Detecting Thread-Safety Violations in Hybrid OpenMP/MPI Programs

Hongyi Ma, Liqiang Wang, and Krishanthan Krishnamoorthy

Department of Computer Science, University of Wyoming. {hma3, lwang7, kkrishna}@uwyo.edu

***Abstract*—Hybrid MPI/OpenMP programming is becoming an increasingly popular parallel programming model on High Performance Computing (HPC), where multiple OpenMP threads could execute within a single MPI process. Such a hybrid model is restricted by several rules stated in the MPI standard for correctness. However, it is very tricky to ensure hybrid MPI/OpenMP programs to be thread-safe. There are several concurrency problems happen in hybrid MPI/OpenMP model, such as data race on source and tag information inside of MPI calls and wrong way to synchronize MPI calls using threads. Concurrency errors in MPI/OpenMP model are very difficult to debug due to the complex mixed semantics by MPI routines and OpenMP directives. So developing an useful and efficient tool is necessary to help users debug errors in the hybrid MPI/OpenMP model. In this paper, we propose an approach by integrating static and dynamic analyses to check the potential problems of hybrid MPI/OpenMP programs in order to ensure correctness. In order to obtain thread level information, we instrument monitored variables in the MPI calls in order to obtain both thread and process runtime information. In our approach, the static analysis would help to reduce unnecessary code instrumentation during runtime detection and create the variable checklist for thread-safety checking. In dynamic analysis, both happen-before and lockset-based classical dynamic algorithms are used to check concurrency problems. Static analysis firstly find the monitored variables and MPI calls using control flow graph, then dynamic analysis would take code instrumentation for verifying concurrent races in these monitored variables, at last, our tool would merge the concurrency reports from dynamic analysis analysis to check whether there are violations to thread-safety specifications. Our experimental evaluation over real-world applications shows that our approach is accurate and efficient. The observed average overhead is around 30% using our test experiment setup.**

***Keywords*-Hybrid MPI/OpenMP, Concurrency, Violations, Thread-safety**

I. Introduction

MPI/OpenMP are the two most popular programming models in High-Performance Computing (HPC). MPI is based on a distributed memory model and supports large-scale processes across compute nodes. OpenMP is based on a shared memory model and usually fork multiple threads within a process. It is not easy to write a correct program using MPI and OpenMP due to the tricky semantics of MPI routines and OpenMP directives.

There are two common problems for MPI programs: message race and deadlock. In MPI programs, the process communicate with each other through message-passing and those messages may arrive at a process in a non-deterministic order by variations in process scheduling and network latency. If two or more messages are sent over communication channels on which a receive listens, and they simultaneously in transit without guaranteeing the order of their arrivals, then a message race [14] occurs in the receive event and causes nondeterministic execution of the program. The unexpected message race is difficult for a developer to debug or reproduce without exploring the full program state space. Although most message races are benign without breaking runtime execution, some may lead to incorrect computation and violation of user defined assertions. Deadlock is a more common problem for MPI programs, which is often caused by improper usage of incompatibility of MPI semantics. But in this paper, we only care about how to detect these thread-safety issues instead of pure MPI errors, since some existing work for MPI errors already presented these issues.

The one of current major techniques to detect message races is using dynamic model checking method to replay all different interleavings of MPI application, then check whether there is non-deterministic order for messages receiving. For deadlock, the dynamic graph-based method is used to detect whether there is a state circle inside of execution.

The OpenMP Specifications 3.0[17] briefly reviews the memory consistency model in OpenMP, which is supported by the commodity hardware have been developed in the past few years. Hoeflinger *et al* [8] presents a thorough discussion of the OpenMP model. Sarita Adve *et al* [3] covers both the sequential consistency model and weak memory model in details. These model structure make the concurrency

error happening in OpenMP. A race condition occurs when two or more concurrent threads perform conflicting accesses (*i.e.*, accesses to the same shared variable and at least one access is a write) and the threads use no explicit mechanism to prevent the accesses from being simultaneous.The major techniques to check OpenMP errors include using happen-before dynamic analysis algorithm [16] to analyze runtime execution order, or using symbolic execution [13] to check whether there is a execution trace can have nondeterministic results, like updating with different values by at least two threads. Recently, concolic execution is used to simulate interleaving execution order of multithreaded programs [4].

MPI-2 supports forking multiple threads in an MPI process, which enables a hybrid MPI/OpenMP programming model [6]. However, implementing thread-safety in MPI is not easy because enforcing proper synchronization among threads and processes is very tricky. For example, by default, a hybrid MPI/OpenMP program allows only the master thread to execute within one process if there is no explicit multi-thread specification in `MPI_Init()` (the current one should `MPI_Init_Thread()`. In Figure 1, only MPI_Send or MPI_Recv is executed, but not both. It is difficult to check the error because there is no compilation errors or warning before running. Even there is no problem in the specification of threads in MPI process, but errors may still arise due to improper MPI communication on threads level. For example, Figure 2 shows that two processes are running with two threads in each process. A deadlock may happen nondeterministically in some executions (but not always). Such a scenario violates the thread-safety specification, which requires that all arrival messages within an MPI process should be differentiated by their tags. Messages with different tags will be handled by different threads.In the example of Figure 2, some MPI_Recv could be blocked because the corresponding thread does not obtain an arrival message, as all arrival messages are not differentiated because of the same tag value. A common solution is to use thread ID as tag to distinguish these messages.

In this paper, we propose a hybrid approach to check thread-safety violations for MPI/OpenMP programs that utilizes the results from static analysis to guide the dynamic analysis on error detection. Specifically, our paper makes the following contributions.

