Practical Model Checking Method for Verifying Correctness of MPI Programs

Salman Pervez[1], Ganesh Gopalakrishnan[1], Robert M. Kirby[1], Robert Palmer[1], Rajeev Thakur[2], and William Gropp[2]

[1] School of Computing
University of Utah
Salt Lake City, UT 84112, USA

[2] Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA

Abstract. Formal verification of programs often requires creating a model of the program and running it through a model-checking tool. However, this model-creation step is itself error prone, tedious, and difficult to do for someone not familiar with formal verification. In this paper, we describe a tool for verifying correctness of MPI programs that does not require the creation of a model and instead works directly on the MPI program. Such a tool is useful in the hands of average MPI programmers. Our tool uses the MPI profiling interface, PMPI, to trap MPI calls and hand over control of the MPI function execution to a scheduler. The scheduler verifies correctness of the program by executing all "relevant" interleavings of the program. The scheduler records an initial trace and replays its interleaving variants by using dynamic partial-order reduction. We describe the design and implementation of the tool and compare it with our previous work based on model checking.

1 Introduction

Parallel programs are notoriously difficult to debug, and MPI programs, particularly those that have intricate control flow or employ relatively new features such as one-sided communication, are no exception. Tools such as MARMOT [8], MPI-check [9], Umpire [16], and Intel Message Checker [3] are capable of detecting many errors in MPI programs. However, these tools do not provide guarantees to the effect that all interleavings of the processes in the program being tested have been systematically examined. While there is an exponential number of such interleavings, the body of knowledge discovered under the heading 'partial-order reduction' [2, Chapter 10]—a class of methods belonging to the area known as model checking [2]—offers a number of specific approaches to examine only *some* (usually a small fraction) of these interleavings, and declare that the effect of examining all the interleavings has been achieved. Partial-order methods are commonly used to verify *models* of parallel programs. For MPI programs, this would mean that programmers must build, either manually or automatically, a *model* (description) of their protocol in a language such as

Promela [7], MPI-SPIN [13], or Zing [10]. This model-creation step is known to be tedious and error prone.

We take the *in-situ* approach to model checking, previously demonstrated in the context of many languages: C programs in tools such as [6, 18], and Java programs in tools such as [17], to name two. During in-situ model checking, programs written in the target language (usually with some obvious simplifications such as data-range reduction) are directly model checked, without first creating a model. In this paper, we describe our tool that performs in-situ model checking of MPI programs that use one-sided communication, employing a *dynamic* version of partial-order reduction (DPOR, [4]) to reduce the number of interleavings, thus being able to, in effect, *exhaustively* examine all traces of small (but intricate) MPI programs. We call our tool *in situ dynamic partial order*, or ISP. ISP handles many standard MPI communication functions including `MPI_Barrier`, various flavors of `MPI_Send` and `MPI_Recv`, and some MPI one-sided functions. In this paper we detail how it handles one-sided functions, partly because of our case study involving one-sided communication, and partly because of the inherent intricacies of handling one-sided communication under in-situ scheduling. The complex nature of MPI one-sided communication also forces us to use information specific to the underlying library, MPICH2 in this case. One restriction placed by MPICH2 is that for passive-target one-sided communication, the target process needs to be inside the MPI progress engine in order to process lock requests. We account for this restriction in ISP as described in Section 2. ISP can easily be extended to efficiently handle other MPI implementations.

```
0: MPI_Init
1: MPI_Win_lock
2: MPI_Accumulate
3: MPI_Win_unlock
4: MPI_Barrier
5: MPI_Finalize
```

Fig. 1. Example 1

To motivate the ISP approach, consider the simple MPI program given in Figure 1, executed by two processes P0 and P1. Figure 2 shows how the ISP tool examines two different interleavings of this program. ISP employs the MPI profiling interface, PMPI, to trap MPI calls and hand over control of the MPI function execution to a scheduling process. This *scheduler* can dictate the order in which each process makes MPI calls. We define the block of code starting from the beginning of an MPI call, going forward in the code path including C program statements, and ending at the beginning of the next MPI call to be a *transition* (in our current example, there are no intervening C program statements). We assume that no transition executes infinitely (MPI calls always complete, and the intervening C statements have no infinite loops). We also assume that the given MPI program is well formed in accordance with the MPI Standard 2.0. The errors detected by ISP are safety properties [2], including: (i) deadlocks, (ii) violations of `assert` statements placed by the user, and (iii) exceptions thrown by the runtime.

