Static Analysis and Symbolic Execution for Deadlock Detection in MPI Programs

Craig C. Douglas[1(✉)] and Krishanthan Krishnamoorthy[2]

[1] School of Energy Resources and Department of Mathematics, University of Wyoming, 1000 E. University Avenue, Laramie, WY 82071-3036, USA
craig.c.douglas@gmail.com

[2] Computer Science Department, University of Wyoming, 1000 E. University Avenue, Laramie, WY 82071-3315, USA
krishuwyo@gmail.com

Abstract. Parallel computing using MPI has become ubiquitous on multi-node computing clusters. A common problem while developing parallel codes is determining whether or not a deadlock condition can exist. Ideally we do not want to have to run a large number of examples to find deadlock conditions through trial and error procedures. In this paper we describe a methodology using both static analysis and symbolic execution of a MPI program to make a determination when it is possible. We note that using static analysis by itself is insufficient for realistic cases. Symbolic execution has the possibility of creating a nearly infinite number of logic branches to investigate. We provide a mechanism to limit the number of branches to something computable. We also provide examples and pointers to software necessary to test MPI programs.

1 Introduction

While impossible to determine when an arbitrary parallel program halts or goes into deadlock, which is equivalent to the halting problem [18], there are many real world codes in which a determination of deadlock or non-deadlock is possible [12]. This paper only applies when a determination can be made for parallel programs using MPI [8] though it could be extended to similar communications systems.

Software model checking provides an algorithmic analysis of programs and a fundamental framework to construct a program model [11]. A binary decision diagram (BDD) [3] is one of the ways to construct the model and investigate the state of the program. A BDD is a decision tree that is used to produce output based on a calculation from Boolean inputs [3]. Even though the BDD and model checking techniques are excellent, if the program system has a very large number of states, then it will be difficult to travel all feasible paths. According to Biere et al. [4], the symbolic model checking with boolean encoding can handle large program states faster than other approaches. We use the symbolic model checking technique to model a MPI program and simulate its execution

Y. Shi et al. (Eds.): ICCS 2018, LNCS 10861, pp. 783–796, 2018.
https://doi.org/10.1007/978-3-319-93701-4_62

while analyzing the states of the program. By using a symbolic model we create constraints to find feasible paths to follow the execution of the routines or to detect deadlock. We use the Satisfiability Modulo Theories (SMT) [2] method and symbolic execution in order to travel through the path in our symbolic model.

Consider a trivial example program for two processes. Each process uses MPI_Send to send a message to the other process. Each process uses MPI_Recv to receive the message from the other process. Each process then ends with $MPI\_Finalize$. This program obviously does not deadlock.

Our process removes unnecessary code in order to analyze it. We are left with as little as possible in addition to the MPI calls. Table 1 represents the remaining code. Table 2 represents the steps that the symbolic execution takes in order to determine that this example does not deadlock.

Table 1. Sample non-deadlock MPI routines

Process 0	Process 1
$MPI\_Send[1]$	$MPI\_Send[0]$
$MPI\_Recv[1]$	$MPI\_Recv[0]$
$MPI\_Finalize$	$MPI\_Finalize$

Table 2. Non-deadlock MPI routines with possible execution steps and index

Process 0	Process 1
Step 1–$MPI\_Send[1]$	Step 3–$MPI\_Send[0]$
Step 2, Step 6–$MPI\_Recv[1]$	Step 4–$MPI\_Recv[0]$
Step 7–$MPI\_Finalize$	Step 5–$MPI\_Finalize$

The remainder of the paper is organized as follows. In Sect. 2 we discuss background issues and similar, related research. In Sect. 3 we discuss the computational process used to extract the relevant part of a MPI code and how the symbolic execution operates. In Sect. 4 we define the symbolic model and how symbolic execution works. In Sect. 5 we show an interesting example. In Sect. 6 we provide conclusions and discuss future research.

