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—We propose an approach by integrating static and dynamic program analyses to detect thread-safety violations in hybrid MPI/OpenMP programs. We innovatively transform the thread-safety violation problems to race conditions problems. In our approach, the static analysis identifies a list of MPI calls related to thread-safety violations, then replaces them with our own MPI wrappers, which involve accesses to some specific shared variables. The static analysis avoids instrumenting unrelated code, which significantly reduces runtime overhead. In the dynamic analysis, both happen-before and lockset-based race detection algorithms are used to detect races on these aforementioned shared variables. By detecting races, we can identify thread-safety violations according to their specifications. Our experimental evaluation over real-world applications shows that our approach is both accurate and efficient.

Keywords—Hybrid MPI/OpenMP, Thread-safety Violation, Race, Static Analysis, Dynamic Analysis

I. RELATED WORKS AND BACKGROUND

There are many tools for detecting these errors in MPI programs. Both static and dynamic methods are used to detect parallel programs. As examples of static analysis, model checking and symbolic execution often suffer a state explosion problem [14]. As examples of dynamic analysis, DAMPI [4], Marmot[5], Umpire[15] and MPI-CHECK [13]. Marmot utilizes a timeout threshold to determine whether deadlock happens. DAMPI uses a scalable algorithm based on lamport clocks and vector clocks to capture non-deterministic interleaving matches for the communications in MPI program. MPI-CHECK instruments the source code at compile time by inserting extra information to MPI calls and allows collective verification for the full MPI standard.

Several research works have been proposed to address error detection issues in OpenMP program. One of the earliest work is the Intel Thread checker [1] that rewrites the program binary code with additional intercepting instructions to monitor the program serial execution and infer possible parallel execution events of the program. Kim et al [7] designed a practical tool to utilize the on-the-fly dynamic monitoring to detect data races in OpenMP programs. Kang et al [6] presented a tool that focuses on the detection of first data races with no explicit happen-before order in OpenMP programs. The paper [2] implemented the dynamic analysis method to detect the data races in multithreaded programs. OpenMP Analysis Toolkit (OAT) [9] uses Satisfiability Modulo Theories (SMT) solver based symbolic analysis to detect data races and deadlocks in OpenMP codes.

II. OVERVIEW

This paper proposes a novel approach to detect thread-safety violations using the integrated static and dynamic program analysis.

Figure 1 shows the workflow of our approach HAME, which consists of two phases: static analysis and runtime checking. During the static analysis phase, we replace the MPI calls inside OpenMP parallel region, which may affect thread-safety, with our own MPI call wrappers. Such selective instrumentations reduce the overhead of dynamic analysis. To identify MPI calls related to thread-safety properties, our static analysis firstly generates the control flow graph of this hybrid MPI/OpenMP program using the Rose compiler. Consequently, each MPI call would be represented as a node in control flow graph. By visiting all the nodes in the control flow graph, our tool inserts shared variables definitions that will work together with our MPI wrappers. We transform the thread-safety violations to races on monitored variables.

III. STATIC ANALYSIS

To determine whether two MPI calls can be concurrently executed at thread level, as shown below, we replace MPI routines with wrappers that can catch runtime information in MPI calls, such as source, tag, communicator, and thread ID information besides executing the original MPI routine.
Different MPI calls require different monitored variables. Section V gives more details about it.

IV. DYNAMIC ANALYSIS

We use Intel Pin [8] to monitor the shared variables defined MPI wrappers during execution. By analyzing locksets and happen-before orders on the accesses to these variables, we then determine thread-safety violations. Pure lockset analysis would find more accuracy races then happens-before based approaches, but may increase the overhead [11]. Since happen-before relationship is a partial order of events that determines whether the happening order of these events, so it may report false positives due to missing correct orders. Hence, we follow the approach in [11] to combine them in order to make our approach more efficient and more accurate.

Intel Pin can monitor not only memory accesses, but also locking primitives operations for threads synchronization. However, there are several challenges in our implementation of dynamic analysis using Intel Pin. The major issues are how to find happen-before relationships between two events and obtain locksets for each event. To figure out the happen-before relationship between two events, it is necessary to consider the OpenMP synchronization points, which include explicit synchronization points #pragma omp barrier, #pragma omp critical, and implicit synchronization directives such as #pragma omp single, #pragma omp critical, and #pragma omp_set_lock(), #pragma omp_unset_lock() during the implementation.