- Our approach uses a static analysis that can statically detect potential unsafe hybrid MPI/OpenMP programming styles and monitored and can reduce some error-free region checking for dynamic runtime analysis.
- Our approach uses a static analysis that can report and statistically provide all possible code locations that are involved in errors in Hybrid OpenMP/MPI programs.
- This approach requires little human intervention or annotation, imposes lightweight performance overhead and produces more precise error report than the purely static and dynamic data race detection approaches for MPI/OpenMP.

In this paper, we summarize all concurrency errors

```
MPI_Init();
omp_set_num_threads(2);
#pragma omp parallel
{
  #pragma omp sections
  {
       #pragma omp section
      if (rank == 0 )
          MPI_Send(rank1);
      #pragma omp section
       if (rank ==0)
          MPI_Recv(rank1);
   }
}
```

Figure 1. A Hybrid MPI/OpenMP case study 1

```
 MPI_Init_thread(0,0,MPI_THREAD_MULTIPLE, &
      provided);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 int tag=0;
omp_set_num_threads(2);

#pragma omp parallel for private(i)
 for(j = 0; j < 2; j++) {
    if(rank==0) {
          MPI_Send(&a, 1, MPI_INT, 1, tag,
               MPI_COMM_WORLD);
          MPI_Recv(&a, 1, MPI_INT, 1, tag,
               MPI_COMM_WORLD,
               MPI_STATUS_IGNORE);
    } // rank == 0
   if(rank==1) {
          MPI_Recv(&a, 1, MPI_INT, 0, tag,
               MPI_COMM_WORLD,MPI_STATUS_IGNORE
               );
          MPI_Send(&a, 1, MPI_INT, 0, tag,
               MPI_COMM_WORLD);
    } // rank == 1
 }
```

Figure 2. Hybrid MPI/OpenMP case study 2.

in hybrid MPI/OpenMP programs and categorize them into different scenarios. We propose an approach by integrate static and dynamic program analysis to check these concurrency errors. Specifically, we instrument monitored variables within MPI runtime libraries to specify *write* operations for these shared variable. Our approach first utilizes static analysis to generate the control flow graph of the source code, and to retrieve the arguments in MPI calls inside of OpenMP parallel region, since thread-safety violations only happen in hybrid programming region of source code. We set up monitored variable using our instrumented function to initialize monitored variables for dynamic analysis. Second, linking to our instrumented MPI wrapper during the runtime execution, after getting result from dynamic concurrency analysis, our tool would check monitored variables with the other thread-safety argument list to see whether they have violations in hybrid MPI/OpenMP program. In order to show our accuracy and efficiency, we artificially modified several testing benchmarks from NPB-MZ hybrid MPI/OpenMP benchmark for testing, also we run Intel Thread Checker and Marmot for comparison with our approach.

We build our test bed in Amazon EC2 cloud platform [1] with 32 instance, each instance has 4 cores. Since we only monitor the variables for issues in hybrid MPI/OpenMP,lots of variables in computation are not covered by detection during this study. In our approach overhead is very low and runtime error detection is more efficient than monitoring all writing and reading operations in program. The highest overhead we observed is 45% using 64 processes. Since we tested our approach using NAS parallel benchmark [15],these well-tested benchmarks do not have thread-safety issues mentioned in this paper.So we artifically implemented several tricky errors inside of these benchmarks for the accuracy testing in our approach. Based on the experiments observation, our approach would detect all the violations we constructed in the application with less overhead, which is 50% less when compared with the Intel Thread Checker [18] and Marmot[6].

Our paper is organized as follows. Section II provides existing work for detecting errors in MPI/OpenMP. Section III provides the thread-safety specifications in MPI and constraints for these specifications in our approach. Section IV describes the workflow of our approach, and details the implementation of the static and dynamic analyses to check traces for thread-safety. Section V presents our experimental evaluation over a set of benchmarks and real world applications. Section VI concludes our results and provides directions for future work.

II. Related Works and Background

In recent years, there are several existing tools developed for the error detections in both of MPI and OpenMP programs. But to the best of our knowledge, there are no tools to ensure thread-safety in hybrid MPI/OpenMP programs which can report violations and locate the issues in programs.

In these existing tools, both static and dynamic methods are used to check parallel programs. When it comes to static program analysis approaches, such as model checking and symbolic execution, often suffer state explosion problem due to checking combinatorial number of schedules or reachable states. Like the MPI-SPIN [22], it utilizes model checking and symbolic execution method to detect deadlock. If a property of execution schedule is violated by exploring reachable states of the model, then an explicit violation example is returned to the user with execution traces.

When it comes to dynamic analysis, we can mention DAMPI [5], Marmot[6], Umpire[23], MPI-CHECK [20], and MUST [7] . Umpire, Marmot and MUST rely on a dynamic analysis of MPI calls instrumented through the MPI profiling interface (PMPI). Like, Marmot and MPI-CHECK, they utilize a timeout approach to detect deadlock, and MUST detects deadlock with a scheduling checking. The timeout approach in above tools sometimes would to produce false positives. When it comes to DAMPI, it uses a scalable algorithm based on Lamport Clocks (vector clocks focused on call order) to capture possible non deterministic matches. For each MPI collective operation, participating processes update their clock, based on operation semantics. Umpire, which relies on dependency graphs with additional edges for collective operations to detect deadlocks. In Marmot, an additional MPI process performs a global analysis of function calls and communication patterns. When it comes to MPI-CHECK [20], it instruments the source code at compile time adding extra arguments to MPI calls and it allows collective verification for the full MPI standard. For Umpire, it monitors the MPI operations of an application by interposing itself between the application and the MPI runtime system using the MPI profiling, which increases the overhead in the communication between processes.