2 Background and Related Research

Initially, we focused not only detect deadlock on but also looked for a solution to prevent executing deadlocked MPI code. When a user executes a MPI program, it is very difficult to identify the process that cause deadlock due to the missing matching MPI_Send for a MPI_Recv in the source code. Our deadlock

prevention system should not change user data in the code because that can produce wrong output. However, if necessary, we can change the order of the MPI Routine without affecting the final results. Therefore, we started to focus on a different direction for our research and we have conducted many research studies in MPI deadlock and prevention mechanism areas. Since most of the MPI deadlock detection research have only focused on dynamic analysis of MPI program, that technique does not lead to deadlock prevention concepts.

In [10] an idea is proposed to find MPI deadlock using a graph based approach. This research idea is primarily based on the Wait-for graph, which helps to detect deadlock in operating systems and relational database systems. Wait-for graph considers each process as a node and keeps track of processes when a MPI program executes [14]. If MPI_Recv causes deadlock on a process, it locks and holds the resources to the process. Suppose more than a single process is waiting for resources, then there is a possibility of a deadlock. The above method still requires the MPI program to execute in real time. In addition, possible overhead and performance drops can happen in the deadlock detection mechanism if there are lot of MPI Routines available in a MPI source code. Furthermore, the method cannot help prevent deadlock before it happens during the execution. However, the proposed method can be useful if we use it before the MPI program executes.

Based on our research, we can choose either static or dynamic analysis in order to accomplish our research goal. In the remainder of this section we discuss both methods. We choose static analysis over dynamic analysis after conducting several research studies. Also, static analysis provides deadlock detection and can prevent execution of MPI program before a deadlock occurs.

We can analyze a software program in two ways: by static and dynamic analysis. Dynamic analysis is a very common method in software testing. To be effective dynamic analysis requires that the program produce output during the execution.

A model checking system basically is a finite-state automation that can formally verify the concurrent systems and binary decision diagrams [6]. Also, a model checking system is automatic, which means it can verify a program with a high level representation of the user specified model and can check whether the program satisfies the model. Otherwise, the system provides a counterexample if the formula is not satisfied. In addition, model checking can be used in two ways: through dynamic and static analysis.

Dynamic model checking is widely used in race condition and deadlock detection. Wang et al. discussed finding race conditions in multi threaded programs [19]. Also, this research study shows better algorithms to reduce the unnecessary interleaving of thread execution with the model checking and code instrumentation. Gupta et al. explained that there is a significant performance impact on instrumenting functions, which increases the size of the functions instrumented in the source code [9].

As a result, researchers have introduced a framework to accomplish the code instrumentation in better ways and that can reduce overhead while injecting

functions into the source code. So, if we can introduce a similar technology in our research, then the code instrumentation can be very helpful for deadlock avoidance. In addition, the implementation introduces possible ways to inject functions into the source code without changing the context of the MPI program.

Symbolic model checking is used to verify a program in an extremely large scale such that 10^{120} states can be verified, which enables us to perform program analysis through boolean encoding and symbolic behavioral states [5]. Due to this research study our research ideas moved towards the static model checking method. Even though static model checking is suitable for our research, King et al. [16] showed that model checking suffers from the well known state-space explosion problem. This research study introduces a better framework that works with symbolic execution [13], which helps to automate the test case generation and solve the state-space explosion problem efficiently.

3 Computational Process

To do the program analysis using a symbolic model, first we parse the MPI code and extract the information about all of the MPI routines using an Abstract Syntax Tree (AST) [1] that the Rose Compiler [17] generates. We extract the variables and functions from the MPI codes. Then we generate the formulas for our deadlock detection main program. Our main program creates a Yices [20] script in a file that is used by the Yices SMT program.

The main program determines the final result from the output of the Symbolic Execution in Yices. We implemented a validation mechanism that verifies the input file and determines if it has valid MPI function calls so that the Symbolic Execution does not fail due to improper arguments. Then we build formulas for Yices based on the MPI functions.

We currently can analyze a MPI program for a very limited number of MPI functions. The code is extensible in the sense that we can add functions and logic formulas for additional MPI functions, which is part of the future work listed in Sect. 6. When the symbolic model is completed we run it using Yices.