V. THREAD-SAFETY IN THE HYBRID MPI/OPENMP PROGRAMMING

The MPI standard [3] allows using multiple threads within an MPI process, which is restricted by several rules stated in the MPI standard. The thread-safety issues in hybrid MPI/OpenMP programs is vital, otherwise the performance of normal thread behavior can show incorrect. Here is the checking list of thread-safety specifications in Hybrid MPI/OpenMP programs.

- **Initialization Violation**: Executing MPI calls within threads should follow the specification in MPI thread initialization. For example, if MPI_THREAD_SINGLE is specified, #pragma omp parallel returns false. If MPI_THREAD_SERIALIZED is specified, it is not allowed to call MPI routines in two concurrent threads. To detect initialization violations, we need to monitor the following conditions.

- **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. We check termination variable to see whether there are two MPI calls executing with MPI_Finalize at the same time. A termination variable is inserted into all MPI wrappers.

- **Concurrent MPI_Send 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_Send with same tag and communicator the order is undefined. In such cases, we can expect a data race.

- **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.

- **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. We check src and tag variables in MPI wrappers for race condition.

- 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.

VI. Experiments

In order to measure the overhead and effectiveness of our approach, we conducted all experiments on a virtual cluster on Amazon EC2. The overhead of our approach is ranging from 10% to 40%. Our approach can detect all 6 different kinds of thread-safety violations mentioned in Section V.

A. System Setup

We evaluate HAME over some microbenchmarks for effectiveness and real world applications for scalability test. Our Amazon EC2 virtual cluster uses M3 instances with 2.6 GHz Intel Xeon E5-2670. We use up to 32 M3 instances. Our experiments are based on NPB-MZ in NAS Parallel Benchmark [10], which includes BT-MZ, SP-MZ and LU -MZ with Class C size. The number of threads in each MPI process is set to 2 in our experiment. Otherwise, the overhead of Intel Thread Checker would be very high if using more threads in a process.

B. Performance Analysis and Comparison

We compare our approach with Intel Thread Checker[12] and Marmot [5] by injecting 6 different kinds of violations in the LU, BT and SP benchmarks in NAS-MZ [10]. Table I shows the effectiveness of the three tools. ITC stands for Intel Thread Checker.

The experiment demonstrates that our tool HAME can detect all injected 6 violations. Intel Thread Checker misses MPI_Probe violation on NPB LU. Marmot detects only violations that happen in real executions, thus some potential violations could be missed. When we use Intel Thread Checker, it can produce a false positive in BT testing, because the omp critical directive cannot be recognized correctly. Therefore, this single thread execution routine is reported as concurrent execution by multiple threads. The overhead in Marmot and Intel Thread Checker are much higher than that in HAME.

Table I
THE EFFECTIVENESS COMPARISON OF THREE TOOLS, HAME, ITC, AND MARMOT.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>HAME</th>
<th>ITC</th>
<th>Marmot</th>
</tr>
</thead>
<tbody>
<tr>
<td>NPB-MZ LU (6)</td>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>NPB-MZ BT (6)</td>
<td>6</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>NPB-MZ SP (6)</td>
<td>6</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

Figures 2, 3, and 4 show the execution runtime with inserted violations in the hybrid MPI/OpenMP benchmarks LU, BT and SP, respectively. In addition, during the testing we added MPI calls with
openmp parallel regions without any computation influence from original benchmark semantics. The Base shows the original runtime of application without any instrumentation. The execution durations of our approach, Marmot and Intel Checker are shown in Figure 2, 3 and 4. Figure 2 shows the performance of program using Intel Thread Checker slows down up to 100% or above. The overhead in our approach is increasing when the number of processes is increasing. The programs with Intel Thread Checker always performs the worst performance due to high overhead.

VII. Conclusions

In this paper, we present a scalable approach to detect thread-safety violations in hybrid MPI/OpenMP program. Our experiments results show that the violations detected by our tool is more accuracy than Intel Thread Checker and Marmot in hybrid MPI/OpenMP programs, also we have less overhead than Intel Thread Checker and Marmot. Our observation experiments shows that overhead ranges from 10% to 40%.

Our future work will focus on continuously reducing runtime overhead and improving the scalability. In addition, we plan to investigate more subtle programming errors in hybrid MPI/OpenMP programs and design effective detection algorithms.

References


