OpenMP aware MHP Analysis for Improved Static Data-Race Detection

Utpal Bora*, Shraiysh Vaishay†, Saurabh Joshi‡ and Ramakrishna Upadrasta§
Computer Science and Engineering, Indian Institute of Technology Hyderabad, India
Email: {*cs14mtech11017, †cs17btech11050}@iith.ac.in, {‡sbjoshi, §ramakrishna}@cse.iith.ac.in

***Abstract*—Data races, a major source of bugs in concurrent programs, can result in loss of manpower and time as well as data loss due to system failures. OpenMP, the de facto shared memory parallelism framework used in the HPC community, also suffers from data races. To detect race conditions in OpenMP programs and improve turnaround time and/or developer productivity, we present a data flow analysis based, fast, static data race checker in the LLVM compiler framework. Our tool can detect races in the presence or absence of explicit `barriers`, with implicit or explicit synchronization. In addition, our tool effectively works for the OpenMP target offloading constructs and also supports the frequently used OpenMP constructs.**

We formalize and provide a data flow analysis framework to perform Phase Interval Analysis (PIA) of OpenMP programs. Phase intervals are then used to compute the MHP (and its complement NHP) sets for the programs, which, in turn, are used to detect data races statically.

We evaluate our work using multiple OpenMP race detection benchmarks and real world applications. Our experiments show that the checker is comparable to the state-of-the-art in various performance metrics with around 90% accuracy, almost perfect recall, and significantly lower runtime and memory footprint.

I. Introduction

Data races are a common source of bugs in parallel programs. Data races might result in errors as well as system failures, and these can be both fatal and/or costly. They are also a source of non-determinism and can be extremely difficult to reproduce.

OpenMP is the most frequently used parallel programming model/framework for High Performance Computing (HPC) within a node, and is based on shared memory parallelism. Recently, the focus of OpenMP has been further broadened to provide support features for device offloading. This is so that it can also be used for accelerators that are increasingly adopted by the HPC community. Around 30% of the top 500 supercomputers from the June 2021 list [1] use accelerators.

Programs written in OpenMP may suffer from data races. Considerable amount of effort has been spent in detecting data races in OpenMP programs. For this purpose, both dynamic as well as (purely) static techniques have been proposed in the literature. Most of the current OpenMP race detectors are based on dynamic analysis techniques.

Since OpenMP internally uses POSIX threads (`pthread`), the dynamic race detection tools that target `pthread` based programs—such as HELGRIND [2], VALGRIND DRD [3], and TSAN [4]—can also naturally detect races in OpenMP programs. However, these tools are ill-equipped to handle many high level OpenMP features, either in the form of constructs, or in the form of semantics. Not considering these features makes these tools prone to producing many false positives. Recent works like ARCHER [5] improve the existing `pthread` based tool TSAN by incorporating high-level semantics of OpenMP and subsequently improving the precision and execution time for race detection.

A major drawback of the dynamic race detection tools is that their turnaround time could be exorbitantly large. These tools are based on running the program on selected inputs. This running time is increased further because of the additional instrumentation overhead. The turnaround time for dynamic tools is also determined by its race reporting strategy. If a tool reports only the first race and quits execution immediately, the time taken by it will be lesser when compared to another tool that reports all the possible races encountered in a particular execution. Dynamic tools that follow the latter race reporting strategy have greater turnaround time, as some programs might take hours, or even days to complete execution. This strategy also becomes tricky for race conditions that are dependent on the execution sequence of the threads and reproducing such bugs using dynamic techniques becomes a developer's nightmare.

Static tools, on the other hand, have the advantage of low turnaround time and also provide greater details with respect to the source location and variable name of the offending memory accesses. However, the static race detection tools for OpenMP framework are not mature enough to provide wide coverage, and do not scale with bigger codebases. Also most static tools such as LLOV [6], OMPVERIFY [7], POLYOMP [8], and DRACO [9] employ only a single technique to detect data races; this method may not be adequate in supporting the multitude of features provided by OpenMP. Hence the existing static race detection tools supports limited features of OpenMP as highlighted in Table I.

In the recent work LLOV [6], the authors used the exact polyhedral dependence analysis for race detection. Though this increases the precision of the tool, the limited scope of polyhedral tools limits the applicability of the tool itself. In the present work, we extend LLOV by incorporating multiple techniques (that we have described in Section III) to increase its coverage and improve the accuracy and recall.

In this paper, we propose a method that uses both data flow analysis techniques as well as polyhedral analysis techniques to detect races in OpenMP programs. Our implementation

is based on LLVM [10] and takes race detection in parallel programs way beyond the state-of-the-art techniques [6], [11].

Main Contributions: In this work, we make the following contributions:

- *OpenMP aware MHP analysis*: We provide a novel May-Happen-in-Parallel (MHP) analysis technique for OpenMP programs using a new data flow analysis in LLVM. We compute the Never-Happen-in-Parallel (NHP) sets, the complement sets of MHP, and use this information to statically reason about possible data races present in the OpenMP programs. Our analysis is independent of the number of threads and their execution order, the (parametric) size of the input program, the execution paths, and the system configuration.
- *Phase Interval Analysis (PIA):* We provide a new static analysis technique, (an adaptation of) Phase Interval Analysis (PIA), to compute the execution phases of an OpenMP program. The granularity of the PIA analysis is at basic block level, and it computes the minimum and the maximum phases in which a basic block can execute. We formalize the Phase Interval Analysis framework and provide an implementation of PIA for OpenMP programs using interval analysis over the positive integer domain. Our MHP analysis internally uses the execution phase intervals of the memory accesses computed by PIA.
- *Increased Coverage:* Our tool LLOV (WITH MHP & POLLY) provides support for a wider set of OpenMP constructs in comparison with LLOV as highlighted in Table I. In addition, we overcome the limitations of LLOV and handle the False Negative (FN) cases effectively using PIA.
- *Visualization of PIA:* We provide a framework to generate TASKGRAPH as DOT files to visualize phase intervals of the program. Each basic block in the graph is annotated with the phase in which the program can execute. This helps in manually verifying the data races and improves the productivity of the parallel programmer.
- We have done an extensive evaluation of our tool on a set of race detection benchmarks commonly used in the literature and on multiple open source software with millions of lines of code demonstrating scalability of our tool. Our experiments show that LLOV (WITH MHP & POLLY) is fastest and requires least amount of memory among the state-of-the-art data race checker.

The paper is organized in the following way: We first cover required background in Section II, and we follow it up with analysis and implementation details in Section III. We discuss related work in Section IV, and then we present experimental evaluation in Section V, and finally we summarize in Section VI.

II. BACKGROUND AND MOTIVATION

Parallel processing is ubiquitous in the present world. Modern processors are mostly multiprocessors and modern operating systems support both multiprocessing as well as multithreading to provide better throughput and user experience. To extract parallelism correctly, two programs (or a single program with multithreading) need to follow the Bernstein conditions [12] for parallelism. The program segments running in parallel must preserve the three types of data dependences – namely, flow-, anti-, and output-dependence. When two program segments do not follow these conditions for parallelism, data races might occur resulting in system failure and/or non deterministic system behaviour.