An issue is how long should the Symbolic Execution run in order to find a result from the Yices SMT solver. We specify a last value as symbolic value so the Symbolic Execution only runs until the last value is reached. Determining the specific last value without loss of performance and creating a path explosion problem is a somewhat difficult.

We have introduced a bound variable B (last value) as the maximum integer available when numbering formulas. The formulas are created dynamically and we check the deadlock condition. If we do not have a deadlock conclusion, then we create a formula again with a fresh copy.

4 Symbolic Model and Execution

4.1 The Model

During the extraction process, each MPI function is checked for erroneous parameters. Consider Table 1. It uses a state-space exploration technique. A *state*

includes a process scheduling, current step of a MPI routine, index, and path condition.

The path condition is a component that specifies the order of a MPI routine. In Table 2 at Step 2 when $MPI\_Receive$ executes we change the execution to process 1 and choose Step 3. The path condition is essential in our constraints and is maintained in all steps.

We can show the above *state* components in symbols, such as

$$\text{process scheduling } (p) \wedge \text{ current step } (j) \wedge \text{ index } (i) \wedge \text{ path condition}$$

The *state* is maintained as we execute each MPI routine in the code and we check the logic condition at each step.

We define a token tk for the path condition implementation, which takes a MPI routine for each index of an execution. The token also has the transition implied by the MPI routines to indicate a ready to execute condition for a particular process and index.

We define the variables in a *state* with symbolic values, e.g.,

$$p(process) = <p_0, p_1, \ldots, p_n>,\ i(index) = <i_0, i_1, \ldots, i_n>,\\ \text{and } j(step) = <j_1, j_2, j_3, \ldots, j_n>.$$

For Table 2, $j_i = i$, $i = 1, \cdots, n = 7$.

The process p takes values according to the feasible path condition in the symbolic model, but index i has consistent values that represent the symbolic variable of the current step. Thus, index i is used when creating a fresh formula with a copy of the current step. We continuously create and execute the current step until the symbolic model satisfies the constraints.

If the symbolic model cannot satisfy the constraints for the current step, e.g., at Step n, MPI_Recev cannot find matching MPI_Send at any index i, then that leads to deadlock for the current process. We do not execute the next step until we execute the current step successfully. We create fresh formulas for the current step as necessary for each index i.

$$token[process][index] = transition(MPIRoutines)$$

is denoted by

$$tk_p[i] = \tau_{transition(p)}.$$

The symbolic model must find a feasible path based on the path conditions and MPI routines (cf. Table 2).

We add a buffer to our model that stores the MPI_Send variable required by the $MPI\_Receive$ routine that may execute later in the code. We denote the buffer implementation as follows:

$$buffer[destinationprocess][channel][index] = full \,|\, empty, \text{ or}\\ buf^c_{p'}[i] = full \mid empty.$$

The channel specifies uniqueness of individual routines in each process and prevents overwriting the buffer. The channel implementation is similar to MPI's

virtual communication channels, which allows *buffer* to keep storing routines for a respective channel so MPI_Send and $MPI\_Receive$ can communicate over the channel.

In Table 2 at step 1 when we execute the MPI_Send routine from process 0 we add a constant value that fills the buffer with the destination process (e.g., set $buf_1^1 = 1$). The constant value indicates that the buffer is full. Since our symbolic execution checks the program states in sequential order, it is important to keep track of which process is eligible to run at the current step, e.g., in Table 2 at step 3, the program jumps to process 1 because at the current step process 0 is not eligible to continue further execution.

We require a scheduling mechanism in the symbolic model that takes the eligible process value p for each i, denoted as $s[i] = p$. Consider Table 2. Then

$$\begin{aligned} &\text{Step } i\text{: } s[i] = 0, \text{ for } i = 0, 1, 6, 7 \text{ and} \\ &\text{Step } j\text{: } s[j] = 1, \text{ for } j = 3, 4, 5. \end{aligned}$$

Without a scheduling implementation it is difficult to add the correct MPI routine to *token* and is impossible to travel through the feasible paths in the symbolic model. It is one of the important components in the constraints to make decisions so that the symbolic execution runs correctly. In order to schedule the process we need to make sure that the token has a MPI routine and the current step is eligible to execute (e.g., if the current routine is a $MPI\_Receive$ we need to check if $buffer$ has the value from the matching MPI_Send before we execute the current step).