Given these assumptions, at Step 1 ISP would find processes P0 and P1 to be runnable (`Options`). Assume that ISP randomly chooses `P1`, executing the instruction shown against `P1.1`, which is an `MPI_Win_lock`. At Step 2, P0 and P1 are both runnable again; ISP picks P1, executing `MPI_Accumulate`. Proceeding in this manner, we reach Step 4 where P1 executes `MPI_Barrier`. At this point,

Step No.	Proc. Options	First Inter-leaving	Trace due to First Interleaving	Second Inter-leaving	Trace due to Second Interleaving
1:	P0 P1	P1	P1.1: MPI_Win_lock	P1	P1.1: MPI_Win_lock
2:	P0 P1	P1	P1.2: MPI_Accumulate	P1	P1.2: MPI_Accumulate
3:	P0 P1	P1	P1.3: MPI_Win_unlock	P1	P1.3: MPI_Win_unlock
4:	P0 P1	P1	P1.4: MPI_Barrier	P1	P1.4: MPI_Barrier
5:	P0	P0	P0.1: MPI_Win_lock	P0	P0.1: MPI_Win_lock
6:	P0	P0	P0.2: MPI_Accumulate	P0	P0.2: MPI_Accumulate
7:	P0	P0	P0.3: MPI_Win_unlock	P0	P0.3: MPI_Win_unlock
8:	P0	P0	P0.4: MPI_Barrier	P0	P0.4: MPI_Barrier
9:	P0 P1	P1	P1.5: MPI_Finalize	P0	P0.5: MPI_Finalize
10:	P0	P0	P0.5: MPI_Finalize	P1	P1.5: MPI_Finalize

Fig. 2. Interleavings explored on Example 1

the only runnable process would be P0, forcing Steps 5 through 8. The execution of `MPI_Barrier` by P0 results in both processes becoming runnable once again. Now ISP picks P1, followed by P0, generating the first interleaving (the last two actions being `MPI_Finalize`).

A naïve implementation of ISP would now backtrack to the *decision point* at Step 9, picking P0 instead of P1, as shown by the second interleaving. We say this is 'naïve' because we know that the order in which `MPI_Finalize` is invoked is immaterial. This is precisely what partial-order reduction does: it computes which actions are *commuting actions*, meaning that their interleavings do not produce any semantically observable changes in program outcome. Another commuting pair in the above program would be the two `MPI_Barrier` invocations. Under a naïve approach, an N-way barrier can generate all $N!$ interleavings of the order in which processes encounter the barriers; under partial-order reduction, we can simply generate one interleaving and claim complete coverage.

If our current example is run with MPICH2 with the `MPI_Win_lock` operation specifying an exclusive lock, the only actions that require interleaving are the `MPI_Win_unlock` calls within which shared variable updates take place. For a process accessing the MPI window remotely, only its `MPI_Win_unlock` call modifies the communication window, posting all the accumulated updates within that particular epoch. For this example, the ISP tool will generate two interleavings as opposed to 504 interleavings[3] if we were to use only the in-situ feature without DPOR. Since the theory of partial-order reduction is vast, we simply present our assumptions as a table of commuting MPI operations (Figure 6), citing past references [11] based on which such tables can be created. Such a table can be adjusted to correspond to any MPI implementation of choice, *or even deliberately adjusted to suppress certain interleavings for quicker bug hunting.* Also, as opposed to *static* partial-order reduction where the commuting nature of the two `MPI_Finalize` invocations would have been determined while going forward

[3] $2 \times (10!/(5!)^2)$

during the first interleaving, we instead follow the *dynamic* approach to partial-order reduction, in which we (i) fully generate the first interleaving, and (ii) walk up the stack trace and mark places where interleavings can be *added*. Space does not permit a fuller description of DPOR; suffices to note that it exploits run-time information to effect better reduction (*e.g.*, wildcard communication, as described in [11]).

