all the clocks an activity is registered with, it prevents a deadlock of the following kind. Instead, if we allow `ClockVar.next`, two activities registered on clocks `c1,c2` may execute `c1.next; c2.next` in different order and run into a deadlock. An important point to note that if an activity is not registered on any clock while it encounters `next` **then the program raises an error**.
- `atomic` : This construct allows the statements in its body to be run in complete isolation and atomically with respect to the rest of the program. This is the reason why `atomic` does not allow any *SyncStmt* inside its body. Only the sequential subset of ADBL (similar to any sequential programming language) is allowed inside an `atomic`. Note that $SeqStmt$ inside an `atomic` can not happen in parallel with any other statement or node making it trivial to compute MHP information for these blocks. Hence, for the rest of the paper, we will only focus on those parts of the program which are outside an `atomic`. We include `atomic` in ADBL for the sake of completeness as this is the only construct providing isolation and mutual exclusion in an ADBL program.

B. Program Structure Tree and Condition Functions

We have described ADBL in the previous section. This section describes *Program Structure Tree* (PST) , used in one of the earlier approaches of MHP analysis in the presence of `async` and `finish` constructs but without clocks. We also introduce the notion of condition functions here. From now on, we will use the terms *statement* and *node* (a basic block containing the statement) interchangeably.

Definition 1: A node n_1 is said to be a PST child of another node n_2 if and only if the statements corresponding to node n_1 are in the immediate lexical scope of n_2.

For the code fragment shown in Fig. 2, corresponding PST is given in Fig. 3.

Definition 2: For two statement instances $S_1\left[i_1,\ldots,i_l,j_1,\ldots,j_m\right]$ and $S_2\left[i'_1,\ldots,i'_l,k_1,\ldots,k_n\right]$, **condition function** $\phi_{n_1<n_2}(i_1,\ldots,i_l,i'_1,\ldots,i'_l,j_1,\ldots,j_m,k_1,\ldots,k_n)$ provides the condition in terms of $i_1,\ldots,i_l,i'_1,\ldots,i'_l,j_1,\ldots,j_m,k_1,\ldots,k_n$ under which node n_1 corresponding to statement S_1 will precede n_2 corresponding to statement S_2 in the execution. Here, S_1 and S_2 have common loops $L_1,\ldots,L_l$. In addition, S_1 is enclosed in loops $LJ_1,\ldots,LJ_m$ and S_2 is enclosed in loops $LK_1,\ldots,LK_n$. Here, $i_1,\ldots,i_l,i'_1,\ldots,i_l,j_1,\ldots,j_m,k_1,\ldots,k_m$ are instance variables. Similarly $\phi_{n_1>n_2}$ and $\phi_{n_1\nless n_2}$ can be defined.
The following code fragment gives an example of S_1 and S_2 inside loops as shown in definition 2.

```
L1:
  ....
    Ll:
    async{
       ....
      LJ1:
       ....
         LJm:
           ...
           S1;
           ...
         }
      async{
       .....
         LK1:
           ...
             LKn:
               ...
              S2
         }
```

There has been two different approach for MHP analysis in async-finish parallelism which does not take into account dynamic barriers (clocks). PST based approach given in [1] uses *condition vectors* to specify conditions under which two statement instances are guaranteed not to execute in parallel. On the other hand, the approach given in [12] uses type inference to compute context sensitive MHP information. Approaches dealing only with the async-finish parallelism provides a conservative solution even in presence of dynamic barriers. For that reason, either of the two approaches given in [1] or [12] can be used for preprocessing. Since the later is costlier in terms of time complexity, we recommend using the approach given in [1]. Similar approach is used in [20] for the preprocessing. MHP information computed by the approach given in [1] is used as starting point to compute additional information in presence of dynamic barriers in [20] as well as in PIA which shall be described later.

C. Clocked Control Flow Graph

In this section we describe Clocked Control Flow Graph (CCFG) using which we will perform Phase Interval Analysis (PIA) using forward data flow analysis technique.