Definition 1 (Data Race): An execution of a concurrent program is said to have a *data race* when two different threads access the same memory location, these accesses are not protected by a mutual exclusion mechanism (e.g., locks), the order of the two accesses is non-deterministic and one of these accesses is a write.

Using static or dynamic program analysis techniques, it is possible to detect conditions in the program that might result in data races. Compilers for most parallel programming languages provide program analysis tools to detect these race conditions. LLOV [6] is one such data race checker for OpenMP programs implemented in the LLVM compiler framework [10].

LLOV can statically detect data races in the most frequently used OpenMP constructs. It uses the Polyhedral model [13] to determine any violation of the dependences due to explicit parallelism. LLOV works on the intermediate representation (IR) of LLVM and hence it can detect data races in C/C++ and FORTRAN programs. LLOV first reconstructs the OpenMP pragmas from the IR and represents them in an in-memory representation. Then it uses LLVM/Polly [14] to model the regions of code marked parallel by the OpenMP pragmas in the polyhedral representation. LLOV then works on the reduced dependence graph (RDG) of the program segment to detect violations of dependences due to user provided explicit parallelism in a loop nest.

LLOV can detect data races in `parallel`, `for`, `simd` pragmas with different data sharing clauses such as `shared`, `reduction`, `private`, `firstprivate`, `lastprivate`, and `threadprivate`. However, LLOV has many limitations and one major drawback is in detecting races across two static control parts (SCoPs). As shown in the example in Listing 1, a thread executing the `parallel for` (Line 3) will not wait for the other threads in the team because of the `nowait` clause (Line 2). Threads that have finished executing the `for` loop are free to continue and execute the `single` construct (Line 5). Since there is a data dependence between write to `a`[i] (Line 4) and read of `a`[9] (Line 6), the program might result in a data race. Such data races due to memory accesses from two different affine regions are missed by LLOV and results in False Negatives (FN).

```
1 #pragma omp parallel shared(b, error) {
2 #pragma omp for nowait
3     for(i = 0; i < len; i++)
4       a[i] = b + a[i]*5;
5 #pragma omp single
6     error = a[9] + 1;
7   }
```

Listing 1: DRB013: example program for which LLOV fails to detect the data race.

LLOV also does not support device offloading constructs and explicit `barriers`. In this work, we use a new static analysis technique, PIA, to overcome the limitations of LLOV. In addition to the FN cases discussed above, we handle many more OpenMP pragmas that are not handled by LLOV as listed in Table I.

III. Analysis and Implementation

We have used LLVM-12.0 [15] compiler infrastructure to implement the new static analyses. LLOV workflow is divided into two phases, analysis and verification. The analysis phase collects OpenMP pragmas from LLVM IR and represents them in an *in-memory representation* described in our prior work [6]. In the current work, we add support for the `barriers` and device offloading pragmas with a focus on the verification phase.

In the verification phase, we construct a static TaskGraph from the LLVM IR and the directives constructed in the analysis phase of LLOV. In the following section, we provide a formal definition, an algorithm, and details about the TaskGraph.

A. Task Graph

Definition 2 (Reduced TaskGraph*):* A reduced TaskGraph $G = \langle V, E, R, T \rangle$ is defined as follows: it consists of a vertex set V which denotes implicit and explicit tasks. Also, its edge set $E \subseteq V \times V$ contains directed edges, a special vertex R for the root of the graph, and a special sentinel vertex T that represents the termination of all the tasks.

A vertex $v \in V$ is either a statement or a basic block in the program. The `barriers` are modeled as standalone vertices, *i.e.,* a basic block will not contain more than one `barrier`.

A directed edge $e \in E$ where $e = (u, v)$ denotes that v can be reached directly from u in some execution of the program.

Since a node is a static representation of a program statement/basic block, it can give rise to multiple instances during an execution.

Definition 3 (Phase): The execution of instances of nodes in the TaskGraph happens in phases. Instances of two nodes $(u, v) \in G$ will execute in two different phases of execution when there exists an implicit or explicit `barrier` separating u and v and one of them dominates the other in the CFG. i.e., each thread executing the nodes must cross the `barrier` in between the nodes. A phase is represented by a positive integer $p \in \mathbb{N}^+ \cup \infty$.

Definition 4 (Phase Interval): To conservatively capture in which phase any instance of $u \in G$ would execute, we associate with it a phase interval $[lb, ub]$, where lb represents the least phase and ub represents the maximum phase in which some instance of the node u can be executed at runtime.

Unlike other task graphs used in the literature where each task is modelled separately [16] or, two instance per task is considered statically [17], we model each task only once and the TaskGraph nodes have one-to-one correspondence with the CFG nodes except for the two special nodes R and T. We then use OpenMP semantics to determine nodes of the TaskGraph that can be executed by more than one thread. A node $u \in G$ can either have only one instance during execution or have multiple instances. Nodes having a single execution instance are marked in light grey while nodes with multiple execution instances are marked in dark grey as shown in Figure 1.

B. Task Graph Construction

The TaskGraph is constructed from the CFG of the program, call graph, and OpenMP semantics. Each node in the TaskGraph corresponds to a basic block in the CFG. In addition to the control flow edges, the TaskGraph also has the function call edges. The TaskGraph also contains edges for OpenMP semantics. For example, for the `single` construct (Line 4) in Listing 2, there will be an edge from the immediate dominator $S1$ (Line 3) of the `single` construct to the immediate post-dominator implicit `barrier` (Line 7) of the `single` construct as all but one of the threads will bypass the `single` construct. This can be seen in Figure 1 that there exists an edge from node `S1` to node `bar1`. Similarly, there exists an edge from node `S3` to node `bar2`.

Each node in the TaskGraph is labelled with a phase interval, for example a phase interval $[lb, ub]$ for node $S1$ denotes that lb is the minimum phase and ub is the maximum phase in which the node $S1$ can execute.

Phase Change: Phase change happens only in one of the following circumstances. Upon entering a parallel region (node `S1` in Figure 1), leaving a parallel region (node `T` in Figure 1), or encountering a `barrier` (nodes `bar1`, `bar2` in Figure 1). The phase change can occur because of an explicit `barrier` or implicit `barrier` present in the constructs- `parallel`, `for`, `workshare`, `single`, `sections`, `taskwait`, `taskgroup`, and `taskloop`.

The phase change for the OpenMP pseudocode in Listing 2 is illustrated in Figure 1. Each node is annotated with the incoming phase interval and the outgoing phase interval, which is computed by the transfer function (discussed later) on the node.