2 Basics of Scheduling and In-Situ Model Checking

In-situ model checking works by letting a *scheduler* control the transitions of the given MPI program. The scheduler opens an array of communication channels (realized through TCP sockets) through which it receives *appeals* from each process. The pseudocode in Figure 3 captures how the MPI call of a process (generically called `Generic_Func`) is processed. Basically, the MPI function call is intercepted by the profiling library. It then conveys the process id (`pID`), the call type (`Generic_Func`), and the remaining arguments to the scheduler through the `sendToSocket` call. By way of reply, the scheduler either provides a 'go-ahead' or a 'loop' to the appealing processes. A 'loop' signal indicates that the appealing process must make an `MPI_Iprobe` call. The `MPI_Iprobe` call is just one side-effect free mechanism that causes control to enter the MPI progress engine to process all queued-up events within it, ensuring forward progress. `MPI_Iprobe` is needed with MPICH2 in order to cause progress to occur on communication with other processes, because MPICH2 does not use an asynchronous progress thread in its progress engine.[4]

```
MPI_Generic_Func(arg1, arg2...argN) {
sendToSocket(pID, Generic_Func, arg1,...,argN);
while(recvFromSocket(pID) != go-ahead)
 MPI_Iprobe(MPI_ANY_SOURCE, 0, MPI_COMM_WORLD...);
return PMPI_Generic_Func(arg1, arg2...argN); }
```

Fig. 3. PMPI instrumentation pseudocode

When the appealing process receives 'go-ahead,' it issues `PMPI_Generic_Func`, which then enters the MPI library. We illustrate further details in the following sections.

2.1 Scheduling Algorithms and Forward Progress

In-situ model checking depends on the designer's understanding of how a given MPI library handles each MPI call in terms of the latitude allowed in the MPI standard. For example, MPICH2 treats a `MPI_Win_lock` operation issued from a remote (non-target) process as a 'no operation.' However, an `MPI_Win_lock` issued by the target process may cause a lock on the one-sided communication window to be acquired. Let us denote the `MPI_Win_lock` issued from a target as `MPI_Win_lock_T`, from a non-target as `MPI_Win_lock_NT`, and also use the altered names `Win_unlock_T` and `Win_unlock_NT` assuming the same conventions. In our framework, these functions are used to indicate when the "trapped control" comes to the `MPI_Generic_Func` associated with these calls.

[4] `MPI_Iprobe` does not have a version corresponding to `MPI_Generic_Func`; otherwise, it would cause an infinite loop when these `MPI_Iprobe`s are trapped.

We also use the notation `PMPI_Win_lock_T` to indicate the `PMPI` call coming after the "trapped" `MPI_Win_lock` call issued by the target, and similarly use the notations `PMPI_Win_lock_NT`, `PMPI_Win_unlock_T`, and `PMPI_Win_unlock_NT`. These `PMPI` calls signal the point in time at which the MPI system first comes to know that these MPI calls are being made. We now present some of the scheduling decisions made by ISP. We explain these with the aid of Figure 1. We rely on the following conventions:

• Because of the assumptions made in Section 1 about *transitions*, we know that each time the scheduler will be handling up to N appeals of the form `sendToSocket(pID, Generic_Func, arg1,...,argN)` (it would be N appeals unless some process has executed a blocking operation).

• The scheduler also keeps track of the lock state of each MPI one-sided window. We will use the terms *window locked* and *window unlocked*.

Consider P0 to be the owner of the window. We call the owner the *target* because that is where all decisions pertaining to locking and unlocking the one-sided MPI window are made. Consider the program in Figure 1 run using processes P0 and P1. Specifically, consider the scheduler actions with respect to the following interleaving:

• P0 does `MPI_Win_lock_T`. The scheduler records that the window is locked, and issues a go-ahead, permitting the `PMPI_Win_lock_T` call to be made.

• P1 does `MPI_Win_lock_NT`. The scheduler treats this as a 'no op' (reasons in Section 1) and gives the go-ahead, allowing P1 to make the `PMPI_Win_lock_NT` call.

• P1 does `MPI_Accumulate`. The scheduler gives the go-ahead, allowing P1 to make the `PMPI_Acccumulate` call.