Fig. 4 shows a code fragment and Fig. 5 shows the corresponding CCFG. CCFG is similar to any control flow graph with the following additions.

- In addition to every procedure of the program, every `async` and `finish` has an *entry* and an *exit* node.
- `async` nodes branch out a new control flow as shown in Fig. 5.
- All clock related operations `next`, `c=new clock()`, `c.drop` are in a separate block containing this single statement.
- If a node defines a new clock using `c=new clock()` then it is labelled with the clock c in addition to other clocks it might receive from its predecessors.
- A node is labelled with the intersection of the sets of clock that it receives from its predecessors.
- If a node drops a clock using *c.drop* then c is removed from its label.
- An *entry* node of a clocked `async` is labelled with the set of clocks passed to it. Observe in Fig. 5, how two different `async` receives $c1$ and $c2$ respectively. Remember that language semantics does not allow an async to be registered with clock c if its parent activity is not registered with c at that moment. Doing so would result in an invalid program.

IV. Phase Interval Analysis

In this section we describe a data flow analysis which we call *Phase Interval Analysis* (PIA).

Definition 3: A phase interval $\langle a,b\rangle_c$ associated with a node n with respect to a reference node R denotes that n will only execute in relative clock phases $\{a,a+1,\ldots,b\}$ of a clock c. By relative clock phase we mean that, **if** the reference node R executes in phase 0 of clock c (interval $\langle 0,0\rangle_c$)) **then** n can only execute in one of the phase from a to b. In that sense, the phase interval computed is relative to the reference point.

Once we have obtained CCFG and PST as described in earlier sections, we can compute phase intervals for nodes of interest. Given two nodes n_1 and n_2, we find out their least common ancestor A_p in PST. Focus only on the subgraph G_p of CCFG induced by descendants of A_p. In another words, take into consideration only the nodes in the lexical scope of A_p. Next, we take the least common ancestor A_q of n_1 and n_2 in this subgraph G_p of CCFG. Take A_q as a reference node and give it a phase interval of $\langle 0,0\rangle$ for the

```
if(b)
{
  while(b1)
  {
    s1;

  }
}
else
{
  s2;
}
```

Figure 2. Code fragment for PST example

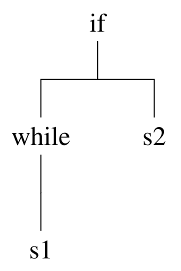

Figure 3. PST example

```
main()
{
   s1;
   finish {
     async {
     c1=new clock();
     c2=new clock();
     async(c1)
     { t1; next; t2; }
     async(c2) {
      p1;
      if(b)
      { next; }
      else
      { c2.drop; }
      p2;
      }
     s2; next; s3;
     }
   }

}
```

