RISKS OF USING MESSAGE PASSING INTERFACE (MPI) PARADIGM

Alaa Ismail El-Nashar\(^1\), Hussain Abo Surrah\(^2\)

\(^1\)Associate professor, Computer Science Department, Minia University, Egypt
Information Technology Department, College of Computers and Information Technology, Taif University, Saudi Arabia (nashar_al@yahoo.com)
\(^2\)Assistant Professor, IT Department, CIT, Taif University, SA, (Salama366@yahoo.com)

Abstract— Message Passing Interface (MPI) is a widely used paradigm in writing parallel programs since it can be employed not only within a single processing node but also across several connected ones. Using MPI paradigm is a risky process due to the inter-process communication among the running parallel processes. In this paper, we classified the risks related to MPI programming. An experimental method to predict the speed up of MPI programs is introduced. A technique for Analyzing MPI programs has been presented. Finally, non-determinacy and deadlock arise in MPI programs have been checked using Maud system.

Keywords: Parallel programming, Message Passing Interface, performance, data flow, control flow, program correctness, parallel bugs.

1. INTRODUCTION

Various parallel programming paradigms can be used to write parallel programs such as open MP [13], Parallel Virtual Machine (PVM) [35], and Message Passing Interface (MPI) [34]. Message Passing Interface (MPI) is a widely used paradigm in writing parallel programs since it can be employed not only within a single processing node but also across several connected ones.

The goal of MPI is to establish a portable, efficient, and flexible standard for message passing to be used for writing message passing programs, and providing an appropriate environment for general purpose message-passing programs, especially programs with regular communication patterns.

MPI is a library of subroutines that enables programmers to identify and implement explicit parallelism using special constructs. It has been implemented for almost every distributed memory architecture and speed.

In message passing paradigm, several separate processes used to complete the overall computation. In this scheme, many concurrent processes are created, and all of the data involved in the calculation is distributed among them using different ways. There is no shared data; when a process needs data held by another one, the second process must send it to the first process. An MPI message passing protocol describes the internal methods and policies an MPI implementation employs to accomplish message delivery.

There are two common message passing protocols, eager and rendezvous [28]. Eager protocol is an asynchronous protocol that allows a send operation to complete without acknowledgement from a matching receive. Rendezvous protocol is a synchronous protocol which requires an acknowledgement from a matching receive in order to complete the send operation.

Since MPI enables the programmer to control both data distribution and also process synchronization, writing MPI parallel programs is a risky task and hence the program correctness will be negatively affected unless it is coded carefully concerning these requirements.

We focus on the parallel programs written by MPI paradigm using MPICH2 [27] which is an MPI open sources implementation, in which the MPI source program is compiled and then linked with the MPI libraries to obtain the executable. The user issues a directive to the operating system that places a copy of the executable program on each processor, the number of processes is provided within the user directive. This implementation works well on a wide range of hardware platforms and also supports using of C/C++ and FORTRAN programming languages.

In this paper, we summarize the risks that are hidden beyond using MPI in writing parallel applications. The paper is organized as follows: section 2 includes the related work. Section 3 covers the MPI risks related to analysis. In section 4, we present the risks of MPI anomalies.

2. RELATED WORK

Several studies have been introduced concerning using MPI as a parallel programming paradigm. Petrini et al. [15] introduced a model based system to predict the performance of programs on machines prior to their construction, and to identify the causes of performance variations from the predictions. These methods pick up the slight variations in a program execution that arise at runtime that cannot be modeled by examining the static code.

Vampire [19] and Diem as [29] are two trace based analysis tools that predict parallel programs performance. These models use a trace file and the user’s selection of network parameters that is used in the communication model to
simulate the program execution.

MPE (Multi-Processing Environment) library and jump shot [1] that are distributed with MPICH implementation provide graphical performance analysis for message passing interface programs.

Some other studies [8], [10], [12], [18], [20], [37] handled the issues of data races, control and data flow for parallel programs especially that are based on MPI paradigm. The main challenge in control and data flow of such program is how to represent the program control flow graph, since the nature of the program under consideration is not as that of the ordinary serial programs.