TABLE I: Comparison of OpenMP pragma handling by OpenMP aware tools.(Y for Yes, N for No)

OpenMP Pragma	LLOV (WITH MHP & POLLY)	LLOV	OMPRACER	OMPVERIFY	POLYOMP	DRACO	SWORD	ARCHER	ROMP
#pragma omp parallel	Y	Y	Y	Y	Y	Y	Y	Y	Y
#pragma omp for	Y	Y	Y	Y	Y	Y	Y	Y	Y
#pragma omp parallel for	Y	Y	Y	Y	Y	Y	Y	Y	Y
#pragma omp critical	Y	N	Y	N	N	N	Y	Y	Y
#pragma omp atomic	Y	N	Y	N	N	N	Y	Y	Y
#pragma omp master	Y	N	Y	N	Y	N	Y	Y	Y
#pragma omp single	Y	N	Y	N	Y	N	Y	Y	Y
#pragma omp simd	Y	Y	Y	N	N	Y	N	N	N
#pragma omp parallel for simd	Y	Y	Y	N	N	Y	N	N	N
#pragma omp parallel sections	Y	N	N	N	N	N	Y	Y	Y
#pragma omp sections	Y	N	N	N	N	N	Y	Y	Y
#pragma omp threadprivate	Y	Y	Y	N	N	N	N	Y	Y
#pragma omp ordered	Y	Y	N	N	N	N	N	Y	Y
#pragma omp distribute	Y	Y	Y	N	N	N	N	Y	Y
#pragma omp task	N	N	N	N	N	N	N	Y	Y
#pragma omp taskgroup	N	N	N	N	N	N	N	Y	Y
#pragma omp taskloop	N	N	N	N	N	N	N	Y	Y
#pragma omp taskwait	N	N	N	N	N	N	N	Y	Y
#pragma omp barrier	Y	N	Y	N	Y	N	Y	Y	Y
#pragma omp teams	Y	N	Y	N	N	N	N	N	N
#pragma omp target	Y	N	Y	N	N	N	N	N	N

```
#pragma omp parallel
  {
    // S1
#pragma omp single
    {
      // S2
    } // implicit barrier
    // S3
#pragma omp single
    {
      // S4
    } // implicit barrier
    // S5
  }
```

Listing 2: Multiple OpenMP `single` constructs

C. Phase Interval Analysis

We adapt Phase Interval Analysis (PIA) [18] for OpenMP programs using a data flow analysis framework. We propose a forward flow analysis to compute phase intervals. Each statement/basic-block is assigned a phase interval comprising of the minimum and the maximum phase of execution. We perform a modified interval analysis, where the domain of the analysis is the interval lattice, and the particular abstract domain is the interval polyhedra [19].

Here we formally define the data flow framework for Phase Interval Analysis.

Let, $(I, \sqsubseteq, \sqcup, \sqcap)$ is a lattice over an interval domain. An interval $PI \in I$ is of the form $[lb, ub]$, where $lb, ub \in \mathbb{N} \cup \{0, \infty\}$ and $0 \leq lb \leq ub \leq \infty$. For this lattice, $\top = [0, \infty]$. Any interval $[lb, ub]$ represents $\perp$ whenever $lb > ub$. For two phase intervals $PI_1 = [lb_1, ub_1]$ and $PI_2 = [lb_2, ub_2]$, $PI_1 \sqsubseteq PI_2$ if an only if $lb_2 \leq lb_1 \leq ub_1 \leq ub_2$. Join ($\sqcup$) of two phase intervals $PI_1 = [lb_1, lb_2]$ and $PI_2 = [lb_2, ub_2]$ is defined as $[\min(lb_1, lb_2), \max(ub_1, ub_2)]$. Similarly, meet ($\sqcap$) of two phase intervals $PI_1 = [lb_1, lb_2]$ and $PI_2 = [lb_2, ub_2]$ is defined as $[\max(lb_1, lb_2), \min(ub_1, ub_2)]$. We also define multiplication of a non-negative integer c with $PI = [lb, ub]$ as $c * PI = [c * lb, c * ub]$. Difference between two intervals $PI_1 = [lb_1, ub_1]$ and $PI_2 = [lb_2, ub_2]$ is defined as $PI_1 \Delta PI_2 = [lb_2 - lb_1, ub_2 - ub_1]$. First and second component of an interval PI can be accessed via $PI.first$ and $PI.second$ respectively.

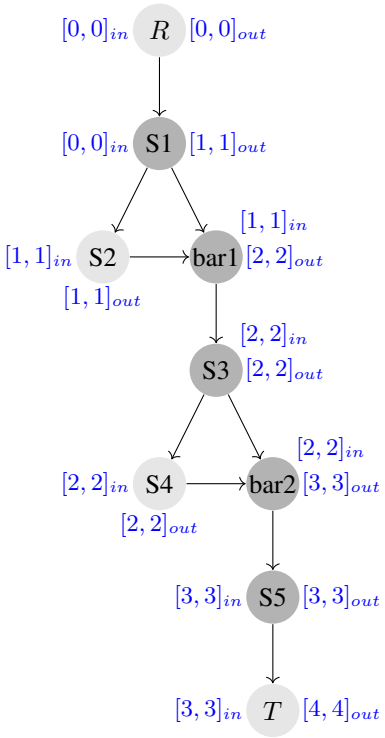

Fig. 1: TASKGRAPH for the OpenMP `single` construct annotated with phase intervals.

Data flow Equation for Phase Interval Analysis for a node $S \in G$ is defined as

$$PI_{in}[S] = \bigsqcup_{p \in pred[S]} PI_{out}[p]$$

where the confluence operator is the join operation ($\sqcup$), p is

an immediate predecessor node of S, and $pred[S]$ is the set of all the immediate predecessor nodes of $S \in G$.

Transfer Function is defined as $PI_{out}[S] = f(PI_{in}[S])$ where f is identity function if the basic block S does not have a `barrier` and, f is $PI_{out}(S) = PI_{in}(S) + [1,1]$ if the basic block S has a `barrier` or S either is starting or exiting block of a parallel construct. The binary operator $+$ is defined as $[lb_1, ub_1] + [lb_2, ub_2] = [lb_1 + lb_2, ub_1 + ub_2]$.

$PI_{out}(S) = f(PI_{in}(S))$

$$f(PI_{in}(S)) = \begin{cases} PI_{in}(S), & \text{Identity function, if } S \\ & \text{does not have a } \texttt{barrier}. \\ PI_{in}(S) + [1,1], & \text{otherwise.} \end{cases}$$

A basic block will have at the most one `barrier` (at the end/beginning). In the case of multiple `barriers` in a basic block, a preprocessing step can split such basic blocks to ensure up to one `barrier` per basic block.

Initialization: All the nodes $u \in G$ are initialized with $PI_{in}[u] = \perp$ and $PI_{out}[u] = \perp$. The root node R is initialized with phase interval $PI_{in}[R] = PI_{out}[R] = [0,0]$.

Widening: Since the interval lattice has infinite elements as well as infinite levels, we need to put a restriction on the height to make the framework terminating. Therefore, we represent ∞ with INT_MAX in our implementation. However, this can be changed with the flag *openmp-pia-lattice-upper-bound*. To reduce time complexity, we apply widening with thresholds using the loop bounds of loops for which the trip count is known. Otherwise widening happens with the default upper bound.