4.2 MPI Logic Formulas

We can derive formulas for MPI_Send and $MPI\_Receive$. For MPI_Send,

$$\begin{aligned} tk_p[i] = \tau_{send(p)} \wedge buf_{p'}^c[i] \neq full) \Longrightarrow \\ update(s[i] = p) \wedge update(buf_p^c[i+1] = full) \wedge update(buf_{p'}^c[i+2] = empty). \end{aligned}$$

This formula means that at the current index, if the token has a MPI_Send routine and the buffer is not full, then we schedule the process p and update the $buffer$ with the next index ($i = i + 1$). Also, we update the $buffer$ index ($i = i+2$) with the empty value so we prevent overwriting $buffer$. The symbolic execution runs correctly.

For $MPI\_Receive$,

$$\begin{aligned} tk_p[i] = \tau_{recev(p)} \wedge buf_p[i] \neq empty \Longrightarrow \\ (update(s[i] = p)) \vee ((p < p_{max}) \longrightarrow (p = p + 1) \vee (p = 0)). \end{aligned}$$

This formula means that at the current index, if the token has a $MPI\_Receive$ routine and the buffer is not full, then we schedule the current process p. In order to update to the next process we check whether the current process is the last available process (represented by p_max and is 1 in Table 1) or not. If the current process itself is the last one, then we update the next process with 0. Otherwise, we update with next available process.

4.3 Symbolic Execution

Symbolic execution [13] is a program analysis technique that utilizes the symbolic values instead of the absolute values of a program. For all program inputs, symbolic analysis represents the values of program variables as symbolic expressions of those inputs. As the program executes, at each step the state of the program executes symbolically and it includes the symbolic values of program variables at that point. By using the symbolic execution we simulate the program. We use the path constraints and the program counter on the symbolic values to simulate the execution of a program.

While the symbolic execution is one of the better approach simulating a program, it is also difficult to apply to parallel programming methods. For instance, tracking the PC and execution steps in a process is a difficult task and requires sophisticated approaches other than just the conventional symbolic approach. Here we propose a different symbolic approach by introducing several constraints to better resolve the symbolic analysis.

4.4 Symbolic Encoding

We present an encoding approach that converts the symbolic model into Satisfiability Modulo Theories (SMT) formulas [20]. We include scheduling constraints (S_i), transition constraints (T_i), finalize constraints (F_i), and deadlock constraints (D_i):

$$S_i \wedge T_i \wedge F_i \wedge D_i \tag{1}$$

or

$$S_i \wedge T_i \wedge F_i \rightarrow \neg D_i \tag{2}$$

We check all constraints in each execution step. Note that (1) is equivalent to checking the satisfiability for (2). We use Yices as our SMT solver [7] to solve (2). If each formula is satisfiable, then the solution gives trace output that leads to the conclusion. Based on the trace output we can draw a conclusion on whether the given MPI routines are under deadlock condition or not. For example, if all the constraints become true then the deadlock constraints become false, so the given MPI code has no deadlock. Alternately, if any of the constraints become false, then the deadlock constraint is true and we add a value to the deadlock buffer.

Our program shows detailed information about deadlock that will occur in a MPI program. The constraints are the tools for us to solve the formula which is generated by our program.

4.5 Symbolic Variables

In the symbolic analysis, we check deadlock conditions up to a predefined step bound value B. For each step $i < B$, we add a fresh copy for each variable. That

is, $var[i]$ denotes the copy of i at the step. For example, $buff_p^c[i]$ holds values for each step as

$$buf_p^c[0],\ buff_p^c[1],\ buff_p^c[2],\ \cdots,\ buff_p^c[B]$$

and each has a value of $full \mid empty$.