Several research works have been proposed to address the error detection issues in OpenMP program. One of the earliest work is the Intel Thread checker [2] which rewrites the program binary code with additional intercepting instructions to monitor the program serial execution and infer possible parallel execution events of the program. However, it lacks specific knowledge about the OpenMP program and is unable to consider the happens-before relationship when checking for the data race in OpenMP programs.Young-Joo Kim *et al* [10] designed a practical tool utilizes the on-the-fly dynamic monitoring to detect the data races in OpenMP programs. Mun-Hye Kang *et al* [9] presents a new tool that focuses on the detection of first data races which are conflicting accesses with no explicit happen-before order in OpenMP programs. The tool first executes the instrumented program in order to obtain the conflicting accesses. Later, it refines the conflicting accesses that are involved in the first data races, by rerunning the program with happened-before analysis. In this paper we show that, our approach is different from other tools. We combine the static analysis results in order to assist on the fly monitoring, and our experiments reveal that the approach is scalable and efficient.

After static and dynamic analysis have been combined for multi-threaded programs, Lee *et la* [11] presents a similar two-phase static/dynamic interprocedural and inter-thread escape analysis. Our previous approach[13] utilized symbolic execution order for simulating the OpenMP multi-threaded runtime behaviors, but this approach has to confront interleaving exploration issues when many threads are encoded in SMT solver.

Our approach performs a static analysis followed by more accurate and faster online dynamic analysis which integrates the information from static analysis. Our tool combines the static analysis for reducing the overhead and providing the check variables for dynamic analysis. In order to improve dynamic analysis accuracy, and using static analysis to retrieve runtime hybrid MPI/OpenMP information without modification semantics on source code level. Then we utilize classical lockset analysis and happen-before analysis to check the concurrency of monitored variables, after merging these concurrency reports into thread-safety specification argument list, violations report would be ready for generation in our tool.

III. Thread-safety in the Hybrid MPI/OpenMP Programming

The MPI standard allows using multiple threads within an MPI process, which is restricted by several rules stated in the MPI standard. This was rened in the MPI-2 standard by introducing thread support. In order to support threads, MPI should be initialized using the following way: (1) `MPI_THREAD_SINGLE` indicates that only one thread is used in MPI process; (2) `MPI_THREAD_FUNNELED` means that multiple threads can exist, but only the main thread can call MPI routines; `MPI_THREAD_SERILIZED` allows multiple threads, but only one thread can call an MPI routine at a time in the current process; (4) `MPI_THREAD_MULTIPLE` allows that multiple threads call MPI routines without restriction. The thread-safety of hybrid MPI/OpenMP programs is vital, otherwise the performance of normal thread behavior can show incorrect.

The correct hybrid MPI/OpenMP application should specify a desired thread level and passes it to the MPI implementation using appropriate above MPI-Thread representations to return to thread level from process level.

A. Thread-safety Properties

According to [6], the violations to thread-safety properties of hybrid MPI/OpenMP programs can be categorized as follows. Totally, we have 6 violations need to detect, which are `isInitializationViolation`, `isMPIFinalizationVoilation` ,`isConcurrentRecvVoilation` , `isConcurrentRequestViolation` , `isProbeViolation` and `isCollectiveCallViolation` . *concurrent* function is results for monitored variables from dynamic concurrency analysis procedure using lockset analysis and happen-before analysis. the return value of *Concurrent(a)* is true, which means a has concurrent execution issues on variable *a* , otherwise, there is no concurrency issues on variable *a* . In order to detect concurrency execution issues of MPI calls at thread level, this *concurrent* function is for monitoring these variables *srctmp*, *tagtmp*, *commtmp*, *collectivecalltmp* and *requesttmp* and *finalizetmp* in MPI calls at thread level. The monitored variable would be introduced more clearly in the MPI wrapper implementation section, since these are inserted into monitored MPI calls for thread-safety violation detection.

- Initialization Violation: Executing MPI calls within threads should follow the specification in MPI thread initialization. For example, if `MPI_THREAD_SINGLE` is specified, `omp parallel` returns false. If `MPI_THREAD_SERIALIZED` is specified, it is not allowed to call MPI routines in two concurrent threads.
  `isInitializationViolation` = $\texttt{MPI\_THREAD\_SINGLE} == true \wedge$ $\texttt{ompparallel} == true) \wedge$ $(\texttt{MPI\_THREAD\_SERIALIZED} == true$ $\wedge (Concurrent(srctmp) \vee$ $Concurrent(tagtmp) \vee$ $Concurrent(commtmp) \vee$ $Concurrent(requesttmp) \vee$ $Concurrent(collectivecalltmp))$ $\wedge (\texttt{MPI\_THREAD\_FUNNELED} == true$ $\wedge \texttt{MPI\_IS\_THREAD\_MAIN}() == false)$