We define widening ($\nabla_{[lbt,ubt]}$) with thresholds lbt and ubt between two intervals PI_1 and PI_2 as $PI_1 \nabla_{[lbt,ubt]} PI_2 = [lbw, ubw]$, where $lbw = lbt$ if $lb_2 < lb_1$, otherwise $lbw = lb_1$, and $ubw = ubt$ if $ub_2 > ub_1$, otherwise $ubw = ub_1$.

For a loop header H, let $PI^1_{in}(H)$ denote the phase interval after the loop body is iterated once by the phase interval analysis. If the trip-count TC is known for a loop, we can safely accelerate computation of $PI_{in}(H) = PI^0_{in}(H) \sqcup (PI^0_{in}(H) + (TC * (PI^0_{in}(H) \Delta PI^1_{in}(H))))$ to capture the effect of the analysis going around the loop TC times. If the trip count TC is not known then $PI_{in}(H) = PI^0_{in}(H) \nabla_{[0,\infty]} PI^1_{in}(H)$.

Termination: The transfer function used for interval analysis is monotone, as it always preserves or increases the interval level in the interval lattice. The flow analysis is guaranteed to terminate [19] due to the widening operation described above.

Algorithm 1 demonstrates the phase interval computation using data flow analysis. It takes the TASKGRAPH G as input and returns the phase intervals of all the nodes in G.

We follow a standard iterative data flow worklist algorithm where the root node is initialized with the interval $[0,0]$ (Line 10). The algorithm runs until the worklist is empty (Lines 11–23), that is, when the framework converges. (Line 13) performs the join operation on the phase intervals of all the immediate predecessors of a node. Widening (Line 15) is applied on a loop header when the phase change $\Delta(PI)$ in one iteration of the loop could be determined. The transfer operation (Line 17) always preserves or increases the phase interval in the interval lattice.

Algorithm 1 Phase Interval Analysis

```
1:  Input: TASKGRAPH G
2:  Output: Phase Intervals PI
3:  Data: worklist
4:  function PIA(G)
5:      worklist.add(G.root)
6:      for all n ∈ G do
7:          PI_out[n] ← ⊥                                   ▷ init to bot
8:          if n has barrier then worklist.add(n) end if
9:      end for
10:     PI_in[G.root] ← [0,0]
11:     while !worklist.empty() do
12:         n ← worklist.pop()
13:         PI_in[n] ← ⨆_{p∈pred[n]} PI_out[p]               ▷ join
14:         if n is loop header then
15:             PI_out[n] ← w(PI_in[n])                     ▷ widen
16:         end if
17:         PI_out[n] ← f(PI_in[n])                         ▷ transfer
18:         if PI_out[n] ≠ PI_in[n] then
19:             for all s ∈ successor(n) do
20:                 worklist.add(s)
21:             end for
22:         end if
23:     end while
24:     return PI_out
25: end function
```

D. May-Happen-in-Parallel Analysis

The PIA described in the above section is used to compute the Never-Happen-in-Parallel (NHP) and its complement May-Happen-in-Parallel (MHP) sets.

We define the binary operator $\bigcap$ on the phase intervals $PI_1 = [lb_1, ub_1]$ and $PI_2 = [lb_2, ub_2]$ as an overlapping function, which returns true if the two intervals overlap partially or fully, and returns false otherwise.

$$PI_1 \bigcap PI_2 = \begin{cases} true, & lb_1 \le ub_2 \le ub_1 \vee \\ & lb_2 \le ub_1 \le ub_2 \\ false, & \text{otherwise.} \end{cases}$$

Instances of two nodes $u, v \in G$ may run in parallel if and only if $PI[u] \bigcap PI[v]$.

Special Case: The `critical` construct needed special treatment since its scope is global. i.e., two `critical` constructs in the global scope can not run in parallel since they share the same lock variable underneath. Therefore, two `critical` constructs will never run in parallel if they share the same lock. Similarly, the `master` construct also needs a special treatment. Two `master` constructs within a `parallel` region will not run in parallel even though they have same phase interval.

E. Race Detection

Once the phase intervals are computed using PIA, potential data races can be detected by checking if the phase intervals of two memory accesses can overlap or not. This means, the source memory access $\texttt{MA}_1$ (in basic block u) and sink memory access $\texttt{MA}_2$ (in basic block v) of a data dependence may potentially race if $PI[u] \bigcap PI[v]$.

LLOV (WITH MHP & POLLY) reports data races with the precise source location of the offending memory accesses. In Listing 3, we show a data race reported by LLOV (WITH MHP & POLLY). It also highlights the source code with line numbers.

```
Data Race detected.
Source : llvm/lib/Transforms/OpenMPVerify/test/10.↩
     race1.c:10:11
Sink : llvm/lib/Transforms/OpenMPVerify/test/10.race1.↩
     c:12:11
==============
9 :      {
10 :  x = 1;
11 : #pragma omp section
12 :  x = 2;
13 :     }
==============
```

Listing 3: Error reporting: LLOV (WITH MHP & POLLY) highlights the source location of the race condition.

F. Limitations

Our PIA analysis in LLOV (WITH MHP & POLLY) does not yet support tasking constructs (as listed in Table I). Although there is no fundamental limitation of the technique to provide support for tasking constructs, we did not explore this aspect in this work. We plan to explore them in the future. Also, the current version of LLOV (WITH MHP & POLLY) does not perform lockset analysis [20] and hence might produce False Positives for memory accesses guarded by exclusive locks.

IV. RELATED WORK

The problem of MHP was first studied by Naumovich et al. [21], [22] for X10 and Java programs. Their technique involves modeling the program as parallel execution graph (PEG) with one CFG per thread, whereas our approach is for OpenMP programs and the TASKGRAPH is independent of the number of runtime threads. Also, they use MHP sets, while our work involves performing phase interval analysis.

Joshi et al. [18] extended the work of Agarwal et al. [23] for programs with dynamic `barriers` and introduced Phase Interval Analysis (PIA) for X10 programs. We adapt PIA for OpenMP semantics and introduce the monotone framework based interval analysis to compute MHP information. Recent works [24], [25] have improved the algorithm by Agarwal et al. [23] for X10 programs.

MHP for OpenMP programs was first explored by Chatarasi et al. [8] using an extended polyhedral framework. Our approach of using interval analysis based on data flow analysis framework is different from the polyhedral phase mapping of the statements incorporated by them.

Extensive work has gone into data race detection in parallel programs using different techniques such as locket analysis based [20], [26]–[30], happens-before [31] relation based [2]–[4], [32], constraint solver based [33], offset span label based [16], [34], polyhedral model based [6]–[9], and MHP information based [8], [17], [18]. A majority of these techniques are either POSIX thread [35] based, or are specific to race detection in explicit parallelism of various programming languages, such as Java, C#, X10, and Chapel.