Yices may take additional index i values to solve the formula, which depends on number of MPI routines available and what order those MPI routines are written in the source code. For example, if a MPI source code consists of five MPI routines, then our program may create 12 entries of the formulas with index $i = 11$, but it depends what order the MPI_Send and $MPI\_Receive$ routines are written in the code. If $MPI\_Receive$ appears before the MPI_Send in all the processes then Yices solves the formula and concludes with deadlock with the minimum number of index i value. In that case, the index i value will be equal to the number process available in the code. However, in order to reduce the path explosion, we have optimized the constraints. Therefore, we can reduce the utilization of index i values and prevent solving the same formula over and over with different index i values. If our program finds either deadlock or non deadlock of a MPI code, then we halt the symbolic execution.

Token Variables. The *token* (tk) is used to store a MPI routine in each execution step. During the transition a MPI routine τ in process p and index i has a token, denoted by $tk_p[i] = \tau$. At any step, a single transition per process has a *token*. When τ is executed, then the token moves to next MPI routine. Define $succ(\tau)$ to be the successor of next transition of τ.

Buffer Implementation. Unlike typical programming languages, we cannot store a value in a Yices program. We use the index i, which is used to create a fresh copy of a variable in Yices. We have fresh copy of $buffer$ with current process p for use to store a value. In our symbolic execution $buffer$ is used to store only $full$ or *empty*. We use specific values to represent the $full$ and *empty* values in Yices depending on the context.

In our symbolic analysis we have six kinds of buffers:

1. Scheduling Buffer
2. Schedule Success Buffer
3. Transition Buffer
4. Transfer Buffer
5. Receive Block Buffer
6. Deadlock Buffer.

We use the $Scheduling\ Buffer$ to store the execution step. We ensure that the current step can be scheduled or that it is necessary to move on to the next process. This situation arises when a $MPI\_Receive$ routine is executed. If $MPI\_Receive$ does not find a matching MPI_Send, then we skip the execution in the current process and move to the next process. Otherwise, we fill the

Scheduling Buffer. We use the *Transfer Buffer* to store each transfer that occurred from one process to another when we do not schedule the current process. Hence, we keep a record of the number of the transfer that happened for each *MPI_Receive* in a process, which helps us to find deadlock in the *Deadlock Constraint*.

The *Scheduling Buffer* avoids conflicts between the MPI routines and stores values for a specific channel and execution index. We fill the *Schedule Success Buffer* when a process is selected to execute. We use *Schedule Success Buffer* to indicate the execution of the current process in *Deadlock Constraint*. If the current *MPI_Receive* does not find a matching *MPI_Send* after some execution and the current *Schedule Success Buffer* is empty, then we use *Schedule Success Buffer* and *Receive Block Buffer* in order to identify a potential deadlock in the code. In this case, *Transfer Buffer* is the number of transfers we made for the current *MPI_Receive* when we attempted to find a matching *MPI_Send*.

If the number of transfers exceeds the number of processes available in the MPI code, then we assume that the current *MPI_Receive* will never find a matching *MPI_Send*. Therefore, we update the *Receive Block Buffer* in *Transfer Buffer Constraint*. As a result, *Schedule Success Buffer* and *Receive Block Buffer* both satisfy the *Deadlock Constraint* formula and becomes *true*. Finally, we update the *Deadlock Buffer* and conclude there is a deadlock in the code.

The *Transition Buffer* is used to store the value or tag of the MPI routine that will identify the matching *MPI_Send* or *MPI_Receive*. For example, in Table 2, if step 1 is permitted to execute, then the *Transition Buffer* acquires a value from *MPI_Send* (or a tag) and the value should be the same for the matching *MPI_Receive* in the destination process. The *MPI_Receive* and *Deadlock Buffers* are tied together. Table 3 shows a deadlock situation in step 2 if the *MPI_Receive* cannot find a matching *MPI_Send*. Then the *Transfer Buffer Constraint* adds the current step into the *Receive Block Buffer*, which occurs in step 4. We perform this operation by using the *Transfer Buffer* and we introduce a constraint to check whether *Transfer Buffer* is *full* or *empty*.

