Verification of MPI Java Programs using Software Model Checking

Waqas Ur Rehman, Muhammad Sohaib Ayub, Junaid Haroon Siddiqui

Department of Computer Science, LUMS School of Science and Engineering, Lahore, Pakistan

{waqas.rehman, 15030039, junaid.siddiqui}@lums.edu.pk

Abstract

Development of concurrent software requires the programmer to be aware of non-determinism, data races, and deadlocks. MPI (message passing interface) is a popular standard for writing message oriented distributed applications. Some messages in MPI systems can be processed by one of the many machines and in many possible orders. This non-determinism can affect the result of an MPI application. The alternate results may or may not be correct. To verify MPI applications, we need to check all these possible orderings and use an application specific oracle to decide if these orderings give correct output. MPJ Express is an open source Java implementation of the MPI standard. We developed a Java based model of MPJ Express, where processes are modeled as threads, and which can run unmodified MPI Java programs on a single system. This enabled us to adapt the Java PathFinder explicit state software model checker (JPF) using a custom listener to verify our model running real MPI Java programs. We evaluated our approach using small examples where model checking revealed message orders that would result in incorrect system behavior.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Model Checking; D.1.3 [*Concurrent Programming*]: Distributed Programming

Keywords Message Passing Interface in Java (MPJ), Model Checking, Java PathFinder(JPF)

1. Introduction

Model checking (Clarke et al. 1999) is a powerful program analysis technique based on systematic exploration of nondeterministic choices in a program. Nondeterministic choices could be program inputs, modeled behavior of an external system, thread interleavings in a multithreaded program, or even message orders in a distributed system. Earlier software model checkers required converting the program into a a modeling language which was then verified for certain properties (Clarke et al. 1999). Recent software model checkers, such as the Java PathFinder (JPF) (Havelund and Pressburger 2000), now provide the foundation of an increasingly effective tool-set for systematic checking of programs written in commonly used languages.

Despite the progress, adapting the model checking techniques for new domains remains a challenging problem in realizing its true potential in increasing our ability to deploy more reliable software systems. One such domain is MPI programs. Message Passing Interface (MPI) is a framework for the development of parallel and distributed applications. It has been implemented in C, FORTRAN, and Java. The Java implementation is called MPJ Express[1]. It is deployed on different machines communicating through an underlying communication channel. MPI programs often form the basis of large distributed systems and their correctness is extremely critical. However, testing of MPI programs is difficult because messages can arrive in many possible orders and it is difficult to find if any of these message orders does not give the desired outcome.

This paper presents a novel technique to adapt model checking for unchanged MPI programs written in MPI Java. While prior work addressed this problem requiring conversion of the MPI program into modeling languages (van Galen 2011) or provided general guidelines of how such a solution can be built (Gopalakrishnan et al. 2011), we provide a concrete end to end implementation to model check unchanged MPI Java Programs. Our choice of using MPI Java enables us to adapt the existing Java PathFinder model checker but the fundamental techniques are not specific to Java.

Our key insight is that an MPI Java proxy model that converts the new source of non-determinism, i.e. message ordering, into events that the existing JPF model checker can understand enables model checking of unchanged MPI Java programs and also enables effective partial order reduction (POR) which is built in Java PathFinder. POR is the mechanism model checking uses to reduce the number of choices explored on the basis of equivalence sets. To implement this, we introduced a custom listener for message related events, a custom choice point to explore alternate message orderings, and a custom MPJ Express proxy that translated calls for the unchanged program.

2. Approach

MPJ Express framework is used to run MPI Java Programs in cluster as well as multicore environment. Each MPJ process is identified by a unique rank number, assigned by framework. MPJ Express built-in functions Rank(), Size(), Recv() and Send() are used to get rank of specific process, total number of processes in communication, receiving a message and sending a message respectively.

We are using Java PathFinder(JPF) as model checker. JPF has limitation that it cannot model check programs executing on different machines. To cater for this issue, we have developed a model of MPJ Express which uses Java reflection for running unchanged MPJ Program. For each MPJ process, we have created a thread and

---

[1] http://mpj-express.org/

Table 1: Roles of threads

Thread	Roles	Level
Rank 0	Initiator, Terminator	-
Rank 1	Ordinary, Intermediary	1
Rank 2	Ordinary	-
Rank 3	Ordinary, Intermediary	0
Rank 4	Ordinary	-

Table 2: Work distribution by Initiator thread (Rank 0)

Ordinary Worker	Work	Average
Rank 1	[1,2,3,4]	2.5
Rank 2	[5,6,7,8]	6.5
Rank 3	[9,10,11,12]	10.5
Rank 4	[13,14,15,16]	14.5

Table 3: Order of message reception at Rank 1 thread

Sr. No.	Local Average	Message Ordering	Average
1.	R1[2.5]	R2[6.5], R3[12.5]	8.5
2.	R1[2.5]	R3[12.5], R2[6.5]	7

assigned a unique rank number to it, starting from 0. Each MPJ process, modeled as thread, can access its rank number in the same manner as in real environment by calling Rank(). We have also provided the support of Size() function which returns total number of threads executing. To model communication channel, we are using a shared synchronized queue ensuring prevention of deadlock or race condition. Threads enqueue and dequeue messages in the queue to exhibit Send() and Recv() behaviors. Our MPJ Express model supports all data types which are supported by MPJ Express.