Race detection in OpenMP parallel programs has been studied recently in LLOV [6], OMPRACER [11], SWORD [36], ROMP [16], DRACO [9], ARCHER [5], POLYOMP [8], and OMPVERIFY [7]. Race detection tools for POSIX threads based parallel programs such as TSAN-LLVM [37], HELGRIND [2], and VALGRIND DRD [3] can also analyze OpenMP programs as OpenMP uses `pthread` underneath it.

We limit the discussion to the state-of-the-art tools that we compare against.

OMPRACER [11] is a static analysis tool that can detect data races in OpenMP programs. OMPRACER employs alias analysis, value-flow analysis using scalar evaluations, and static happens-before graph to detect data races. It constructs the static happens-before graph for all the memory operations and models threads using two nodes per operation in the graph. OMPRACER does not generate any warning for pragmas, such as `task`, `sections`, `ordered`, `doacross`, etc., that are not supported by it.

ARCHER [5] uses both static and dynamic analyses for race detection. It uses happens-before relations which enforces multiple runs of the program to find races. ARCHER reduces the analysis space of `pthread` based tool TSAN-LLVM by instrumenting only parallel sections of an OpenMP program. ARCHER uses OMPT [38] callbacks to instrument the code with happens-before annotations for TSAN-LLVM.

SWORD [36] is a dynamic tool based on operational semantic rules and uses OpenMP tools framework OMPT [38]. SWORD uses locksets to implement the semantic rules by taking advantage of the events tracked by OMPT. SWORD cannot detect races in OpenMP `simd`, `task` and target offloading constructs.

ROMP [16] is a dynamic data race detection tool for OpenMP programs which maintains access history for memory accesses. ROMP builds upon the offset-span-labels of OpenMP threads and constructs task graphs to detect races.

HELGRIND [2] is a dynamic data race detection tool in the Valgrind framework [39] for C/C++ multithreaded programs. HELGRIND maintains *happens-before* relations for each pair of memory accesses and forms a directed acyclic graph (DAG).

VALGRIND DRD [3] is another dynamic race detection tool in Valgrind. It can detect races in multithreaded C/C++ programs. It is based on happens-before relations similar to HELGRIND.

TSAN-LLVM [37] is a dynamic tool based on THREADSANITIZER [4]. TSAN-LLVM uses LLVM to instrument the binaries in place of Valgrind. TSAN-LLVM instrumented binaries incur less runtime overhead compared to THREAD-

SANITIZER. However, it still has similar memory requirements and remains a bottleneck for larger programs.

Tools such as RELAY [29], LOCKSMITH [40], and RACERX [26] use ERASER's [20] lockset analysis algorithm and statically detect races in `pthread` based C/C++ programs. Other tools such as ERASER [20], FASTTRACK [32], CORD [28], and RACETRACK [27] use the lockset algorithm to dynamically detect races in parallel programs. Since they are not specific to OpenMP, we have not discussed them here.

V. EXPERIMENTAL SETUP & RESULTS

In this section, we evaluate our approach on a set of standard data race benchmarks commonly used in the literature. We also study the scalability of our approach on a few real world applications adding up to a million lines of code that use OpenMP for shared memory parallelism.

We compare our work in three different modes- 1) in LLOV (WITH MHP), we enable our new MHP verification engine, 2) in LLOV (WITH POLLY), we enable only Polyhedral verification, and 3) in LLOV (WITH MHP & POLLY), we enable both MHP and Polyhedral verification. In mode 2 and 3, alias analysis checks are disabled whereas they were enabled by default in LLOV [6].

Our toolchain is setup such that both MHP and Polyhedral verification engines are enabled by default. This is same as LLOV (WITH MHP & POLLY). However, one can disable each verification engine using command line flags. Polyhedral verification engine can be disabled with *openmp-verify-use-polly-da=false* and MHP verification engine can be disabled with flag *openmp-verify-use-llvm-da=false*.

System Configuration: Our experimental setup consists of a server with two Intel Xeon E5-2697 v4 processors having a clock frequency of 2.30 GHz. Each processor has 18 hardware cores and 2 hyper-threads per core. The system has total shared memory of 128 GB and is running Ubuntu 20.04.2 LTS (64 bit) and Linux Kernel version 5.4.0-71.

We used LLVM OpenMP runtime version 5.0 for TSAN-LLVM, LLOV, OMPRACER, ARCHER, and SWORD. For HELGRIND, VALGRIND DRD, and ROMP, we used gcc version 9.3.0 and GOMP in our experiments. clang 9.0.1 was required for OMPRACER and clang 12.0.1 was used for the rest.

Performance Metrics: *True Positive* (TP): The outcome of a tool on a kernel with a known data race is considered as TP if the tool reports a race at least once on that kernel in all its runs. If the tool fails to report a race even once on such a kernel then it is considered as *False Negative* (FN).
True Negative (TN): The outcome of a tool on a kernel with no data race is considered as TN if across all its runs on the kernel the tool reports no races. If the tool erroneously reports a race then it is considered a *False Positive* (FP).

In addition, precision, recall, accuracy, F1 score, and diagnostics odds ratio are defined as follows:
Precision = TP / (TP + FP)
Recall = TP / (TP + FN)
Accuracy = (TP + TN) / (TP + FP + TN + FN)
F1 Score = (2 * Precision * Recall) / (Precision + Recall)
Diagnostics Odds Ratio (DOR) = $\frac{LR+}{LR-}$ where,
Positive Likelihood Ratio $(LR+) = \frac{TPR}{FPR}$,
True Positive Rate $(TPR) = \frac{TP}{TP + FN}$,
False Positive Rate $(FPR) = \frac{FP}{FP + TN}$,
Negative Likelihood Ratio $(LR-) = \frac{FNR}{TNR}$,
False Negative Rate $(FNR) = \frac{FN}{FN + TP}$, and
True Negative Rare $(TNR) = \frac{TN}{TN + FP}$.

A. Performance Evaluation

For performance evaluation, we have used DataRaceBench v1.3.2 consisting of kernels for which the absence or presence of a data race is known. We compare our work against the state-of-the-art race detection tools TSAN-LLVM, ARCHER, VALGRIND DRD, HELGRIND, ROMP, SWORD, and OMPRACER. The version numbers and flags used in our experiments are listed in Table II.

DataRaceBench v1.3.2 [41] is a seeded OpenMP benchmark designed for data race detection, and is being commonly used in the literature.

DataRaceBench is a collection of OpenMP programs making use of a variety of OpenMP features and pragmas. The programs exhibit common data race conditions due to various reasons, such as, missing data sharing clauses, input dependent data races, improper synchronizations, and data races that are dependent on number of runtime threads. The benchmark (v1.3.2) consists of 172 kernels out of which 89 kernels do not have any data races and the remaining 83 kernels have known data races. Most of the kernels with races have only one data race each. A few kernels, however, have more than one data race, either in the form of shared induction variables or scalar accesses to shared memory inside a loop.

We consider an outcome of a tool to be TP if it can detect at least one race in the kernels with more than one data races.