Program correctness is one of the most difficult challenges in parallel programming. Model-based testing is a software testing approach in which test cases are derived from a model that describes the system under test. Practically, model-based test works only for small base blocks of an application. In most cases it is very difficult to automatically build a model on the basis of the code; also the manual creation of models is a hard and error prone process. This approach suffers from the problem of quick extension of state space that is can be partially controlled by using reduction methods. For MPI programs, this approach would require that programmers build, either manually or automatically, a model of their applications in a language such as MPI-SPIN [31],[38], or Zing [30],[34].

3. ANALYSIS-BASED RISKS

We categorize risks that are hidden beyond using MPI in writing parallel applications into two main categories as shown in figure 1. The first one concerns with program analysis including speed up, control and data flow. The second category concerns with the anomalies that may arise in MPI programs including non-determinacy and deadlocks.

![Figure 1. MPI Risks](image)

3.1 Speed up Challenges

Running parallel applications requires special and expensive processing resources to obtain the required results within a reasonable time. Before parallelizing serial applications, some analysis is recommended to be carried out to decide whether it will benefit from parallelization or not. In this section we propose an experimental method to predict the speed up of MPI applications.

Execution time reduction is one of the most challenging goals of parallel programming. Theoretically, adding extra processors to a processing system leads to a smaller execution time of a program compared with its execution time using a fewer processors system or a single machine[17]. Practically, when a program is executed in parallel, the hypothesis that the parallel program will run faster is not always satisfied. If the main goal of parallelizing a serial program is to obtain a faster run then the main criterion to be considered is the speedup gained from parallelization.

Speed up is defined as the ratio of serial execution time to the parallel execution time [6], it is used to express how many times a parallel program works faster than its serial version used to solve the same problem. Many conflicting parameters such as parallel overhead / computation ratio [2], hardware architecture, programming paradigm, programming style may negatively affect the execution time of a parallel program making its execution time larger than that of the serial version and thus any parallelization gain will be lost. In order to obtain a faster parallel program, these conflicted parameters need to be well optimized.

3.2 Experimental Speed up Prediction

In some cases, the predicted performance may differ from that achieved experimentally. In this section we present an experimental method to predict the speed up of MPI applications as a performance measure. The proposed method is summarized in the following steps:

1. Execute the serial version of MPI application on a single processor machine.
2. Record the serial execution time, \( T_s \).
3. Execute the parallel MPI application on the same single processor machine repeatedly using arbitrary number of MPI processes, 1,2,3,...,n.
4. Record the parallel execution times, \( T_{p1},T_{p2},...,T_{pn} \), for each run.
5. Graph the obtained results as a two dimensional graph. The X-axis for MPI processes number and the Y-axis for the parallel execution times, \( T_{p1},T_{p2},...,T_{pn} \).
6. If the parallel execution time is rapidly increases as the number of MPI processes increases, this implies that the MPI application will exhibit a poor speed up if it is run in parallel on multiple physical processors.
7. If the parallel execution time remains constant or slowly increases as the number of MPI processes increases, this implies that the MPI application will exhibit a linear speed up if it is run in parallel on multiple physical processors.

We applied the proposed method on two MPI applications. The first one solves the concurrent wave equation and the second finds the number of primes and also the largest prime
number within an interval of integers. The two applications are also executed in parallel on multiple physical processors. The recorded serial execution time, $T_s$, for both applications is used to find out their experimental speed up to be compared with the predicted ones.

Since modern parallel machines are very costly and not easy to be access, the parallel MPI applications that solve both wave equation and prime numbers generator problems [4] were executed on a hardware architecture that consists of 8 DELL machines. Each of these machines consists of Intel i386 based P4-1.6GHz processor with 512MB memory running on Microsoft Windows XP Professional Service Pack 2. These machines are connected via a Fast Ethernet 100Mbps switch. These machines are not as powerful as the recent cluster machines in terms of the hardware and performance but they can reasonably perform for testing purposes and also for solving small and middle size parallel problems. The experiments programs were written in Fortran 90 using MPICH2 version 1.0.6p1, as a message passing implementation. Serial execution time, parallel execution time on a single processor using multiple processes and also parallel execution time on multiple processors for both problems are shown in table 1.