Finally, our program concludes as a deadlock if the *Deadlock Buffer* includes one or more *MPI_Receive* routines. If even one *MPI_Receive* is in the *Deadlock Buffer*, then some *MPI_Receive* could not find a matching *MPI_Send*. So the execution will not continue at least for the blocking *MPI_Send* and *MPI_Receive* as in real MPI execution and will be considered as a potential deadlock in the code (Table 4).

The formulas for both *MPI_Send* and $MPI| - Receive$ are quite complex. In [15] are tables that break down the conditions to simple expressions, based on tables, that can be followed to determine correctness.

4.6 MPI Logic Reformulations

The MPI formulas from Sect. 4.2 are reformulated in this section for what they are with the details of this section.

Table 3. Deadlocked MPI routines with possible execution steps

Process 0	Process 1
Step 1–MPI_Send[1]	Step 3, Step 5–MPI_Receive[0]
Step 2, Step 4–MPI_Receive[1]	$MPI\_Receive[0]$
$MPI\_Finalize$	$MPI\_Finalize$

Table 4. Another deadlocked MPI routines with possible execution steps

Process 0	Process 1
Step 1, Step 3–MPI_Receive[0]	Step 2, Step 4–MPI_Receive[0]
MPI_Send[1]	$MPI\_Send[0]$
$MPI\_Finalize$	$MPI\_Finalize$

The main job of the *Scheduling Constraint* is to generate formulas that are responsible for process scheduling. In real MPI execution, each process will execute the MPI routines that belong to the process. Since execution is simulated sequentially, we determine that the current process is eligible to schedule before we execute MPI routines. If the scheduling formula does not execute, then further execution will not take place.

We introduce a program counter (PC) in the MPI constraints. It is used to keep track of duplicate executions of the same MPI routine. In Table 2 after Step 5 and before Step 6, Yices can execute the MPI_Send routine, but it ignores the execution because MPI_Send is already executed successfully in step 1 so we can prevent solving the formula twice and move on to the next step. Therefore, in Table 2 we directly evaluate formulas for $MPI\_Receive$ in Step 6, which helps to minimize the usage of index i and can potentially reduce overhead in our symbolic execution.

The updated formula for MPI_Send is

$$\begin{aligned}&\textstyle\sum_{k=0}^{k=N}(PC_p[k] \neq full \wedge \exists k \in i) \longrightarrow (tk_p[i] = \tau_{send(p')} \wedge \\ &buf_{p'}^{c}[i] \neq full) \longrightarrow update(s[i] = p) \wedge \\ &update(schedule\_success\_buf_p[i] = full) \wedge \\ &update(buf_{p'}^{c}[i+1] = full) \wedge update(buf_{p'}^{c}[i+2] = empty) \vee \\ &(update(buf_{p'}^{c}[i+1] = empty)) \wedge \delta(\{i, \tau, j\}).\end{aligned}$$

The updated formula for $MPI\_Receive$ is

$$\begin{aligned}&\textstyle\sum_{k=0}^{k=N}(PC_p[k] \neq full \wedge \exists k \in i) \longrightarrow (tk_p[i] = \tau_{receive(p)} \wedge \sum_{l=0}^{l=N} buf_p[l] \neq empty) \\ &\longrightarrow (update(s[i] = p) \wedge update(schedule\_success\_buf_p[i] = full)) \vee (((p < p_{max}) \\ &\longrightarrow (update(p_{i+1} = p + 1)) \vee (update(p_{i+1} = 0))) \wedge update(tk_{p+1}[i+1] = \\ &succ(\tau)) \wedge update(transfer\_buf_p[i][j_{i+1}] = full)) \wedge \exists l \in i \wedge \delta(\{p, i, \tau, j\}).\end{aligned}$$

5 Experiments

All experiments were run on a computer with an Intel Core i7 7700K running at up to 4.20 GHz, 16 GB of DRAM, and a 500 GB solid state drive. We used a virtual environment of a VMware workstation player installed under Windows 10 as the host operating system with Ubuntu 16.04 as the guest operating system.

In Table 5 we show experiments taken from deadlocked MPI code. The MPI codes used in our experiments were based on ones the Internet and we also created some complex MPI codes. The codes all fall into deadlock, though not in an obvious manner.