• P1 does `MPI_Win_unlock_NT`. Noting that the window is locked, the scheduler gives the go-ahead, allowing P1 to make the `PMPI_Win_unlock` call. It records that P1 is blocked.

• P0 does its `MPI_Accumulate`, receiving a go-ahead.

• P0 does its `MPI_Win_unlock_T`. Clearly, the scheduler must issue a go-ahead to P0, causing `PMPI_Win_unlock_T` to occur, thus freeing up the window. Note that P1 has already made its `PMPI_Win_unlock_NT` call. However the following race condition could occur: P0 could hurry through the MPI progress engine upon issuing `PMPI_Win_unlock_T`. Suppose P1's lock request reaches P0 only after P0's `PMPI_Win_unlock_T` call has returned. However, since the MPICH2 progress engine has no separate thread to grant locks, P1's successful acquisition of the window is at the mercy of P0 entering the progress engine again, which happens when P0 executes its `PMPI_Barrier` call.

• For simplicity, our scheduler is implemented in such a way that it moves only one process at a time—in the current example, after we let go P1, we await P1 to make its next MPI command appeal before entertaining any other process.

• However, if we keep P0's appeal in abeyance, the following deadlock could occur: the `PMPI_Win_unlock_NT` can cause an event to be placed in the target's event queue. These events are processed only when the progress engine is entered. However since we have kept P0 in abeyance, the progress engine won't be entered.

```
   S   backtrack First Interleaving     S'    backtrack'   S''  backtrack''
------ -------- -------------------   ------- ---------  ------ ----------
P0  P1    P1    P1.1: MPI_Win_lock    P0  P1     P1      P0  P1    P1
P0  P1    P1    P1.2: MPI_Accumulate  P0  P1     P1      P0  P1    P1
P0  P1    P1    P1.3: MPI_Win_unlock  P0  P1     P0 P1   P0  P1    P0
P0  P1    P1    P1.4: MPI_Barrier     P0  P1     P1
    P1    P0    P0.1: MPI_Win_lock        P1     P0
    P1    P0    P0.2: MPI_Accumulate      P1     P0
    P1    P0    P0.3: MPI_Win_unlock
    P1    P0    P0.4: MPI_Barrier
P0  P1    P1    P1.5: MPI_Finalize
    P1    P0    P0.5: MPI_Finalize
```

Fig. 4. DPOR algorithm applied to Example 1

• Instead of keeping P0 in abeyance, our policy is to keep sending 'loop' to P0, which causes the `IProbe`'s to be issued. This will ensure that P1's event will be processed, causing P1 to reach its next MPI command, at which point we can stop sending 'loop' to P0.

3 In-Situ with Dynamic Partial Order

The algorithm of Figure 5 is how ISP exhaustively explores all *relevant* interleavings of the given MPI process as determined by DPOR. The first interleaving is chosen at random by following a standard depth-first search. Once we have the first random interleaving, it is simply a matter of traversing up the stack, having DPOR identify points where adding interleavings might be useful, and carrying on the search from there.

The data structures used by ISP are the following: **S** contains the set of processes that are able to run at each depth, **backtrack** contains the set of processes that are allowed to run at each depth, **done** contains the set of processes that have been explored at each depth, **servers** is the set of n server connections, one with each MPI process, and **active processes** is initialized to the number of MPI processes n. The most interesting of these is the *backtrack* set. Readers may view it as a set of sets that keeps track of meaningful interleavings at each depth. Note that the sets S and *backtrack* are almost identical except that S contains the meaningless interleavings as well. So a naïve implementation of ISP would simply discard the *backtrack* set and refer only to S instead. We now explain how this algorithm works by referring to the interleavings shown in Figure 4. Note that these interleavings correspond to the MPI program of Figure 1.

S is initialized so that its first element contains both P0 and P1. The first element of *backtrack* contains only P1, chosen at random. *depth* is initialized to 0. In lines 12-14 we choose a process randomly from *backtrack* at depth 0. Note that for the entire first interleaving, there will only be one possible choice at each depth. We then indicate to P1 that it may make its MPI call `MPI_Win_unlock`, by answering its appeal with a go-ahead token. ISP must now update its internal

bookkeeping information. It does so in line 19 by noting which processes are blocked/unblocked as a result of executing the chosen MPI process. Note that we have reached a point in the search where *backtrack* will indicate no possible choices in the next step. Line 29 is responsible for calculating the runnable processes so they can be added to the *backtrack* set. Again, both P0 and P1 will be added to S as runnable processes. The last step is to increment *depth* and continue our random depth-first-search algorithm.