Applying the proposed speed up prediction method to wave equation problem using 10 MPI processes on a single physical processor we predicted that the application will exhibit a poor speed up if it is executed in parallel using multiple physical processors.

Table 1. Serial and parallel execution times for Wave Equation and Primes Generator

<table>
<thead>
<tr>
<th>Problem</th>
<th>Serial execution time</th>
<th>Parallel execution</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Single physical</td>
<td>Multiple physical processors</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MPI processes</td>
<td>processors</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Execution time</td>
<td>Execution time</td>
</tr>
<tr>
<td>Problem 1</td>
<td></td>
<td></td>
<td>1</td>
<td>1.3561</td>
</tr>
<tr>
<td>Wave Equation</td>
<td>0.80216</td>
<td></td>
<td>2</td>
<td>3.6942</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>6.3833</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>9.4002</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>12.5629</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>15.301</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>18.1778</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>21.5001</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td>24.1733</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td>27.3349</td>
</tr>
<tr>
<td>Problem 2</td>
<td>55.625</td>
<td></td>
<td>2</td>
<td>55.5887</td>
</tr>
<tr>
<td>Primes Generator</td>
<td></td>
<td></td>
<td>4</td>
<td>55.464</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>54.9653</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td>55.5158</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td>55.1428</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>20</td>
<td>55.9213</td>
</tr>
</tbody>
</table>

Our prediction is based on that the execution time is rapidly increases as the number of MPI processes as shown in figure 2. To prove that our prediction was true, we executed the same MPI code on 8 physical processors. Knowing the execution time of the serial code version, the experimental speed up was calculated.

Figure 3 shows that the maximum speed up achieved by 8 physical processors was only 0.66228534 and hence our prediction was true.

To be unbiased, we also re-executed the same parallel code using different number of processes on the same 8 physical processors. Figure 4 shows that the execution time was negatively affected as the number of MPI processes increases except in case of running a small number of MPI processes using 8 physical processors.
The experimental results show that there is no significant speed up improvement as shown in figure 5. This also proves that our prediction was true. Applying the proposed method to prime numbers generator problem using 20 MPI processes on a single physical processor, we predicted that the application will exhibit a linear speed up if it is executed in parallel using multiple physical processors. Our prediction is based on that the execution time is slowly increases or seems to be constant as the number of MPI processes as shown in figure 6. Running the same MPI code on 8 physical processors achieved a linear speed up as shown figure 7 and hence our prediction was also true.

3.3 Control and Data Flow Challenges

New trends towards multiple core processors imply using standard programming models to develop efficient, reliable and portable programs for distributed memory multiprocessors and workstation PC clusters. Message passing using MPI is widely used to write efficient, reliable and portable applications. Control and data flow analysis concepts, techniques and tools are needed to understand and analyze MPI programs. If our point of interest is the program control and data flow analysis, to decide to parallelize or not to parallelize our applications, there is a question to be answered, "Can the existing concepts, techniques and tools used to analyze sequential programs also be used to analyze parallel ones written in MPI?". In this section we'll answer this question. MPI programs are coded in a special manner, in which each process executes the same program with unique data. All parallelism is explicit; the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs. Figure 8 shows a pseudo code of an MPI-based program.

1. sum=3
2. Initialize MPI environment.
3. Determine the number of MPI Processes and identities.
4. if myid=0 then
5. Receive "sender_id" from any process
6. sum = sum + received
7. x0 = sum
8. endif
9. if myid=1 then
10. x = 5
11. if x<0 then
12. x = x +1
13. else
14. x = x-1
15. endif
16. Send "x" to process 0
17. process_id = myid
18. Send "process_id" to process 0
19. endif
20. if myid = 2 then
21. x = 7
22. x = x * 2
23. Send "x" to process 0
24. process_id = myid
25. Send "process_id" to process 0
26. endif
27. Finalize MPI environment.
28. END