For the dynamic tools, our experiments use two parameters similar to Liao et al. [42] and Bora et al. [6]: (i) the number of OpenMP threads, and (ii) the input size for variable length arrays. The number of threads that we considered for the experiments are $\{3, 36, 45, 72, 90, 180, 256\}$. For the 16 variable length kernels, we considered 6 different array sizes as follows: $\{32, 64, 128, 256, 512, 1024\}$. With each particular set of parameters, we ran each of the 172 kernels 5 times. Both the number of threads and array sizes have been used in prior studies [6], [42], and we have used the same setup for uniformity. The 16 kernels with variable length arrays were run 3360 (= 16 kernels $\times$ 7 thread sizes $\times$ 6 array sizes $\times$ 5 runs) times in total. The remaining 156 kernels were run 5460 (156 kernels $\times$ 7 thread sizes $\times$ 5 runs) times in total. For the static tools, we run each kernel 5 times. For every compilation and run, we have used a timeout of 300 seconds for compilation as well as execution separately.

In Table III, column "TP" represents the number of True Positive results reported by a tool. Similarly, columns "FN", "TN", and "FP" represents False Negative, True Negative,

TABLE II: Race detection tools with the version numbers and flags used for comparison

Tools	**Version / Commit**	**Flags**
HELGRIND [2]	3.15.0	*--tool=helgrind*
VALGRIND DRD [3]	3.15.0	*--tool=drd --check-stack-var=yes*
TSAN-LLVM [37]	12.0.1	*ignore_noninstrumented_modules=1*
ARCHER [5]	5b37681	*ignore_noninstrumented_modules=1*
SWORD [36]	7a08f3c	*--analysis-tool=sword-race-analysis*
ROMP [16]	b3e248e	
OMPRACER [11]	0.1.1	*--no-filter --silent --nolimit*
LLOV (WITH POLLY) [6]		*openmp-verify-use-llvm-da=false*
LLOV (WITH MHP)		*openmp-verify-use-polly-da=false*
LLOV (WITH MHP & POLLY)		*-Xclang -disable-O0-optnone*

TABLE III: Maximum number of Races reported by different tools in DataRaceBench v1.3.2

Tools	Race: Yes		Race: No		Coverage	Performance Metrics				
	TP	FN	TN	FP	(172)	Precision	Recall	Accuracy	F1 Score	DOR
HELGRIND	67	4	7	68	146	0.50	0.94	0.51	0.65	1.72
VALGRIND DRD	67	4	33	42	146	0.61	0.94	0.68	0.74	13.16
TSAN-LLVM	**74**	6	48	38	166	0.66	0.93	0.73	0.77	15.58
ARCHER	69	12	**78**	7	166	0.91	0.85	0.89	0.88	64.07
SWORD	51	19	45	6	121	0.89	0.73	0.79	0.80	20.13
ROMP	63	17	74	11	165	0.85	0.79	0.83	0.82	24.93
OMPRACER	66	15	69	12	162	0.85	0.81	0.83	0.83	25.30
OMPRACER (notasks)	65	6	49	11	131	0.86	0.92	0.87	0.88	48.26
LLOV (WITH POLLY)	46	7	24	3	80	**0.94**	0.87	0.88	0.90	52.57
LLOV (WITH MHP)	70	1	45	24	140	0.74	**0.99**	0.82	0.85	131.25
LLOV (WITH MHP & POLLY)	70	1	60	8	139	0.90	**0.99**	**0.94**	**0.94**	**525.00**

and False Positive outcomes respectively. Coverage out of 172 kernels is shown in column "Coverage". The performance metrics for each tool is also shown in a similar fashion. The best numbers are highlighted in bold.

OMPRACER does not support tasking constructs, and there is no option to know at runtime when a pragma is not supported. Hence, we have also added the numbers for OMPRACER (notasks) by excluding the kernels that have tasking constructs. On the other hand, LLOV (WITH MHP & POLLY) explicitly flags when a pragma is not supported. Also, kernels with unhandled pragmas are not taken into account towards its coverage.

It can be seen from Table III that LLOV (WITH MHP & POLLY) performs the best in terms of Recall, Accuracy, F1 Score, and DOR. LLOV (WITH POLLY) has the best Precision, followed by ARCHER and LLOV (WITH MHP & POLLY) closely in third position.

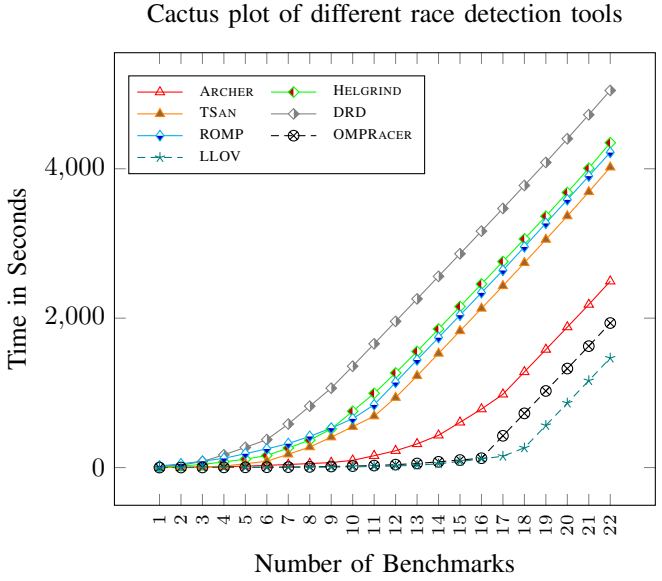

Fig. 2: Cactus plot of different race detection tools on miscellaneous benchmarks listed in Table IV. X axis represents number of benchmarks analyzed, Y axis represents cumulative time.

B. Scalability Analysis

ECP Proxy applications: The Exascale Computing Project (ECP) is a push towards developing an exascale ecosystem for scientific computing in the United States. As part of the ECP project, multiple mini proxy applications were developed to design and test new programming models, technologies, and architectures for supercomputing. We have used the proxy apps AMG, miniAMR, miniQMC, miniVite, SW4lite, and XSBench from the ECP Proxy Apps Suite Release 4.0 which have implementations of parallel algorithms using OpenMP. Other significant proxy applications includes CoMD, LULESH, miniBUDE, miniFE, and RSBench. Proxy applications and their sizes in lines of code are listed in Table IV.

Large Applications: We evaluated LLOV (WITH MHP & POLLY) on multiple large OpenMP applications with thousands of lines of code such as COVID-SIM, Sundials solver, Rodinia, and Parallel Research Kernels as listed in Table IV.