- Concurrent MPI finalizing violation: `MPI_Finalize` should be called in the main thread at each process. In addition, prior to call `MPI_Finalize`, each thread needs to finish existing MPI calls under its own thread to make sure there is no pending communication.
  `isMPIFinalizationVoilation` = $\texttt{MPI\_IS\_THREAD\_MAIN}() == false \wedge$ $mpitype_{tk} = MPIFinialize) \vee$ $Concurrent(terminationtmp) \wedge$ $timestamp(\texttt{MPI\_Finialize}())$ $<$ $timestamp(MPICalls)$
- Concurrent MPI_Recv violation: Each thread within an MPI process may issue MPI calls; however, threads are not separately addressable. In other words, the rank of a send or receive call identifies a process instead of threads, which means when two threads call MPI_Recev with same tag and communicator the order is undefined. In such cases, we can expect a data race. Moreover, we can prevent such data races using distinct communicators or tags for each thread.
  `isConcurrentRecvVoilation` = $Concurrent(srctmp) \wedge$ $Concurrent(tagtmp)$ $\wedge Concurrent(commtmp)$ $\wedge mpitype_{t_1} == MPIRecv \wedge mpitype_{t_2} ==$ $MPIRecv$ $\wedge t_1 \neq t_2$
- Concurrent request violation in MPI_Wait and MPI_Test: MPI does not allow that two or more threads are concurrently invokes $MPI\_Wait(request)$ and $MPI\_Test(request)$ with the same shared request variable.
  `isConcurrentRequestViolation` = $Concurrent(requesttmp) \wedge$ $(mpitype_{t_1} == MPIWait \vee MPITest)$ $\wedge (mpitype_{t_2} == MPIWait \vee MPITest)$ $\wedge (t_1 ! = t_2)$

- Probe Violation in MPI_Probe and MPI_IProbe: Two concurrent invocations of $MPI\_Probe()$ or $MPI\_IProbe()$ from different threads on the same communicator should not have the same "source and "tag as their parameters.
  `isProbeViolation` = $Concurrent(srctmp) \wedge Concurrent(tagtmp)$ $\wedge (mpitype_{t_1} == MPIProbe, MPIIProbe)$ $\wedge (mpitype_{t_2} == MPIRecev) \wedge (t_1 \neq t_2)$

- Collective Call Violation: According to the MPI requirements all processes on a given communicator must make the same collective call. Furthermore, the user is required to ensure that the same communicator is not concurrently used by two different collective calls by threads in the same process.
  `isCollectiveCallViolation` = $Concurrent(collectivetmp) \wedge$ $mpitype_{t_1} == collectiveroutine \wedge$ $mpitype_{t_2} == collectiveroutine \wedge t_1 \neq t_2$
  The lockset analysis and happen-before analysis help to detect concurrency for monitored variables. Later, HOME will deliver the matching rules to detect violations for thread-safety specification.

IV. Detecting Thread-Safety Violations Using Integrated Static and Dynamic Analysis

This paper proposes a novel approach to detect thread-safety violations using the integrated static and dynamic program analysis. Dynamic analysis reasons about behavior of a program through observing its executions. It is usually performed by instrumenting source code or byte/ binary code and monitoring the programs' executions. The observed events can be online analysis (i.e., during executions) or offline (i.e., after executions terminate). In order to detect concurrency errors, dynamic analysis extends the traditional testing techniques. In other words,

it inspects potential concurrency errors by searching specific patterns based on the current observed events, even the errors do not show up in the current execution paths. Most dynamic analysis approaches have the same weakness on path coverage issue, which is that they cannot detect concurrency and logic errors in unexecuted code. Pure Static analysis makes predictions about a program's runtime behavior based on analyzing its source code. Static analysis can often explore the whole code, but sacrifice accuracy and may report many false positives.

In order to avoid reporting false positives and high overhead, we combine static and dynamic analysis techniques together to utilize their advantages and avoid their shortcomings in HOME implementation. Specifically, we design a lightweight technique to check thread-safety violations without sacrificing analysis accuracy and precision. Our proposed hybrid program analysis speculatively approximates the behaviors of unexecuted code by instantiating its static summary using runtime information. In addition, the monitoring overhead is another problem of dynamic analysis, which usually slows down the speed of programs by a factor of 2 to 100. Our proposed approach significantly reduces the overhead by using a static analysis to perform selective monitoring.

A. *Workflow*

Figure 3 shows the architecture overview of our approach HOME, which consists of two phases: compile-time checking and runtime checking. During the compilation phase, we classify code sections into correct and potentially erroneous. The correct code sections are filtered out, and MPI routine calls in the potentially erroneous code are instrumented. This filtering approach avoid systematic instrumentation, thus reducing the overhead of the dynamic analysis. Since compile-time checking procedure does not require running program, so we often call it static analysis. In this compile-time analysis procedure, HOME would generate the control flow graph of this hybrid MPI/OpenMP program. In this way, each MPI call would be represented as a node in control flow graph. At beginning of visiting all the nodes in the control flow graph, we first insert several monitored variables into control flow graph at global variable region. These variables are inserted into MPI wrappers by our instrumented and can help to detect the violations of MPI calls in thread level. The reason is that, since one property in most of the violations to thread-safety specifications in hybrid

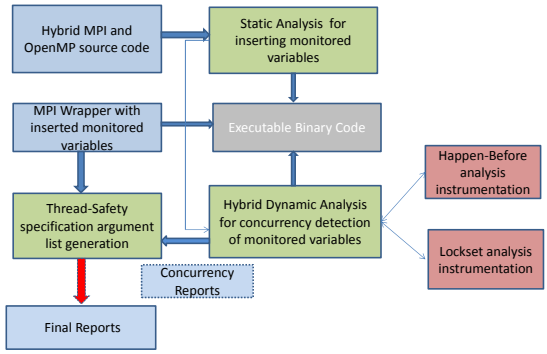

Figure 3. The architecture of the tool HOME.

MPI/OpenMP is about at least two threads execute the same operations at the same time, which can be abstracted to concurrency execution by several threads. By using the MPI wrapper designed by our instrumented MPI library, each monitored variable is associated to one violation to thread-safety specification. If this monitored variable is detected to be a concurrent operations on that, then we can say this MPI calls are executed concurrently by associated threads.