Figure 2. Execution time using 10 processes on a single CPU for problem 1
Figure 3. Experimental speed up for problem 1
Figure 4. Effect of processes number on execution time using 8 CPUs for problem1
Figure 5. Experimental vs. ideal speed up for problem 1
Figure 6. Execution time using 20 processes on a single CPU for problem 2
Figure 7. Experimental speed up for problem2
Figure 8. pseudo MPI –based code
Running the executable of the listed code several times using three process may yields one of two outputs, one of them indicates that the value 4 is sent from process 1 to process 0 and the sum value is 7, the other one indicates that the value 14 is sent from process 2 to process 0 and the sum value is 17. The order of these outputs is unpredictable.

This situation reflects the non-deterministic behavior of program execution. These results demonstrate that the affected statements are not the only affected ones but also there are some other statements that should be affected as shown in table 2 and table 3. This indicates that the ordinary data flow analysis fails to demonstrate the nature of the source code.

<table>
<thead>
<tr>
<th>Case</th>
<th>Affected statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>definition of sum in line 1</td>
<td>6 sum=sum + received</td>
</tr>
<tr>
<td>definition of sum in line 6</td>
<td>7 x0 = sum</td>
</tr>
<tr>
<td>definition of x in line 10</td>
<td>11 if x&lt;0 then 12 x = x +1 14 x = x – 1</td>
</tr>
<tr>
<td>definition of x in line 12</td>
<td>16 Send &quot;x&quot; to process 0</td>
</tr>
<tr>
<td>definition of x in line 14</td>
<td>16 Send &quot;x&quot; to process 0</td>
</tr>
<tr>
<td>definition of x in line 21</td>
<td>22 x = x * 2</td>
</tr>
<tr>
<td>definition of x in line 22</td>
<td>25 Send &quot;x&quot; to process 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case</th>
<th>Statements should be</th>
</tr>
</thead>
<tbody>
<tr>
<td>definition of x in lines 12, 14</td>
<td>4 Receive &quot;received&quot;</td>
</tr>
<tr>
<td>definition of x in line 22</td>
<td></td>
</tr>
</tbody>
</table>

3.4 Implementation of MPI-CFG Construction

Now we present our technique to build the MPI-CFG to cover the statements that not covered by using ordinary data flow analysis. This flow graph resembles program execution graph [10], parallel flow graph [16], and parallel program flow graph PPFG [9]. The technique is summarized in the following steps:

1. MPI-based program statements identification.

In this phase, each program statement is assigned a unique number "type" to be identified from the other statements of the program.

2. Building Basic Blocks

This phase uses the output of the previous phase to build the program basic blocks. We construct two extra special types of basic blocks called "message block" and "finalize block".


This phase connects the basic blocks generated in the previous phase with the appropriate edges. The edges are classified into three categories, sequential, parallel, and synchronization edges. The complete algorithm is described in [3].

Figure 9 shows the MPI-CFG of the real code corresponds to the pseudo code listed in Figure 8.

Program correctness is one of the most difficult challenges in parallel programming. Message Passing Interface MPI is widely used in writing parallel applications. Since MPI is not a compiled language, the programmer will be encharged with several programming bugs. This paper presents the most common programming bugs arise in MPI programs to help the programmer to compromise between the advantage of parallelism and the extra effort needed to detect and fix such bugs. Debugging parallel programs is classified as a difficult problem. Most of parallel Application programmers focus only on the constructive part of creating a parallel algorithm for a particular problem and how to implement it, but ignore the issues of debugging [7]. Parallel programming adds a new category of bugs caused by the interaction of multiple communicated parallel tasks. These parallel bugs ‘Heisen bugs’ [14], [22] are difficult to be detected and resolved due to the nondeterministic nature of parallel tasks running which makes the bugs may disappear when one attempt to detect them. Non-determinacy and deadlock are two types of bugs that can’t be detected during linking and compilation phases. In many cases, communications among running processes lead to either a non-determinate results or deadlock.

In [25], [5] an algebraic specification of an MPI-like programming language, called Simple MPI (SMPI), which supports message passing between several processes with functions "send" and "receive". The specification of SMPI is described in the algebraic specification language Maude.
[26],[33], which is a member of the OBJ family, and is a successor of the algebraic specification language OBJ3 [21].