Average maximum RSS of the tools across all the benchmarks

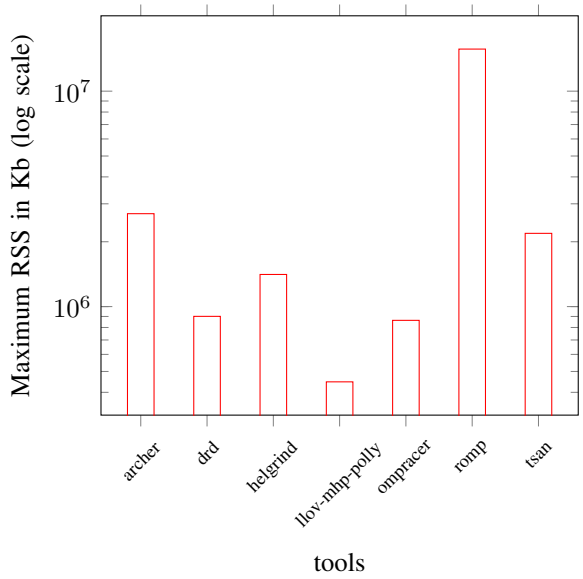

Fig. 3: Average maximum resident set size of the tools across all the 22 benchmarks listed in Table IV

Each of the tools reported combined thousands of data races on the 22 benchmarks. Since these applications have total lines of code exceeding a million, and they are not annotated with known data races, we have not yet validated the True Positive cases. Rather, we analyze the tools based on the time taken to analyze them and their memory requirements. Each of the 22 benchmarks is executed 3 times for each of the race detection tools using 32 OpenMP threads and default program parameters specified in the benchmarks. Timeout of 300 seconds was used for compilation and execution separately.

Figure 2 shows a plot of time taken by various tools for the 22 benchmarks listed in Table IV. Horizontal axis represents the number of benchmarks processed by a tool and vertical axis represents the cumulative time taken (compiletime and runtime for dynamic tools and only compiletime for static tools) to process all the benchmarks. A tool is penalized by the timeout value of 300 seconds if it fails to analyze a benchmark due to compilation error or runtime failure. The side-effect of the timeout/penalty is visible in the plot as one can see a linear increase in time after a certain point for each tool.

Note that LLOV (WITH MHP & POLLY) scaled exceedingly well with the real world applications and could analyze them, on average, in less than 60 seconds including penalties. It can be seen that LLOV (WITH MHP & POLLY) could process maximum number of benchmarks in the least amount of time, closely followed by OMPRACER.

Figure 3 shows the arithmetic mean of the maximum resident set size (RSS) depicting the memory consumption for each of the tools. The memory requirement for LLOV (WITH MHP & POLLY) is significantly low compared to the other tools.

TABLE IV: Number of Lines of Code in 22 Miscellaneous Benchmarks

Benchmark	Lines of Code	Benchmark	Lines of Code
AMG	64.9k	QuickSilver	10k
CoMD	6.1k	RSBench	6k
COVID-SIM	19k	Sundials	186.9k
LCALS	6.6k	sw4lite	54.2k
LULESH	5.5k	TriangleCounting	.8k
miniAMR	17.7k	XSBench	6.7k
miniBUDE	9.5k	Rodinia	186.4k
miniFE	297.8k	ParallelResearchKernels	106.2k
miniQMC	31.9k	OpenMP-Microbench	1k
miniVite	2.4k	openmp-tutorial	10.4k
NBody	0.4k	NAS-Parallel	15.9k

VI. SUMMARY AND FUTURE WORK

We propose a May-Happen-in-Parallel (MHP) based analysis for data race checking of parallel programs written in OpenMP. Our static analysis based data race checker is based on Phase Interval Analysis (PIA).

Through our experiments it is established that while using MHP analysis increases the coverage, it is the combination of MHP as well as the exact dependence analysis provided by Polly which provides the best results in terms of accuracy. We have compared LLOV (WITH MHP & POLLY) with several other state-of-the-art tools, and the results show that this combination provides for much superior overall accuracy.

We also proved the improved scalability of our tool on code bases with a combined lines of code exceeding a million. LLOV (WITH MHP & POLLY) is not only significantly faster on big benchmarks but also has a very small memory footprint as compared to other race-checkers.

In future, we want to improve the coverage of the pragmas by adding support for tasking constructs. We would like to explore using PIA in compiler optimization for OpenMP programs.

REFERENCES

[1] TOP500.Org, "Top 500 Supercomputer Sites," https://top500.org/lists/top500/2021/06/, 2021, [Online; accessed 28-June-2021].

[2] Valgrind-project, "Helgrind: a thread error detector," http://valgrind.org/docs/manual/hg-manual.html, 2007, [Online; accessed 08-May-2019].

[3] ——, "DRD: a thread error detector," http://valgrind.org/docs/manual/drd-manual.html, 2007, [Online; accessed 08-May-2019].

[4] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: Data race detection in practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, ser. WBIA '09. New York, NY, USA: ACM, 2009, pp. 62–71. [Online]. Available: http://doi.acm.org/10.1145/1791194.1791203

[5] S. Atzeni, G. Gopalakrishnan, Z. Rakamarić, D. H. Ahn, I. Laguna, M. Schulz, G. L. Lee, J. Protze, and M. S. Müller, "ARCHER: effectively spotting data races in large openmp applications," in *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*. IEEE Computer Society, 2016, pp. 53–62. [Online]. Available: https://doi.org/10.1109/IPDPS.2016.68

[6] U. Bora, S. Das, P. Kukreja, S. Joshi, R. Upadrasta, and S. Rajopadhye, "LLOV: A Fast Static Data-Race Checker for OpenMP Programs," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 4, Dec. 2020. [Online]. Available: https://doi.org/10.1145/3418597

[7] V. Basupalli, T. Yuki, S. Rajopadhye, A. Morvan, S. Derrien, P. Quinton, and D. Wonnacott, "ompverify: Polyhedral analysis for the openmp programmer," in *OpenMP in the Petascale Era - 7th International Workshop on OpenMP, IWOMP 2011, Chicago, IL, USA, June 13-15, 2011. Proceedings*, ser. Lecture Notes in Computer Science, B. M.

Chapman, W. D. Gropp, K. Kumaran, and M. S. Müller, Eds., vol. 6665. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 37–53. [Online]. Available: https://doi.org/10.1007/978-3-642-21487-5_4

[8] P. Chatarasi, J. Shirako, M. Kong, and V. Sarkar, "An extended polyhedral model for SPMD programs and its use in static data race detection," in *Languages and Compilers for Parallel Computing - 29th International Workshop, LCPC 2016, Rochester, NY, USA, September 28-30, 2016, Revised Papers*, ser. Lecture Notes in Computer Science, C. Ding, J. Criswell, and P. Wu, Eds., vol. 10136. Springer, 2016, pp. 106–120. [Online]. Available: https://doi.org/10.1007/978-3-319-52709-3_10

[9] F. Ye, M. Schordan, C. Liao, P. Lin, I. Karlin, and V. Sarkar, "Using polyhedral analysis to verify openmp applications are data race free," in *2nd IEEE/ACM International Workshop on Software Correctness for HPC Applications, CORRECTNESS@SC 2018, Dallas, TX, USA, November 12, 2018*, I. Laguna and C. Rubio-González, Eds. IEEE, 2018, pp. 42–50. [Online]. Available: https://doi.org/10.1109/Correctness.2018.00010

[10] C. Lattner and V. Adve, "The LLVM Compiler Framework and Infrastructure Tutorial," in *LCPC'04 Mini Workshop on Compiler Research Infrastructures*, West Lafayette, Indiana, Sep 2004.