Figure 4. Code fragment for CCFG example

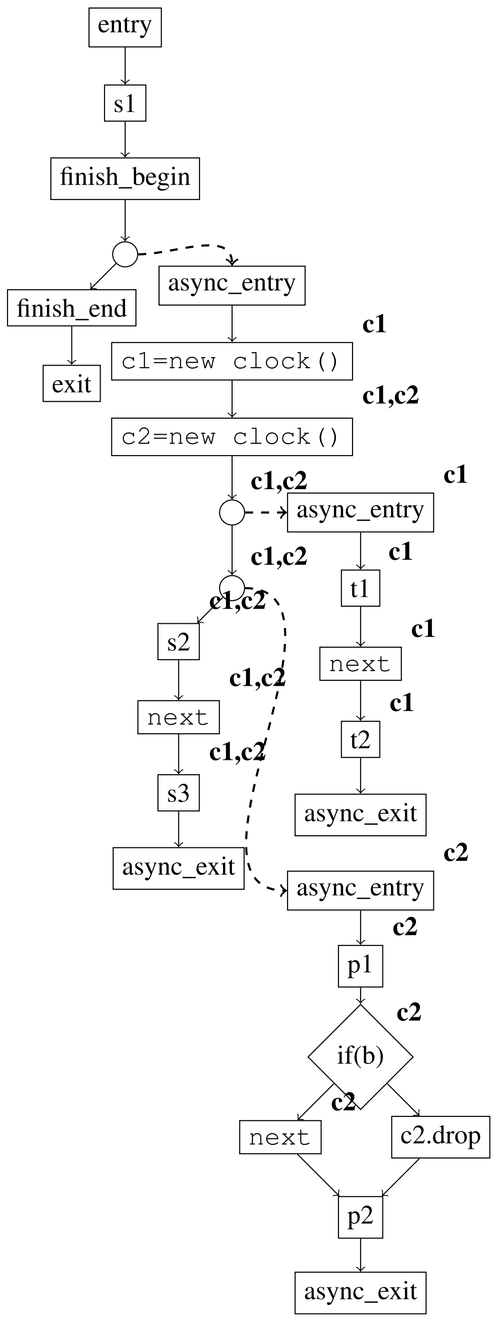

Figure 5. CCFG example

clocks shared by n_1 and n_2. Now, the data flow proceeds by computing the minimum and the maximum number of barrier synchronization `next` encountered along all paths to a node from the reference point.

Algorithm 1 computes PIA for nodes of interest n_1 and n_2 with respect to their least common ancestor A_q. The analysis holds only if A_q is registered on some set of common clocks. If not, n_1 and n_2 does not share a clock. For such a pair of nodes, we may be able to infer an order via indirect

Algorithm 1 PIA

```
1: input: Subgraph $G_p$ of CCFG induced by nodes inside the scope of $A_p$ ( least common ancestor of $n_1$ and $n_2$ in PST )
2: output: Phase intervals for all nodes in $G_p$.
3: $A_q \leftarrow$ LEASTCOMMONANCESTOR($n_1, n_2, G_p$)
4: The following portion is repeated for every clock $c$ common to $n_1$ and $n_2$.
5: PHASEINTERVAL($A_q$) $\leftarrow \langle 0, 0\rangle$
6: $G_{pb} \leftarrow G_p \setminus$ all the backedges
7: for $n$ in the topological sort of $G_{pb}$ do
8:     $a \leftarrow min_{i \in parent(n, G_{pb})}$ PHASEINTERVAL(i).first
9:     $b \leftarrow max_{i \in parent(n, G_{pb})}$ PHASEINTERVAL(i).second
10:    if $n$ is a next node then
11:        PHASEINTERVAL($n$) $\leftarrow \langle a+1, b+1\rangle$
12:    else if $n$ is a drop node for clock $c$ then
13:        PHASEINTERVAL($n$) $\leftarrow \langle a, \infty\rangle$
14:    else if $n$ is a looptail node then
15:        Let $h$ be a corresponding loophead node
16:        $\langle ha.hb\rangle \leftarrow$ PHASEINTERVAL($h$)
17:        $h.oneiter \leftarrow \langle a - ha, b - hb\rangle$
18:        if $h$ is a do loop then
19:            PHASEINTERVAL($n$) $\leftarrow \langle (a - ha)h.itercount + ha, (b - hb)h.itercount + hb\rangle$
20:        else
21:            PHASEINTERVAL($n$) $\leftarrow \langle ha, \infty\rangle$
22:        end if
23:    end if
24: end for
```

synchronization which we shall discuss later. We take A_q as a reference point, and we compute phase intervals only with respect to this reference point and the clocks that are common to n_1 and n_2. We initialize A_q to $\langle 0, 0\rangle$ (line 5). We ignore all the backedges, as we shall discuss later how to handle loops by having some extra information attached to the nodes in the loop. Ignoring the backedges gives us a directed acyclic graph. The nodes are now topologically sorted and will be processed in that order(line 7). Now, for each node n, we set the lower bound to be equal to the minimum of all the incoming nodes and upper bound to be the maximum of all the incoming nodes (line 8-9). When we have a `next` node, we add 1 to this bound, as `next` is the only construct advancing a phase(line 11). We set the upper bound to ∞ when we encounter a `drop` on the clock, as after an activity drops out of a clock, it can not be determined in which phase a node of that activity will be executed (line 13).