As related studies, Salman et al have shown how formal verification based on model checking can be used to find actual deadlocks in algorithms that use the MPI one-sided communication primitives [32]. Nakamura et al. have proposed a behavioral specification of imperative programs in Cafe OBJ [24]. Café OBJ [11] is another successor of OBJ3. In this section, we give a rewrite specification of parallel imperative programs in Maude. One of the strong points of rewrite specifications is that it provides automatic exhaustive searching for verifying a given specification.

4.1 Non-determinacy Challenges

Non-determinate results arise according to the nondeterministic data arrival order to the process that computes the results and hence the output results may vary for multiple runs of the same program with the same input data. We used Maud system to detect the non-determinacy of a given SMPI code. Consider the SMPI program example list in fig 10. In this example, The process 0 repeatedly receives the messages from all other processes whose IDs are np-1, np-2, …, 2, 1. Therefore, even if a process P sends a message before another process Q whose ID is larger than P’s ID, the process P should wait until the process 0 receives all messages sent from the processes whose IDs are larger than P’s ID.

1. if(not(pid = 0)){
2.     send(pid,0);
3. }
4. if(pid = 0){
5.     int x ; int y ; int i ;
6.     y := 1 ; i := np ;
7.     while (i > 1) {
8.         i := i - 1 ;
9.         recv(x,i) ;
10.        y:= x - y;
11.     }
12. }

Figure 10: SMPI program
P1

result List:

| (pid :: 1) np :: 5 | (pid :: 2) np :: 5 |
| (pid :: 3) np :: 5 | (pid :: 4) np :: 5 |
| (i :: 1) (x :: 4) (y :: 3) (pid :: 0) np :: 5 |

Figure 11: Parallel execution of P1

Fig. 11 shows the Maude result indicating that the value of the variable y is 3 = 4 – (3 – (2 – (1 -1))). We check whether in all normal forms the value is also 3 or not by the Maude search command as shown in Fig. 12.

Ten solutions are returned. In Solution 1, the value of the variable y is 1. In Solution 2, it is 5, and in Solution 10, its value is –3.

The obtained solutions reflect the nondeterministic feature of MPI programs. The reason of non-determinacy in this case is that using of recv (x, any) implies that the computation involved will be affected by the nondeterministic arrival order of x, and hence the final assigned value may varies for each run.

This situation did not appear in case of using the expression “Y = Y *X” instead of the statement 10, “Y = X - Y” because multiplication is commutative and will not be affected by the arrival order of X, in contrast to the case of using the expression “Y = X-Y”, in which subtraction operation will be affected yielding these nondeterministic results. Thus, we conclude that this program is non-determinate.

Maude> search mpirun(5,…) =>! ((pid :: 0) (y :: Y:Int) S:Store |
L:List) such that (Y:Int /= 3).

Solution 1  (state 67255) …
L:List --> (pid :: 1) np :: 5 | (pid :: 2) np :: 5
    | (pid :: 3) np :: 5 | (pid :: 4) np :: 5
S:Store --> (i :: 1) (x :: 3) np :: 5
Y:Int --> 1

Solution 2  (state 67266) …
L:List --> (pid :: 1) np :: 5 | (pid :: 2) np :: 5
    | (pid :: 3) np :: 5 | (pid :: 4) np :: 5
S:Store --> (i :: 1) (x :: 4) np :: 5
Y:Int --> 5

Solution 10  (state 67265) …
L:List --> (pid :: 1) np :: 5 | (pid :: 2) np :: 5
    | (pid :: 3) np :: 5 | (pid :: 4) np :: 5
S:Store --> (i :: 1) (x :: 1) np :: 5
Y:Int --> -3

No more solutions.

states: 67266  rewrites: 2176921 in 5349ms cpu (5467ms real)
(406918 rewrites/second)