[11] B. Swain, Y. Li, P. Liu, I. Laguna, G. Georgakoudis, and J. Huang, "OMPRacer: A Scalable and Precise Static Race Detector for OpenMP Programs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20. IEEE Press, 2020.

[12] A. J. Bernstein, "Analysis of programs for parallel processing," *IEEE Transactions on Electronic Computers*, vol. EC-15, no. 5, pp. 757–763, 1966.

[13] P. Feautrier and C. Lengauer, *Polyhedron Model*. Boston, MA: Springer US, 2011, pp. 1581–1592. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_502

[14] T. Grosser, A. Größlinger, and C. Lengauer, "Polly – performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, 2012. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/S0129626412500107

[15] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

[16] Y. Gu and J. Mellor-Crummey, "Dynamic data race detection for openmp programs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. IEEE Press, 2018.

[17] R. Barik, "Efficient computation of may-happen-in-parallel information for concurrent java programs," in *Languages and Compilers for Parallel Computing, 18th International Workshop, LCPC 2005, Hawthorne, NY, USA, October 20-22, 2005, Revised Selected Papers*, ser. Lecture Notes in Computer Science, E. Ayguadé, G. Baumgartner, J. Ramanujam, and P. Sadayappan, Eds., vol. 4339. Springer, 2005, pp. 152–169. [Online]. Available: https://doi.org/10.1007/978-3-540-69330-7_11

[18] S. Joshi, R. K. Shyamasundar, and S. K. Aggarwal, "A New Method of MHP Analysis for Languages with Dynamic Barriers," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, 2012, pp. 519–528.

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

[20] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, Nov. 1997. [Online]. Available: https://doi.org/10.1145/265924.265927

[21] G. Naumovich and G. S. Avrunin, "A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel," in *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '98/FSE-6. New York, NY, USA: Association for Computing Machinery, 1998, p. 24–34. [Online]. Available: https://doi.org/10.1145/288195.288213

[22] G. Naumovich, G. S. Avrunin, and L. A. Clarke, "An efficient algorithm for computing ¡i¿mhp¡/i¿ information for concurrent java programs," in *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-7. Berlin, Heidelberg: Springer-Verlag, 1999, p. 338–354.

[23] S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar, "May-happen-in-parallel analysis of x10 programs," in *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 183–193. [Online]. Available: https://doi.org/10.1145/1229428.1229471

[24] S. Saha and V. K. Nandivada, "On the fly mhp analysis," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 173–186. [Online]. Available: https://doi.org/10.1145/3332466.3374541

[25] A. Sankar, S. Chakraborty, and V. K. Nandivada, "Improved mhp analysis," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 207–217. [Online]. Available: https://doi.org/10.1145/2892208.2897144

[26] D. Engler and K. Ashcraft, "Racerx: Effective, static detection of race conditions and deadlocks," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: Association for Computing Machinery, 2003, pp. 237–252. [Online]. Available: https://doi.org/10.1145/945445.945468

[27] Y. Yu, T. Rodeheffer, and W. Chen, "Racetrack: Efficient detection of data race conditions via adaptive tracking," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, ser. SOSP '05. New York, NY, USA: Association for Computing Machinery, 2005, pp. 221–234. [Online]. Available: https://doi.org/10.1145/1095810.1095832

[28] B. Kasikci, C. Zamfir, and G. Candea, "CoRD: A Collaborative Framework for Distributed Data Race Detection," in *Proceedings of the Eighth USENIX Conference on Hot Topics in System Dependability*, ser. HotDep'12. USA: USENIX Association, 2012, p. 4.

[29] J. W. Voung, R. Jhala, and S. Lerner, "Relay: Static race detection on millions of lines of code," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. New York, NY, USA: Association for Computing Machinery, 2007, pp. 205–214. [Online]. Available: https://doi.org/10.1145/1287624.1287654

[30] P. Pratikakis, J. S. Foster, and M. Hicks, "Locksmith: Context-sensitive correlation analysis for race detection," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 320–331. [Online]. Available: https://doi.org/10.1145/1133981.1134019

[31] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978. [Online]. Available: http://doi.acm.org/10.1145/359545.359563

[32] C. Flanagan and S. N. Freund, "Fasttrack: Efficient and precise dynamic race detection," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 121–133. [Online]. Available: https://doi.org/10.1145/1542476.1542490

[33] P. Chatarasi, J. Shirako, and V. Sarkar, "Static data race detection for spmd programs via an extended polyhedral representation," in *Proceedings of the 6th International Workshop on Polyhedral Compilation Techniques (IMPACT)*, 2016.

[34] J. Mellor-Crummey, "On-the-fly detection of data races for programs with nested fork-join parallelism," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '91. New York, NY, USA: Association for Computing Machinery, 1991, pp. 24–33. [Online]. Available: https://doi.org/10.1145/125826.125861

[35] P. A. J. W. Group, "Ieee standard for information technology–portable operating system interface (posix(r)) base specifications, issue 7," *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*, pp. 1–3951, Jan 2018.

[36] S. Atzeni, G. Gopalakrishnan, Z. Rakamarić, I. Laguna, G. L. Lee, and D. H. Ahn, "SWORD: A bounded memory-overhead detector of openmp data races in production runs," in *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21-25, 2018*. IEEE Computer Society, 2018, pp. 845–854. [Online]. Available: https://doi.org/10.1109/IPDPS.2018.00094

[37] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov, "Dynamic race detection with llvm compiler," in *Proceedings of the Second International Conference on Runtime Verification*, ser. RV'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 110–114. [Online]. Available: https://doi.org/10.1007/978-3-642-29860-8_9

[38] A. E. Eichenberger, J. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, R. Dietrich, X. Liu, E. Loh, and D. Lorenz, "Ompt: An openmp tools application programming interface for performance analysis," in *OpenMP in the Era of Low Power Devices and Accelerators*, A. P. Rendell, B. M. Chapman, and M. S. Müller, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 171–185.

[39] N. Nethercote and J. Seward, "Valgrind: A program supervision framework," http://valgrind.org/, 2003, [Online; accessed 08-May-2019].

[40] P. Pratikakis, J. S. Foster, and M. Hicks, "Locksmith: Practical static race detection for c," *ACM Trans. Program. Lang. Syst.*, vol. 33, no. 1, Jan. 2011. [Online]. Available: https://doi.org/10.1145/1889997.1890000

[41] G. Verma, Y. Shi, C. Liao, B. Chapman, and Y. Yan, "Enhancing dataracebench for evaluating data race detection tools," in *2020 IEEE/ACM 4th International Workshop on Software Correctness for HPC Applications (Correctness)*, 2020, pp. 20–30.

[42] C. Liao, P.-H. Lin, J. Asplund, M. Schordan, and I. Karlin, "Dataracebench: A benchmark suite for systematic evaluation of data race detection tools," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: ACM, 2017, pp. 11:1–11:14. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126958