The way to detect the concurrent execution of monitored variables is to combine lockset analysis [21] and happen-before analysis [16], both of these two algorithm are classical algorithm to detect data races in multi-threaded programs. For locksets analysis, the key idea behind it is to track lock sets that govern access to each shared location. A data race on monitored variable is an access to a shared variable that is not governed by a set of locks. The purpose of happens-before analysis is to establish partial ordering of events across different threads. The reason why dynamic analysis procedure combines the algorithm of lockset analysis algorithm and happen-before algorithm is to reduce false positive and overhead on binary code instrumentation. Also, these two data race detection approaches do not require errors real happen in runtime. After the results of these monitored variables are generated during dynamic code instrumentation analysis, then our HOME would analyze the recorded arguments information with the concurrency issues of monitored variables in MPI calls, then match them to determine whether there is a match to violation to thread-safety specification in MPI standard.

B. MPI Wrapper Instrumentation

Our mechanism utilizes the concurrency execution property of monitored variables in MPI calls at thread level to determine whether two MPI calls can be executed at the same time at thread level. In order obtain the runtime information for dynamic analysis, we instrument MPI wrappers that can catch runtime information in MPI call, such as source, tag, communicator, and thread ID information, these MPI wrapper would execute the appropriate MPI call to perform MPI functionality. $MPI\_MonitorVariableSetup$ is the function to For example, in Figure IV-B, our wrapper for MPI_Recv is named as $HMPI\_Recv$, when HOME would detect Concurrent MPI_Recv violation, then dynamic analysis using lockset analysis and happen-before analysis obtains the thread ID execution information, then it would detect the `WRITE` operations on source *src*, tag *tag* and communicator *comm* in MPI calls whether there are concurrent executions happen on *src*, *tag* and *comm* at the same time, if there are concurrent execution issues on these three variables at the same time and have at least two different thread IDs in log. Then we would report there is a violation.

Listing 1. source code

```
#include<mympi.h>

MPI_MonitorVariableSetup
     (srctmp, tagtmp,
     terminationtmp,
     requesttmp,
     collectivetmp);

int main()
{
 code();
 return
}
void code()
{
...
HMPI_Recv(&var, count,
     tag, dest)


}
```

Listing 2. mympi.h

```
#include<mpi.h>

//! different routines
     has its own
     monitored vairbale
//! MPI receive is not
     enough for covering
      all cases
int HMPI_Recv(&var,
     count, src, tag,
     comm)
{
  int tid = getThreadID
       ();
  tagtmp = tag;
  srctmp = src;
  commtmp = comm;
  mpitype = receive;
  StartExecLog();// to
       record all the
       arguments in log
  MPI_Recv(var,count,src
       , tag, comm);
}
```

Listing 3. source code

```
#include<mympi.h>

MPI_MonitorVariableSetup
     (srctmp, tagtmp,
     terminationtmp,
     requesttmp,
     collectivetmp,
     commtmp);

int main()
{
 code();
 return
}
void code()
{
...
HMPI_Recv(&var, count,
     tag, dest)


}
```

Listing 4. mympi.h

```
#include<mpi.h>

//! different routines
     has its own
     monitored vairbale
//! MPI receive is not
     enough for covering
      all cases
int HMPI_Wait(request)
{
  int tid = getThreadID
       ();
  mpitype = mpiwait;
  requesttmp = request;
  StartExecLog();// to
       record all the
       arguments in log
  MPI_Wait(var,count,src
       , tag, comm);
}
```

Listing 5. source code

```
#include<mympi.h>

MPI_MonitorVariableSetup
     (srctmp, tagtmp,
     terminationtmp,
     requesttmp,
     collectivetmp,
     commtmp);

int main()
{
 code();
 return
}
void code()
{
...
HMPI_Recv(&var, count,
     tag, dest)


}
```

Listing 6. mympi.h

```
#include<mpi.h>

//! different routines
     has its own
     monitored vairbale
//! MPI receive is not
     enough for covering
      all cases
int HMPI_Barrier(comm)
{
  int tid = getThreadID
       ();
  mpitype =
       mpicollective;
  commtmp = comm;
  StartExecLog();// to
       record all the
       arguments in log
  MPI_Barrier(comm);
}
```

C. Static Analysis

The idea of overhead reduction is to reduce the number of monitored variables as many as possible during runtime code instrumentation. Since the violations only happen in hybrid MPI/OpenMP region, so non-hybrid region is guaranteed error-free region which have no violations, when it comes to hybrid MPI/OpenMP regions, the violation to thread-safety specification in MPI standard would only happen in this region, so we assume this part is potential error region. For this concept, the static analysis in our approach HOME utilizes the control flow graph of the hybrid MPI/OpenMP program to help distinguish whether the MPI call node is in omp parallel region, which can significantly reduce the overhead of OpenMP binary code instrumentation during the runtime. In Algorithm 1, the node of

Algorithm 1 Static Analysis Procedure

```
1: % source code level pre-processing based on Control Flow Graph %
2:
3: void StaticAnalysis(){
4: src represents the source code of Hybrid MPI/OpenMP program
5: List srcCFG = CFGGeneration(src)
6: add MPI_MonitoredVariables() at beginning of global region
7: While (!srcCFG.isEmpty())
8: if (srcCFG.get(i) == ompParallelBegin()) then
9:     k = i;
10:    While (srcCFG.get(k)!= ompParallelEnd() )
11:    if (srcCFG.get(k).type == MPIcalls) then
12:        replace srcCFG.get(k) with our instrumented MPI call
13:    end if
14:    k++;
15:    it }
16: end if
17: i++;
18: }
19: }
```

source code CFG is put into a list srcCFG, when static analysis procedure traverse all the node is srcCFG, if one node is indicated as *omp parallel* or *omp parallel for*, then the following MPI calls after this omp parallel block would be replaced using our MPI wrapper util it reaches the end of this omp parallel block. The other MPI calls which are not replaced by our MPI wrapper would be skipped during binary code instrumentation in order to reduce the unnecessary overhead.