n_1 and n_2 can be surrounded by loops which requires a special care. As a preprocessing, we augment all the loops to have a $loophead$ and $looptail$ node. Control flow graph of a normal loop looks like the one shown in Fig. 6. We transform it into the one shown in Fig. 7. Loop header can easily be identified by standard analysis techniques. We call it the $loophead$ of the loop. Then, introduce a dummy node called $looptail$ which immediately follows the loop. A dotted line (an *order* edge) shown in Fig. 7 is added from the last node of the loop, just to ensure that $looptail$ is processed, after all the nodes which are inside the loop, when we topologically sort the nodes. The last node within the loop is the one which has a backedge going to the $loophead$ and all the other nodes in the loop can reach it without going through $loophead$. Alternatively, one can add *order* edges from all the nodes belonging to the loop to the $looptail$. In addition, dashed line in Fig. 7 matches $looptail$ and $loophead$ making it easier to reach from one to the other. The moment $looptail$ is encountered, we know how many phases can pass in one iteration of the loop by calculating the difference in the intervals of $looptail$ and $loophead$(line 17). In $loophead$ we store the information $itercount$ which says how many times the loop will execute in case of `do` loops. From this, we can calculate in which phase $looptail$ will start its execution (line 19). If it is a while loop, we may not be able to compute how many iteration it will take because it is undecidable in general [19]. Therefore, we calculate a conservative bound for $looptail$ by taking the lower bound to be the lower bound of the corresponding $loophead$, taking into account the possibility of 0 iterations, and the upper bound to be ∞(line 21).

Within the scope of A_q if n_1, having phase interval of $\langle a, b\rangle$, is nested within loops $LJ_1, \ldots, LJ_m$ with phase intervals at the heads of these loops being $\langle a_1, b_1\rangle, \ldots, \langle a_m, b_m\rangle$, then the phase interval for a statement instance $n_1\,[j_1, \ldots, j_m]$ can be given by

$$\langle a_1, b_1\rangle + \sum_{i=1}^{m-1} (\langle LJ_i.oneiter\rangle j_i + \langle a_{i+1} - a_i, b_{i+1} - b_i\rangle) + \langle LJ_m.oneiter\rangle j_m + \langle a - a_m, b - b_m\rangle$$

The way it works is as follows. Calculate the interval corresponding to the number of times the outermost loop completes j_1 iterations using $\langle LJ_1.oneiter\rangle j_1$. Then move on to the head of the next loop LJ_2 by adding the interval difference between the head of LJ_2 and LJ_1 and then repeat the same process.

Here, additions and subtractions over the interval is pairwise and a scalar multiplied to the interval indicates that it is multiplied to both upper as well as lower bound.

$$\langle a_1, b_1\rangle + \langle a_2, b_2\rangle \equiv \langle a_1 + a_2, b_1 + b_2\rangle$$

$$\langle a, b\rangle C \equiv \langle a * C, b * C\rangle$$

An example is given in Fig. 8 shows how these intervals are calculated when a statement is nested within multiple loops. The second loop $l2$ has a `next` within a branch causing $\langle 0, 1\rangle$ for one iteration of that loop. The statement following the loop $l2$ gets the interval $\langle 0, N2\rangle$ as the loop can iterate exactly $N2$ times. The following `next` advances the phases by 1. An `async` following it creates a branch in the control flow which does not return, hence at the end of loop $l1$, the interval remains $\langle 1, N2{+}1\rangle$. $i1$-th instance of `async` will be generated after loop $l1$ has completed $i1$ iterations[2], hence

[2]instances are counted from 0 as mentioned in the ADBL `do` loop description

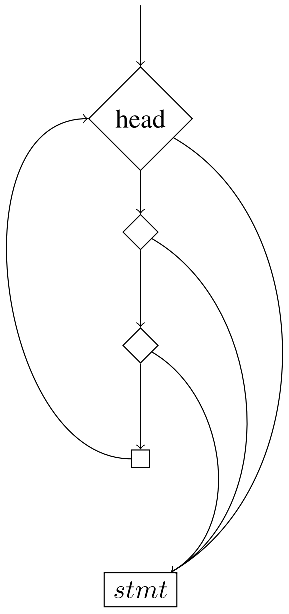

Figure 6. Flow graph of a loop

the async gets the interval $\langle 1, N2+1\rangle + \langle 1, N2+1\rangle * i1$. Similar, argument goes when $s1$ is nested inside $l3$ and in the end it gets $\langle 1, N2+1\rangle + \langle 1, N2+1\rangle * i1 + \langle 1,1\rangle * i3 + \langle 1,1\rangle$.