The first significant digression from this pattern occurs when P0 calls `MPI_Finalize`. At this point, since both processes have called `MPI_Finalize`, we execute the *else* clause of line 45. The idea is to remove all the choices that we have already made, so that in the next execution of the loop, a different interleaving can be explored. So, we remove the `MPI_Finalize` executed by P0. At this point, the `updateBacktrackInfo` function is called on line 51. The purpose of this function is to traverse up the set S and identify any transitions that may need to be interleaved with the `MPI_Finalize` that was just removed. If any such transitions are identified, the corresponding MPI process is added to the *backtrack* set.

Following our DPOR assumptions, this results in no change to the *backtrack* set. We continue to remove choices until we reach the `P0.3: MPI_Win_unlock` call. This time, the `updateBacktrackInfo` function updates the *backtrack* set to look like backtrack' in Figure 4. This indicates that the `MPI_Win_unlock` functions of P0 and P1 must be interleaved in order to get a different, meaningful interleaving. We continue removing all choices that have already been taken until the backtrack set looks like backtrack". At this point, we are ready to start our search from the beginning.

MPIFunctions	*Dependence*
`MPI_Init`	`None`
`MPI_Send`	`MPI_Send, MPI_Ssend, MPI_Recv`
`MPI_Ssend`	`MPI_Send, MPI_Ssend, MPI_Recv`
`MPI_Recv`	`MPI_Send, MPI_Ssend`
`MPI_Barrier`	`None`
`MPI_Win_lock`	`None`
`MPI_Win_unlock`	`MPI_Win_unlock`
`MPI_Win_free`	`None`
`MPI_Finalize`	`None`

Fig. 6. Supported MPI functions

The DPOR-based algorithm of Figure 5 identifies all such meaningful interleavings and terminates the search either when (i) it encounters a deadlock scenario indicated on line 30, or (ii) the search is completed. The table of commuting MPI operations assumed by ISP is in Figure 6.

4 Case Study: Byte-Range Locking

Our previous work in [12] describes how we model checked the byte-range-locking protocol presented in [15]. This uncovered a subtle but crucial deadlock bug that had gone unnoticed during testing. With the help of ISP, we were able to successfully catch this bug in the source code of this protocol. Note that this required no modeling effort whatsoever. In fact, no changes to the source code

were required. The results are presented in Figure 7. ISP has been tested on other smaller protocols and has worked as expected. It can be viewed as an exhaustive testing facility which gives the *effect* of examining all interleavings of small but intricate MPI programs.

Program	*#procs*	*interleavings w/o DPOR*	*interleavings with DPOR*
`byterange reduced depth`	2	2289	119
`byterange full depth`	2	-	1522

Fig. 7. Experimental results

The most striking feature of these results is that ISP was unable to find this bug without using DPOR. The search algorithm was aborted once it did not finish within 24 hours. Our knowledge of the algorithm allowed us to reduce the search depth and find the bug more quickly. However, by enabling DPOR within ISP, we were able to reproduce the deadlock scenario without having to reduce the search depth. While a hand-written model of the same protocol using SPIN could find the same bug without employing partial-order reduction [12], with ISP we have eliminated the nontrivial task of modeling MPI programs in Promela. The ISP approach is especially beneficial if the intervening C statements between MPI calls cannot easily be modeled in Promela, the actual MPI library in use cannot faithfully be modeled, and/or the error is triggered by a bug in the MPI library.

5 Related Work and Conclusions

Model checking has been used for verifying MPI programs by Siegel *et al.* in [13, 14]. The closest related work to ours is [18] where *distributed* in-situ model checking for Pthreads programs has been presented.

In our experience with the byte-range-locking algorithm, the initial program developed and presented in [15] did not exhibit any discernible bugs, despite conventional testing. However, porting of the same program onto a laptop machine caused deadlock. This prompted us to model the protocol in Promela, with the bugs reported in [12]. This paper comes a full circle, and shows that the same bugs can be detected at the C program level *without* model extraction.