D. Hybrid Dynamic Analysis

Bases on the monitored variables provided from the static analysis, the list of variable accessed in MPI call in hybrid MPI/OpenMP region sites are instrumented using the tool Intel Pin [12] for monitored variable concurrency detection. Then we run the instrumented program with the dynamic lockset analysis combining with happen-before analysis to detect concurrency issues for monitored variables. Since we would like to check one important property in thread-safety specification in MPI standard, which is whether there two MPI calls can be executed by different threads at the same time.

Since the operations of MPI calls are not categorized in `READ` or `WRITE`, so we have to leverage variables inside of MPI calls at thread level to represent current execution status, so HOME utilizes monitored variables *srctmp*, *tagtmp*, *commtmp*, *collectivecalltmp* and *requesttmp* and *finializetmp* for representing concurrency status in MPI calls. That is the most important innovation for our approach, since the dynamic analysis would detect the concurrency status of these variables to determined there is concurrent execution for MPI calls at thread level or not. Another benefit of using lockset analysis and happen-before analysis is that combination analysis of them do not require these races must happen in the runtime, then some potential violations would be detected no matter it real happens during runtime.

The dynamic analysis techniques in HOME for concurrency detection of monitored variables are fully based on dynamic instrumentation. HOME observes a stream of events generated by instrumentation inserted into the program and sets up several rules to determine concurrency happen conditions for monitored variables. The instrumented program would output a sequence of events to our detection approach. Since pure lockset analysis would find more races then happens-before based tools, but would increase the overhead. That is why we implement happen-before analysis for this step also. In this paper, we treat each event in sequence has following properties:

Table I
TABLE OF NOTATION IN DYNAMIC ANALYSIS

$VarNameMem(m_i, a_i, t_i$	a set of events emory access location is m_i for variable a_i at thread t_i.
t	thread t.
e_i	event at step i.
(a_{t_m}, b_{t_n})	a vector clock for event a and b at thread t_m and t_n
LockSets	a set of locks.
`READ, WRITE`	the two possible access types for a memory access in event.

The Lockset-based analysis approach is implemented based on several hypothesis: Whenever two different threads access a shared memory location, and one of the accesses is a write, the two accesses are performed holding some common lock. Formally, given an access sequence e_i, dynamic analysis procedure in HOME would maintain the lock sets for each monitored variable before step i by a thread t by using $LockSets_i(t)$ for current lockset updates, $LockSets_i(t)$ for each live *thread t*, can be efficiently maintained online as acquisition and release events are received.

`IsPotentialLockSetRace(i, j)` = $e_i = VarNameMem(m_i, a_i, t_i) \wedge e_j = VarNameMem(m_j, a_j, t_j)$

$\wedge\, t_i \neq t_j \wedge m_i = m_j \wedge$
$(a_i = WRITE) \vee (a_j = WRITE) \wedge$
$LockSets_i(t_i) \cap LockSets_j(t_j) = \emptyset$

When it comes to Happens-before, which is a partial order of all events of all threads in a concurrent execution. For any single thread, events are ordered in the order in which they occur. The happens-before relation was first defined by Lamport as a partial order on events occurring in a distributed system[16]. To report the data race more accurately and efficiently, we apply a happen-before analysis to break down the execution of each thread into several periods by the the synchronization events incurred by the thread synchronization, like *omp barrier* directive.

The happens-before relation can be computed online using standard vector clocks, and each thread has a vector maintainces the event order, and all the vectors should have a global order to represent one partial order of execution. For example, thread t_1 has event a happens before event b, then we have $(a_{t1}, 0)$ ¡ $(b_{t1}, 0)$, and thread t_2 has event c happens before event d, then we have $(0, c_{t2})$ ¡ $(0, d_{t2})$, and we supposed that have $(a_{t1}, 0)$ ¡ $(0, c_{t2})$, which means event *a* at thread t_1 happens before event *c* at thread t_2 , but we have no conditions for the order of event *c* and *b*, so there is concurrency issues happen on event c and event d, if both of c and d using the same memory address with *WRITE* within different threads, then we say there is concurrency issue. Then we apply the lockset analysis for these two events again.

The formal representation for happen-before analysis is listed below. `IsPotentialHappenBeforeRace(`e_i`,` e_j`)` =
$e_i = VarNameMem(m_i, a_i, t_i) \wedge e_j = VarNameMem(m_j, a_j, t_j) \wedge$
$t_i \neq t_j \wedge m_i = m_j \wedge$
$(a_i = WRITE) \vee (a_j = WRITE) \wedge$
$\neg(e_{it_i} \rightarrow e_{j_{t_j}}) \wedge \neg(e_{j_{t_j}} \rightarrow e_{it_i})$

There are several challenges in the instrumentation of the OpenMP binary programs using Intel Pin. Since implementation API within Intel Pin only provides memory access *read/write*, synchronization *wait* and locking *lock, unlock* operations, in order to support the OpenMP standard specifies several high-level synchronization points in dynamic analysis, the explicit synchronization points include *#pragma omp barrier* , *#pragma omp critical* and implicit synchronization include *#pragma omp single* should be supported.