Once we have phase intervals for n_1 and n_2, we may be able to come up with an ordering as follows :

1) Let the set of common clocks between n_1 and n_2 be $c_1, \ldots, c_r$. Let phase intervals associated with n_1 and n_2 be $\langle x^{c_1}, y^{c_1}\rangle \ldots \langle x^{c_r}, y^{c_r}\rangle$ and $\langle z^{c_1}, w^{c_1}\rangle \ldots \langle z^{c_r}, w^{c_r}\rangle$ respectively.
2) We can now generate the condition for their ordering in the following manner. $\phi_{n_1<n_2} \leftarrow \bigwedge_{i=1}^{r} y^{c_i} < z^{c_i}$. Similarly, condition function $\phi_{n_1>n_2}$ can be computed. The condition function mandates that the upper bound for n_1 must be less than the lower bound for n_2 for all the clocks under consideration. If, n_1 and n_2 are not nested inside loops within the scope of A_q then all the intervals would have constant bounds in the interval making it possible to determine whether the condition function evaluates to $true$ or $false$.
3) If A_q itself is nested within loops $L_1, \ldots, L_l$ then we augment the condition function with the condition that the instance of n_1 and n_2 should come from the same iteration of these common loops $\bigwedge_{j=1}^{l} i_j = i'_j$.

An alternate way to look at the phase interval computation is using inference rules as shown in Fig. 9. Rule for SKIP captures the effect of statements which does not have any synchronization or parallel constructs like async,finish, next etc. It says that any such statement (e.g. x = y+z) can be treated as $skip$ and has no effect on the phase interval.