Figure 12: Verification of an SMPI program non-determinacy

4.2 Deadlock Challenges

Deadlocks errors arise if the dependencies between parallel tasks described by an MPI construct can’t be satisfied. Our specification can also be used to detect other kinds of bugs.
Maude> rewrite mpirun(5,
if(not(pid = 0)){
    int x ;
    send(pid,0);
    recv(x,0);
}
if(pid = 0){
    int x ; int i ;
    i := 1 ;
    while (np > i) {
        recv(x,any) ;
        send(x,i) ;
        i := i + 1 ;
    }
}
}.

rewrites: 737 in 0ms cpu (3ms real) (930555 rewrites/second)
result List:

(x :: 1) (pid :: 1) np :: 5 | (x :: 2) (pid :: 2) np :: 5
|x :: 3) (pid :: 3) np :: 5 | (x :: 4) (pid :: 4) np :: 5
| (i :: 5) (x :: 4) (pid :: 0) np :: 5

Figure 13: Parallel execution of P2

One of the typical bugs in message-passing parallel programs is a deadlock. Consider the program example, P2, listed in fig 13. In this program, each processes except 0 tries to send its ID to the process 0, and then tries to receive a message from the process 0 and assign the message to the variable x.

The process 0 tries to receive a message from any source and send it to each process in the ascending order. In the store of each process in the above result, we can see that its ID is assigned to x. However, as we mentioned, it does not guarantee that all possible parallel execution work well like that.

Now, we check whether the process 0 finishes the program in all possible parallel execution or not. In Fig. 14, the pattern of the search command means that the process 0 stops running with some part of the program P1 P2 remaining.

The Maude system returns the six normal forms (solutions) which are matched with the pattern. In Solution 6, the process 0 stops with P1:Pgm --> send(x,i); as the head of the remaining program and the store S:Store --> (i :: 3)(x :: 4)(np :: 5), which tells us that the process 0 stops when trying to send a message to process 3. We can see that the current state of the process 3 in the list of states (L) which contains (pid :: 3), and that the process 3 also stops with send(pid,0);. Both the process 0 and 3 try to send a message to each other, and fails into a deadlock.

Since we treat SMPI programs themselves as terms to be rewritten, the search result is easy to be read. We can directly see the point of the problem in the program. The readability of not only specifications but also the results of their executions and the traces of their verifications is one of the most important features of algebraic specification languages.

Maude> search mpirun(5,...)

... Solution 1 (state 11078)

Solution 6 (state 16639)
states: 16860 rewrites: 664229 in 1504ms cpu (1701ms real)
(441482 rewrites/second)
L:List -->
(x :: 1) (pid :: 1) np :: 5
| (x :: 2) (pid :: 2) np :: 5
| ((x :: na) (pid :: 3) np :: 5) send(pid,0); recv(x,0);
    if pid = 0{int x ; int i ; i := 1 ;
        while np > i{recv(x,any); send(x,i); i := i + 1 ;} end
    } while np > i{recv(x,any); send(x,i); i := i + 1 ;}

S:Store --> (i :: 3) (x :: 4) np :: 5
P1:Pgm --> send(x,i);
P2:Pgm --> i := i + 1 ; while np > i{recv(x,any); send(x,i);
    i := i + 1 ;} end

No more solutions.
states: 19232 rewrites: 729193 in 1679ms cpu (1878ms real)
(434063 rewrites/second)

Figure 14: Verification of deadlock

5. CONCLUSION AND FUTURE WORK

In this paper we studied the risks hidden behind the using of MPI paradigm. We categorized these risks into two main categories. The first one concerns with program analysis including speed up, control and data flow. The second category concerns with the anomalies that may arise in MPI programs including non-determinacy and deadlocks.

Concerning speed up, an experimental method for prediction parallel speed up was presented. Control and data flow for MPI has been studied and a technique for analyzing MPI programs has also been introduced. Non-determinacy and deadlocks bugs arise in MPI have been discussed. An algebraic specification language SMPI for presenting MPI programs has also been introduced. Non-determinacy and
deadlocks bugs have been verified using Maud system. In future we hope to create an integrated system to handle all MPI communication and synchronization primitives.

REFERENCES


[33] The Maude system: http://maude.cs.uiuc.edu/