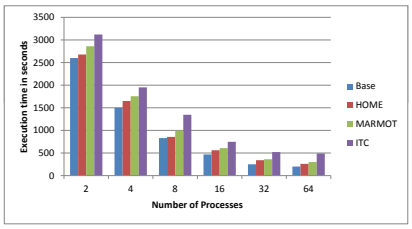

Figure 4. LU-MZ hybrid MPI/OpenMP Testing

V. Experiments

We conducted all the experiments on real cloud environment, Amazon EC2, to demonstrate the our approach HOME is accuracy and efficiency and to measure the overhead of our approach. We summarize major results we observed in this section. The overhead of HOME is ranging from 16% to 45%. We can detect all 6 kinds of violations mentioned in this paper using artificial inserted violations.

A. System Setup

This section discusses our experimental evaluation of HOME over some microbenchmarks for case study and real world applications for scalability test. In Amazon EC2 cluster, our instance type is C3 instances with 2.8 GHz Intel Xeon E5-2680v2 . The numbers, *e.g.*, 8, 16, 32, 64, 128, in the figures of this section represent the number of MPI processes in our cloud platform. We use up to 32 C3 instances totally. Our experiments are performanced on NPB 3.3-MZ in NAS Parallel Benchmark, which includes BT, SP and LU with Class C size. The number of threads is set up to 2 by default in our experiment. Otherwise, the overhead of Intel Thread Checker would be very high with number increasing of threads in processes.

B. Performance Analysis and Comparison

We also compare our approach HOME with Intel Thread Checker[19] and Marmot [6] using evaluation results with injected or modified violations in benchmark. In above Table I, it shows that we inserted 6 violations () into source code programs. *ITC* stands for Intel Thread Checker in this experiment section. The first column lists benchmark names which are LU, BT and SP in NAS-MZ [15]. The second column shows our detected results for all these inserted 6 violations in LU, BT and SP

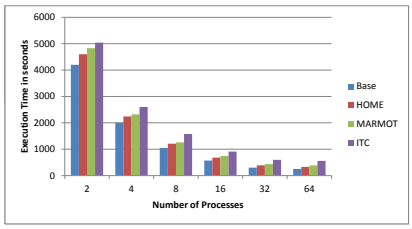

Figure 5. BT-MZ hybrid MPI/OpenMP Testing

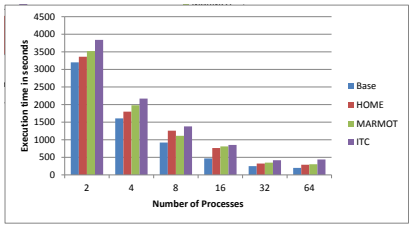

Figure 6. SP-MZ hybrid MPI/OpenMP Testing

benchmark, the experiment shows that our HOME can detect all of these expected violations (termination, communication and so on). For NPB LU, the reason why Intel Thread Checker reports 5 falses is because it cannot recognize *omp critical* directives correctly, so it would not detect violations to *request* in MPI Probe. When it comes to Marmot, since their approaches only depends on the violations which are real happening, so some potential violations would be ignored. There is one false positive happen in BT test using Intel Thread Checker, since the *omp critical* directive can not be recognized correctly, so this single thread execution routine would be reported as concurrent execution using different threads. Also both of the marmot and intel thread checker would detect similar violations like HOME, but the overhead in Marmot and Intel Thread Checker are much higher than that in HOME.

Benchmarks	HOME	ITC	Marmot
NPB-MZ LU (6)	6	5	5
NPB-MZ BT (6)	6	7	6
NPB-MZ SP (6)	6	6	5

The way we add violations

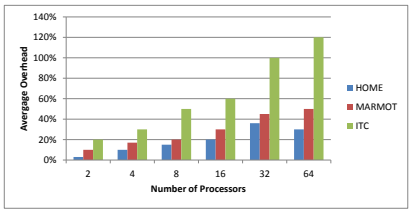

Figure 7. Overhead measurement

In Figure 4, Figure 5 and Figure 6, these figures shows the execution runtime with inserted violations of LU, BT and SP hybrid MPI/OpenMP benchmarks, we only inserted MPI calls within openmp parallel regions without any computation influence on original benchmark semantics, so some extra overhead would also caused by these inserted MPI calls in thread level. The *Base* in the figure means original runtime of application, and *ITC* represents Intel Thread Checker. The runtime time in HOME, Marmot and Intel Checker shown in the Figure 4, 5 and 6 includes overhead caused by instrumentation.

The problem of the Marmot error detection is that it can only detect violations if they actually appear in a run made with MARMOT. It would not find the errors which is a possible violation but not happen during checking runtime. For the MPI calls of a hybrid application different runs may have a different execution order of the MPI calls and it might happen that certain MPI calls are issued by different threads. So this approach would have false negatives on some right execution order but has potential violations applications.

The runtime detection for these errors with very high overhead, since intel thread checker may monitor all the thread level instructions, and the source and tag information in $MPI\_Probe()$ is not detected by intel thread checker. So our approach would have better performance and scalability on these hybrid MPI/OpenMP error checking with less overhead on runtime.

C. Overhead Analysis for HOME

The overhead of HOME in Figure 7 is ranging from 16 % to 45 % based on our experimental observation. The overhead is caused by extra instructions executed in MPI wrapper and binary code instrumentation for dynamic analysis. Since binary code instrumentation is very expensive, so only monitored

variables mentioned for thread-safety specification checking are instrumented during dynamic analysis. The other variables or arguments lists in function call during computation are not considered in this approach, since lots of approaches are developed to detect concurrency errors in pure MPI and pure OpenMP programs.