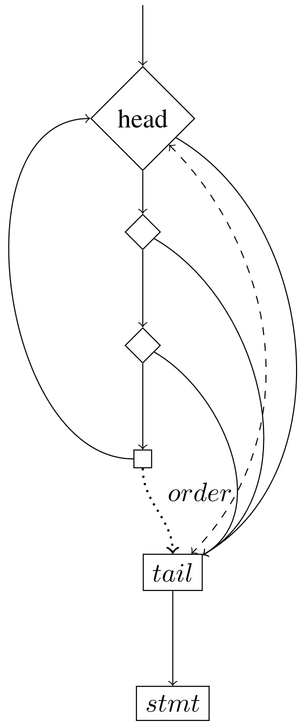

Figure 7. Modified flow graph for a loop for PIA

SEQ indicates the natural sequential composition of intervals of statements s_1 and s_2. next is the only statement which increments the phase interval to indicate the phase change. A clock can advance its phase only through next. ITE rule describe the natural way to merge two disjoint control flow paths. The computation can take any one of the two path hence it is conservative to say that when the paths merge, the minimum phase that a clock could be in is the minimum of the two. Similar argument goes for the upper bound of the interval. ASYNC poses an interesting case. The interval does not change because async starts a new computation along a separate control flow path. Any changes in the clock phase along that path has no direct effect on the phase change along the current path. FINISH also does not change the interval, because according to the language semantics given in section III-A, a finish can only have a set of non clocked async. Hence, any child activity transitively spawned within finish can not share a clock with the activity in which finish is executing. LOOP-INST dictates that if any statement s forming the body of the loop (do or while) which starts executing in interval $\langle a, b\rangle$ and finishes in interval $\langle a_1, b_1\rangle$ then the i-th instance has to start and finish with intervals described by the rule. For a general looping construct when it is not possible to statically determine the number of iteration it might take before the completion of the loop, it is natural to declare that the upper bound of the phase interval could be ∞ as noted in LOOP-GEN. LOOP-COUNT however, gives the phase interval at the end of the

```
async(c)                                              // <0,0>
{
l1 : do N1 times with i1 {                            // <0,0> , oneiter<1,N2+1>
l2:    do N2 times with i2 {                          // <0,0>, oneiter<0,1>
          if(b) { next; }
        } od                                          //enddo:l2 <0,1>
        // <0,N2>
          next;                                       // <1,N2+1>
          async(c)                                    // <1,N2+1>, finally : <1,N2+1> + i1*<1,N2+1>
          {
l3:         do N3 times with i3 {                     //<1,N2+1>, oneiter<1,1>
              next;
              s1;                                     // <2,N2+2>, finally : <1,1> + i3*<1,1> + <1,N2+1> + i1*<1,N2+1>
             }od                                      //enddo:l3 <2,N2+2>
                                                      //<1,1>*N3 + <1,N2+1>
          }
      } od                                            //enddo:l1 <1,N2+1>
                                                      // <1,N2+1>*N1
}
```

Figure 8. Phase Interval Analysis for nested loops

$$\frac{}{\langle a,b\rangle\ skip\langle a,b\rangle} \quad \text{(SKIP)}$$

$$\frac{\langle a,b\rangle s_1\langle a_1,b_1\rangle \qquad \langle a_1,b_1\rangle s_2\langle a_2,b_2\rangle}{\langle a,b\rangle s_1; s_2\langle a_2,b_2\rangle} \quad \text{(SEQ)}$$

$$\frac{}{\langle a,b\rangle next\langle a+1,b+1\rangle} \quad \text{(NEXT)}$$

$$\frac{\langle a_1,b_1\rangle s_1\langle a_2,b_2\rangle \qquad \langle a_1,b_1\rangle s_2\langle a_3,b_3\rangle}{\langle a,b\rangle if\ b\ then\ s_1\ else\ s_2\langle min(a_2,a_3), max(b_2,b_3)\rangle} \quad \text{(ITE)}$$

$$\frac{\langle a,b\rangle s\langle a_1,b_1\rangle}{\langle a,b\rangle async\{s\}\langle a,b\rangle} \quad \text{(ASYNC)}$$

$$\frac{\langle a,b\rangle s\langle a_1,b_1\rangle}{\langle a,b\rangle finish\{s\}\langle a_1,b_1\rangle} \quad \text{(FINISH)}$$

$$\frac{\langle a,b\rangle s\langle a_1,b_1\rangle \qquad loop\ s}{\langle (a_1-a)i+a,(b_1-b)i+b\rangle s^i\langle (a_1-a)(i+1)+a,(b_1-b)(i+1)+b\rangle} \quad \text{(LOOP-INST)}$$

$$\frac{\langle a,b\rangle s_1\langle a_1,b_1\rangle}{\langle a,b\rangle while(b)s_1\langle a,\infty\rangle} \quad \text{(LOOP-GEN)}$$

$$\frac{\langle a,b\rangle s\langle a_1,b_1\rangle}{\langle a,b\rangle do\ \mathbf{N}\ times\ with\ i\ s\langle (a_1-a)N+a,(b1-b)N+b\rangle} \quad \text{(LOOP-COUNT)}$$

$$\frac{}{\langle a,b\rangle_c\ c.drop()\langle a,\infty\rangle_c} \quad \text{(DROP)}$$

Figure 9. Inference rules for PIA

loop, when it is possible to statically determine the number of iterations that the loop might take. DROP indicates that whenever a clock c is dropped, upperbound of the phase interval goes to infinity. The reason is that other activities might be still registered with that clock. However, once the clock is dropped, there is no way to infer in which phase of the clock the statement may execute. The statement, which drops the clock is guaranteed to execute in or after the lower bound associated with it. Hence, all the statements that follow along this control flow path will respect this lower bound.