In JPF, we have disabled all choice generators except Root and Terminate choice generators. Terminate choice generator is responsible for creating thread interleavings. Additional choice generators, of ThreadChoiceFromSet type, are created when needed. JPF is extended with a listener and a set of classes (for logging) to perform model checking of MPJ programs. JPF starts execution with Root choice generator with only one thread. As a result of termination of main thread, a terminate choice generator is created with all runnable threads as choices. JPF starts execution with one of the thread choice while listener observing the execution. Threads in our model are either in blocked or unblocked state. Blocked threads are those which are waiting for the message(s) in queue. Unblocked threads are those which either never executed since program started or unblocked after the creation a choice generator to executed all the blocked threads. Partial order reduction is already integrated in JPF and helping to prune state space reasonably in our model checking process.

3. Evaluation

In this section, we are going to present an example of calculating running average of first 16 integers. In this example, five threads (modeled MPJ Express processes) are used with atleast one role. The role might be Initiator, Intermediary, Ordinary or Terminator. Initiator is responsible for distributing the work among other threads. Ordinary workers are responsible for processing only. Intermediary, as name suggests, is responsible for collecting results from ordinary threads, consolidate them and forward results to either higher level intermediaries or terminator. Terminator collects the final result.
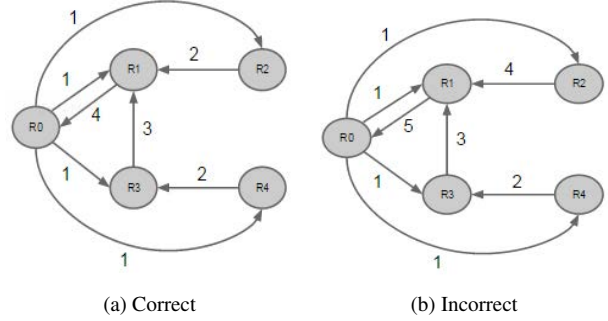

(a) Correct (b) Incorrect

Figure 1: Message Interleaving at Rank 1

As discussed above, our MPJ Express model assigns each thread a unique rank number starting from 0. Rank 0 thread is playing the role of Initiator and Terminator so it is responsible for distribution of work and collection of final result. Rank 1 thread acting as ordinary worker as well as intermediary of level 1. It will perform calculations for the work assigned and collect results from other lower level intermediaries (Rank 3) and ordinary workers (Rank 2), and forward consolidated result to terminator thread. Rank 3 thread is also an intermediary with level 0 and responsible for collecting result from Rank 4 thread. Thread with lower the level is less responsible for collecting results from other intermediaries/ordinary workers. Rank 2 and Rank 4 processes are just ordinary workers. They calculate the average of the numbers assigned to them and forward result to Rank 1 and Rank 3 workers respectively. Table 1 summarizes the roles of each thread in our example.

Rank 0 thread distributes work according to Table 2 and waits for the final result from Rank 1 for output. Now, each worker will first calculate the average of task assigned to it and forward result to respective intermediaries. Here it is important to consider the order of messages being received at intermediary nodes. As mentioned earlier, Rank 1 thread is responsible for receiving results from Rank 2 and Rank 3 threads. Now, there are two possibilities that Rank 1 will either receive message from Rank 2 then Rank 3 or vice versa. Both these message orderings are shown in Figure 1a and Figure 1b where labeled arrows show the order of messages sent/received. Rank 2 and Rank 4 will calculate averages of numbers assigned to them and forward messages to Rank 1 and Rank 3 respectively. Rank 3 receives result from Rank 4 and consolidate it with locally calculated average and forward result to Rank 1. Rank 1 collects results from Rank 2 and Rank 3 processes and calculates the final result in order to forward it to Rank 0. Results generated from both orderings at Rank 1 will produce different final averages as shown in Table 3. These orderings are due to the unpredictable behavior of network. Message ordering in Figure 1a will produce average 8.5 which is correct whereas message orderings in Figure 1b will produce 7 which is wrong result .

References

E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT press, 1999.

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. *Communications of the ACM*, 54(12): 82–91, 2011.

K. Havelund and T. Pressburger. Model Checking Java Programs Using Java Pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

R. van Galen. Towards Verification of MPJ-based Java Programs. *15th Twente Student Conference on IT*, 15, 2011.