Table 5. Experiments for deadlocked MPI codes

MPI Routines	Time taken for 10 experiments (secs.)									
	1	2	3	4	5	6	7	8	9	10
4	3.049	3.082	3.035	3.306	3.366	3.401	3.339	3.346	3.380	3.301
8	3.361	3.390	3.364	3.330	3.385	3.440	3.279	4.283	3.391	3.437
8	4.575	4.285	4.198	4.745	4.094	4.156	5.117	5.077	4.062	4.076
12	4.102	4.911	4.159	4.024	5.022	4.233	4.201	4.248	4.145	5.363
24	4.007	3.937	3.979	4.078	3.950	4.039	4.064	4.007	4.127	3.945
24	4.186	4.261	4.203	4.223	4.149	4.274	4.357	4.272	4.199	4.330
48	5.127	5.017	5.030	5.107	5.099	5.031	5.155	4.948	5.042	5.085
64	5.761	5.577	5.724	5.804	5.788	5.605	5.715	5.967	5.677	5.854

MPI Routines	Procs.	Average Time
4	2	3.2605
8	2	3.4660
8	3	4.4385
12	3	4.4408
24	3	4.0133
24	4	4.2454
48	5	5.0641
64	6	5.7472

In some contexts we added several processes instead of including many MPI routines in a few processes. We used 2 and 3 processes for 8 MPI routines. Similarly, we used 3 and 4 processes for 24 MPI routines. We tested with different processes to evaluate the time difference between the number of processes. The results show some differences since the symbolic execution may consume more time as the number of processes increase in the MPI code. We observe that when 24 MPI routines are executed the average time for the execution is less than the previous results. The reason for this difference could be among 24 MPI routines the orphan $MPI\_Receive$ is situated in nearly the best case scenario in the MPI code.

According to the Table 5 for the deadlock detection, the best case scenario would be an orphan *MPI_Receive* executed in the first step in process 0. If an orphan *MPI_Receive* executes at the last step in the final process, then it is the worst cast scenario. The average experiment time in Table 5 is the time the main program took to accomplish all of the tasks, which includes parsing the MPI codes, generating the AST using the ROSE compiler, extracting information from the AST and ROSE compiler, generating Yices codes, running symbolic execution in Yices, analyzing Yices output, and generating the conclusion from results.

Table 6 shows the experiment results for a non-deadlock MPI code. Time consumption for the 24 MPI routines case is higher when compared to Table 5. Since the MPI code is not under deadlock, Yices must run symbolic execution until it finds the last MPI routine in the final process. Hence, Yices consumes more time than running symbolic execution in a similar deadlocked MPI code.

Table 6. Experiments for non-deadlock MPI code

MPI Routines	Time taken for 10 experiments (secs.)									
	1	2	3	4	5	6	7	8	9	10
4	4.88	3.72	3.69	3.58	3.60	3.46	3.71	3.64	3.76	3.60
8	4.17	4.23	4.13	4.11	4.25	4.17	4.15	4.32	5.06	4.19
8	3.65	3.63	4.00	3.67	3.41	3.81	3.56	3.64	3.52	3.48
12	6.79	6.86	6.77	7.20	6.69	6.55	6.70	6.78	6.94	6.76
24	70.39	69.62	69.20	71.32	75.42	72.58	73.40	72.33	71.07	70.14
24	83.94	83.75	77.97	79.6	77.60	79.44	76.72	77.06	77.61	76.86
48	73.02	74.16	75.56	73.76	80.34	77.53	80.91	73.80	74.38	76.53
64	105.11	130.01	105.70	103.30	103.29	107.56	106.71	103.97	104.34	103.64

MPI Routines	Procs.	Average Time
4	2	3.76
8	2	4.28
8	3	3.64
12	3	6.80
24	3	71.55
24	4	79.06
48	5	76.00
64	6	107.36

6 Conclusions and Future Work

We have proposed a novel approach to find deadlock in simple MPI codes using static analysis and symbolic execution. We chose static analysis over dynamic analysis because it helps to verify a program of extremely large scale plus we can

find deadlock in MPI programs without numerous executions of the code. Static analysis allows analysis of MPI codes by using static model checking techniques. To perform the static model checking we construct a symbolic model that is the basic element for building the constraints and formulas. Symbolic Execution runs the formulas that we create from constraints in the Yices SMT solver.