The advantage of using phase intervals is that it can be computed for each statement separately with respect to some reference point. So, for every procedure one can assume that the phase interval is $\langle 0,0\rangle$ at the beginning. With respect to that, phase intervals associated with each statement can be computed in isolation of other procedures. Once such an analysis is done for a procedure, say foo, whenever foo is called with some phase interval $\langle a,b\rangle$, all we need to do is shift (add) intervals associated with each statement inside foo by $\langle a,b\rangle$. Computation from the scratch is not needed. As shown in Fig. 10, if control can enter foo with phase interval $\langle a_2,b_2\rangle$ and if starting from $\langle 0,0\rangle$ foo finishes with phase interval $\langle a_f,b_f\rangle$ then in the calling context of bar , foo will finish with phase interval $\langle a_2+a_f,b_2+b_f\rangle$. If some other statement s_4 has an interval $\langle a_4,b_4\rangle s_4\langle a_4',b_4'\rangle$ with $b_4' < a2$, we know that s_4 will precede this instance of foo hence they can not be executed in parallel. This also saves us the trouble of figuring out whether any statement inside this instance of foo may execute in parallel with

```
//<a1,b1>
int bar()
{
s1
s2
//<a2,b2>
foo()
        //<a2,b2> + <af,bf> = <a2+af,b2+bf>
s3
}
```

Figure 10. PIA for context sensitive analysis

s_4 or not. Compared to earlier approaches, PIA (phase interval analysis) gives the advantage of context sensitive MHP analysis without inlining the procedures.

Proof of correctness : Algorithm 1 maintains an invariant that lower bound is equivalent to the minimum number of `next` encountered by any control path from A_q and the upper bound is equivalent to the maximum number of `next` encountered by any control path from A_q to the node. Hence, the intervals calculated are correct with respect to the reference node A_q and the instance of the clocks associated with A_q. Let us assume that there are at most r different declaration for the clocks. Note that any of these declarations may be inside a loop giving rise to many instances associated with the same clock variable. We need to show that the order established between nodes n_1 and n_2 is with respect to the same instance of the clock. We take care of it by adding the constraint $\bigwedge_{j=1}^{l} i_j = i'_j$ when A_q is within the scope of loops $L_1, \ldots, L_l$ ensuring that the order computed is always with respect to the clock instance associated with the reference point A_q.

Complexity Analysis : Algorithm 1 proceeds by first computing a topological sort on the subgraph. Topological sort of a directed acyclic graph requires $O(e)$ time where e is the number of edges. For any reducible graph produced by structured programming constructs $O(n) = O(e)$ [2,10] where e is the number of edges and n is the number of nodes or vertices. Phase interval information flows through edges only once, making the processing time for $O(n)$ nodes to be $O(n)$ making amortized constant time required for each node. Finding the phase interval for instances require $O(h)$ time where h is the height of PST as that is the maximum number of nested loops we can have. This is done for every clock requiring $O(r)$ processing time. Hence, for a single node, entire process requires $(O(1) + O(h))(O(r))$. There are $O(n^2)$ pairs making the total complexity to be $O(n^2rh)$. Since, r is the number of different clock declarations, in practice it would be much smaller than the total number of nodes ($r \ll n$) making the complexity to be $O_r(n^2h)$. This is clearly an improvement over earlier work [20] having a complexity bound of $O(n^4)$.

Indirect Synchronization : Let us take an example shown in Fig. 1. Activity B and C does not share a clock, making it impossible to infer ordering in a direct manner. However, advantage of having an ordering is that it is transitive. Using algorithm 1 we can infer that $p_1 < t_2$ as well as $t_2 < s_3$. Making it possible to infer that $p_1 < s_3$. Only condition is that the analysis has to have been done using a common reference point for all nodes. Condition function for $p_1 < s_3$ in this case would be $\phi_{p_1<t_2} \wedge \phi_{t_2<s_3}$. In the given example, condition functions evaluates to $true$. In general, for nodes $n_1, \ldots, n_k$, we can infer $n_1 < n_k$ if with respect to a common reference point we have $\forall_{i=1}^{k-1} n_i < n_{i+1}$. The condition function would just be $\bigwedge_{i=1}^{k} \phi_{n_i<n_{i+1}}$.