We proved that our optimization would significantly reduce the lots of overhead. In Figure 7, the overhead is increasing with number of processes in MPI raises. he reason is that our approach requires all the processes run the code instrumentation during runtime, and these checking requires the extra procedure maintenance for lockset analysis and happen-before analysis during the runtime. With the number of processes increasing, the overhead of HOME is increasing also, the reason is that each thread would have to being instrumented during dynamic analysis, so with our observation, overhead of HOME is ranging from around 16% to 45%, when it comes to Marmot it is ranging from 15% to 56%, and overhead it is much higher using Intel Thread Checker which is up to around 200%.

VI. Conclusions and Future Works

In this paper, we present a practical and scalable error detection approach for violations to thread-safety specification in hybrid MPI/OpenMP program, which combines the static analysis on providing monitored variable to improve the dynamic data race detection. Since HOME applies a static analysis to replace MPI calls with our MPI wrappers to get a list of variables that can be involved in the concurrency checking for monitored variables, which are associate with violations in Hybrid MPI/OpenMP. Then the dynamic analysis focus on monitoring these variables in the runtime and apply a enhanced lockset analysis and happen before analysis to detect the concurrency issues on these variables more accurately and efficiently, the overhead is about range from 16% to 45% and each specified variable race is associate with specified violations in thread-safety specification in hybrid MPI/OpenMP application. Our future works include testing HOME's scalability and accuracy on more large-scale benchmarks, which would be incorporating with inter-procedure analysis provided by front-end compiler to produce more refined and precise static analysis results in GUI, extending HOME to handle not only MPI and OpenMP but also the other distributed and shared memory programming model, like UPC and PThreads Programming.

VII. Acknowledgment

The work was supported in part by NSF under Grant 1118059 and CAREER 1054834.

References

[1] Amazon EC2. http://http://aws.amazon.com/ec2/.

[2] Intel thread checker. http://software.intel.com/en-us/intel-thread-checker/.

[3] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29:66–76, December 1996.

[4] P. Garg, F. Ivancic, G. Balakrishnan, N. Maeda, and A. Gupta. Feedback-directed unit test generation for c/c++ using concolic execution. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 132–141, Piscataway, NJ, USA, 2013. IEEE Press.

[5] G. Gopalakrishnan, R. M. Kirby, S. Siegel, R. Thakur, W. Gropp, E. Lusk, B. R. De Supinski, M. Schulz, and G. Bronevetsky. Formal analysis of mpi-based parallel programs. *Commun. ACM*, 54(12):82–91, Dec. 2011.

[6] T. Hilbrich, M. S. Müller, and B. Krammer. Detection of violations to the mpi standard in hybrid openmp/mpi applications. In *Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism*, IWOMP'08, pages 26–35, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] T. Hilbrich, J. Protze, M. Schulz, B. R. de Supinski, and M. S. Müller. Mpi runtime error detection with must: Advances in deadlock detection. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[8] J. P. Hoeflinger and B. R. De Supinski. The openmp memory model. In *Proceedings of the 2005 and 2006 international conference on OpenMP shared memory parallel programming*, IWOMP'05/IWOMP'06, pages 167–177, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] M.-H. Kang, O.-K. Ha, S.-W. Jun, and Y.-K. Jun. A tool for detecting first races in openmp programs. In *PaCT '09: Proceedings of the 10th International Conference on Parallel Computing Technologies*, pages 299–303, Berlin, Heidelberg, 2009. Springer-Verlag.

[10] Y.-J. Kim, M.-Y. Park, S.-H. Park, and Y.-K. Jun. A practical tool for detecting races in openmp programs. In *PaCT*, pages 321–330, 2005.

[11] K. Lee and S. P. Midkiff. A two-phase escape analysis for parallel java programs. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 53–62, New York, NY, USA, 2006. ACM.

[12] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI

'05, pages 190–200, New York, NY, USA, 2005. ACM.

[13] H. Ma, S. R. Diersen, L. Wang, C. Liao, D. Quinlan, and Z. Yang. Symbolic analysis of concurrency errors in openmp programs. In *Proceedings of the 2013 42Nd International Conference on Parallel Processing*, ICPP '13, pages 510–516, Washington, DC, USA, 2013. IEEE Computer Society.

[14] R. H. B. Netzer, T. W. Brennan, and S. K. Damodaran-Kamal. Debugging race conditions in message-passing programs. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, SPDT '96, pages 31–40, New York, NY, USA, 1996. ACM.

[15] Nasa nas parallel benchmarks, OpenMP c versions 2.3. Available from www.nas.nasa.gov/Software/NPB.

[16] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 167–178, New York, NY, USA, 2003.

[17] OpenMP Architecture Review Board. Openmp application program interface. Specification, 2008.

[18] P. Sack, B. E. Bliss, Z. Ma, P. Petersen, and J. Torrellas. Accurate and efficient filtering for the intel thread checker race detector. In *ASID '06: Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, pages 34–41, New York, NY, USA, 2006. ACM.

[19] P. Sack, B. E. Bliss, Z. Ma, P. Petersen, and J. Torrellas. Accurate and efficient filtering for the intel thread checker race detector. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, pages 34–41, New York, NY, USA, 2006. ACM.

[20] E. Saillard, P. Carribault, and D. Barthou. Parcoach: Combining static and dynamic validation of mpi collective communications. *Int. J. High Perform. Comput. Appl.*, 28(4), Nov. 2014.

[21] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, Nov. 1997.

[22] S. F. Siegel. Verifying parallel programs with mpi-spin. In *Proceedings of the 14th European PVM/MPI User's Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 13–14, Berlin, Heidelberg, 2007. Springer-Verlag.

[23] J. S. Vetter and B. R. de Supinski. Dynamic software testing of mpi applications with umpire. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, SC '00, Washington, DC, USA, 2000. IEEE Computer Society.