Also, in this research we delivered a deadlock detection program that can find deadlock in MPI codes that include only basic MPI communicative routines, e.g., MPI_Send and $MPI\_Receive$. Future research will enable many more MPI routines, such as $MPI\_Barrier$, MPI_Isend, $MPI\_Ireceive$, etc. into our deadlock detection mechanism.

Acknowledgments. This research was supported in part by grants DMS-1722692, ACI-1541392, and ACI-1440610 from the National Science Foundation.

References

1. Aho, A.V., Ullman, J.D.: Principles of Compiler Design. Addison-Wesley, Boston (1977)
2. Barrett, C., Sebastiani, R., Seshia, S., Tinelli, C.: Satisfiability modulo theories. In: Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 825–885. IOS Press (2009)
3. Becker, B., Drechsler, R.: Binary Decision Diagrams: Theory and Implementation. Springer, Heidelberg (1998). https://doi.org/10.1007/978-1-4757-2892-7
4. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49059-0_14
5. Chou, C.N., Ho, Y.S., Hsieh, C., Huang, C.Y.: Symbolic model checking on systemc designs. In: DAC Design Automation Conference 2012, pp. 327–333. IEEE Press (2012)
6. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. **16**, 1512–1542 (1994)
7. Elwakil, M., Yang, Z., Wang, L., Chen, Q.: Message race detection for web services by an SMT-based analysis. In: Xie, B., Branke, J., Sadjadi, S.M., Zhang, D., Zhou, X. (eds.) ATC 2010. LNCS, vol. 6407, pp. 182–194. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16576-4_13
8. Gropp, W., Lusk, E.: Using MPI: Portable Parallel Programming with the Message-Passing Interface. Scientific and Engineering Computation, 3rd edn. MIT Press, Cambridge (2014)
9. Gupta, S., Pratap, P., Saran, H., Arun-Kumar, S.: Dynamic code instrumentation to detect and recover from return address corruption. In: Proceedings of the 2006 International Workshop on Dynamic Systems Analysis, WODA 2006, pp. 65–72. ACM, New York (2006)
10. Hilbrich, T., de Supinski, B.R., Schulz, M., Mueller, M.S.: A graph based approach for MPI deadlock detection. In: Proceedings of the 23rd International Conference on Supercomputing, ICS 2009, pp. 296–305. ACM, New York (2009)
11. Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surv. **41**, Article ID 21 (2009)
12. Jiang, B.: Deadlock detection is really cheap. ACM SIGMOD Rec. **17**, 2–13 (1988)

13. King, J.C.: A new approach to program testing. In: Hackl, C.E. (ed.) IBM 1974. LNCS, vol. 23, pp. 278–290. Springer, Heidelberg (1975). https://doi.org/10.1007/3-540-07131-8_30
14. Kitsuregawa, K.M., Tanaka, H.: Database Machines and Knowledge Base Machines. Springer, New York (1988). https://doi.org/10.1007/978-1-4613-1679-4
15. Krishnamoorthy, K.: Detect Deadlock in MPI programs using static analysis and symbolic execution. Master's thesis, University of Wyoming, Computer Science Department, Laramie, WY (2017)
16. Khurshid, S., Păsăreanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 553–568. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36577-X_40
17. rosecompiler.org: ROSE compiler. http://www.rosecompiler.org/. Accessed 3 Mar 2018
18. Turing, A.: On computable numbers, with an application to the entscheidungsproblem. Proc. Lond. Math. Soc. **42**, 230–265 (1937)
19. Wang, C., Yang, Y., Gupta, A., Gopalakrishnan, G.: Dynamic model checking with property driven pruning to detect race conditions. In: Cha, S.S., Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 126–140. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88387-6_11
20. yices.csl.sri.com: The Yices SMT solver.http://yices.csl.sri.com/. Accessed 3 Mar 2018