There could be times when the static evaluation of the condition function may not be possible. Even then, compiler can generate the code in a manner shown below, where the optimization is conditional depending upon whether condition function is $true$ at runtime. Compiler can introduce auxiliary variables in the generated code to capture instance variables in the condition functions if needed.

```
if ($\phi_{n_1 \nparallel n_2}$)
  generateOptimizedCode()
else
  generateNormalCode()
```

V. Summary

MHP analysis opens up a lot of optimization and debugging opportunities for parallel programs. Application of MHP analysis in concurrent static single assignment, global value numbering, loop invariant detection and data race analysis for parallel programs has already been shown in [20]. The more precise an MHP analysis is, the more optimization opportunity it provides. In this paper we have introduced a new analysis for parallel programs with dynamic barriers called *Phase Interval Analysis*. PIA improves over earlier work [20] by being able to compute condition functions under which two statement instances may not execute in parallel. Condition function is an enriched notion of *condition vectors* introduced in [1]. Reasoning about statement instances in presence of dynamic barriers was not handled by any of the earlier work. As PIA allows us to infer ordering between statements, it enables us to capture indirect synchronization as well which was not possible so far. PIA is a generic analysis technique which can be adapted to various programming languages like X10 and Habanero-Java.

References

[1] Shivali Agarwal, Rajkishore Barik, Vivek Sarkar, and Rudrapatna K. Shyamasundar. May-happen-in-parallel analysis of

x10 programs. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 183–193, New York, NY, USA, 2007. ACM.

[2] A. V. Aho and J. D. Ullman. Node listings for reducible flow graphs. In *Proceedings of seventh annual ACM symposium on Theory of computing*, STOC '75, pages 177–185, New York, NY, USA, 1975. ACM.

[3] R. Barik and V. Sarkar. Interprocedural load elimination for dynamic optimization of parallel programs. In *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, pages 41–52, Sept. 2009.

[4] Rajkishore Barik. Efficient computation of may-happen-in-parallel information for concurrent java programs. In *LCPC*, pages 152–169, 2005.

[5] Rajkishore Barik, Zoran Budimlic, Vincent Cavé, Sanjay Chatterjee, Yi Guo, David M. Peixotto, Raghavan Raman, Jun Shirako, Sagnak Tasirlar, Yonghong Yan, Yisheng Zhao, and Vivek Sarkar. The habanero multicore software research project. In *OOPSLA Companion*, pages 735–736, 2009.

[6] Zoran Budimlic, Vincent Cavé, Raghavan Raman, Jun Shirako, Sagnak Tasirlar, Jisheng Zhao, and Vivek Sarkar. The design and implementation of the habanero-java parallel programming language. In *OOPSLA Companion*, pages 185–186, 2011.

[7] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[8] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

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

[10] Amelia C. Fong and Jeffrey D. Ullman. Finding the depth of a flow graph. *Journal of Computer and System Sciences*, 15(3):300 – 309, 1977.

[11] A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, 1962.

[12] Jonathan K. Lee and Jens Palsberg. Featherweight x10: a core calculus for async-finish parallelism. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel computing*, pages 25–36, New York, NY, USA, 2010. ACM.

[13] Jonathan K Lee, Jens Palsberg, and Rupak Majumdar. Complexity results for may-happen-in-parallel analysis. Unpublished menuscript, 2010.

[14] Lin Li and Clark Verbrugge. A practical mhp information analysis for concurrent java programs. In *of Lecture Notes in Computer Science*, pages 194–208. Springer, 2004.

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

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

[17] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. An efficient algorithm for computing *mhp* information for concurrent java programs. In *ESEC / SIGSOFT FSE*, pages 338–354, 1999.

[18] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, 2000.

[19] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74:358–366, 1953.

[20] Harshit Shah, R. K. Shyamasundar, and Pradeep Varma. Concurrent ssa for general barrier-synchronized parallel programs. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12, May 2009.