Clearly, restarting ISP from `MPI_Init` in order to explore each new interleaving is a huge overhead even with partial-order reduction dramatically reducing the number of interleavings (although it needs to be done only during code development and testing). To reduce the overhead further, we plan to explore three ideas: (i) divide the program using MPI barriers, and interleave only the code between two subsequent barriers. This will cut down the extent of interleavings, and also help localize errors; (ii) the use of MPI checkpointing systems (*e.g.*, [5]) to see whether we can checkpoint intermediate states and restart from there as opposed to restarting the search from `MPI_Init`; (iii) distributed ISP. In [18], a Pthreads program that took 11 hours on a single node was verified in 11 minutes using 64 processors, but only after finding suitable heuristics for work distribu-

tion (albeit for Pthreads where *most of the details are different*). We hope to research these topics in the context of ISP. The code of ISP is available at [1].

References

1. 2007. Preliminary release of the ISP Software at http://www.cs.utah.edu/formal_verification/isp.tar.gz.
2. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.
3. Jayant DeSouza, Bob Kuhn, Bronis R. de Supinski, Victor Samofalov, Sergey Zheltov, and Stanislav Bratanov. Automated, scalable debugging of mpi programs with intel message checker. In *SE-HPCS '05*, pages 78–82, 2005.
4. Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, pages 110–121, 2005.
5. Q. Gao, W. Yu, W. Huang, and D. K. Panda. Application-transparent checkpoint/restart for MPI programs over InfiniBand. In *ICPP*, August 2006.
6. P. Godefroid. Model checking for programming languages using Verisoft. In *POPL 97: Principles of Programming Languages*, pages 174–186, 1997.
7. Gerard J. Holzmann. *The Spin Model Checker*. Addison-Wesley, 2003.
8. Bettina Krammer and Michael M. Resch. Correctness checking of MPI one-sided communication using MARMOT. In *EuroPVM/MPI 2006*, pages 105–114, September 2006. LNCS 4192.
9. G. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva, and Y. Zou. MPI-CHECK: A tool for checking Fortran 90 MPI programs. *Concurrency and Computation: Practice and Experience*, 15:93–100, 2003.
10. Robert Palmer, Steve Barrus, Yu Yang, Ganesh Gopalakrishnan, and Robert M. Kirby. Gauss: A framework for verifying scientific computing software. In *Workshop on Software Model Checking*, 2005. ENTCS 953.
11. Robert Palmer, Ganesh Gopalakrishnan, and Robert M. Kirby. Semantics driven dynamic partial-order reduction of MPI-based parallel programs. In *PADTAD*, 2007. Accepted; Preprint http://www.cs.utah.edu/formal_verification/verification_environment.
12. Salman Pervez, Ganesh Gopalakrishnan, Robert M. Kirby, Rajeev Thakur, and William Gropp. Formal verification of programs that use MPI one-sided communication. In *EuroPVM/MPI*, pages 30–39, 2006.
13. Stephen F. Siegel. Model checking nonblocking MPI programs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, January 2007.
14. Stephen F. Siegel and George S. Avrunin. Verification of MPI-based software for scientific computation. In *SPIN Workshop*, pages 286–303, April 2004.
15. Rajeev Thakur, Robert Ross, and Robert Latham. Implementing byte-range locks using MPI one-sided communication. In *EuroPVM/MPI*, pages 120–129, September 2005.
16. Jeffrey S. Vetter and Bronis R. de Supinski. Dynamic software testing of MPI applications with Umpire. In *Proc. of SC2000*, pages 70–79, 2000.
17. Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *ASE*, September 2000.
18. Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Distributed dynamic partial order reduction based verification of threaded software. In *Workshop on Model Checking Software (SPIN 2007)*, July 2007. accepted.

```
S.push_back(<0...n-1>)
/* randomly choose a proc to run at each depth, e.g. n-1 */
backtrack.push_back(<n-1>)
done.push_back(<n-1>)

if(!fork())
  execlp(MPI program) /* run the given MPI program */
make all server connections

while(backtrack.size() > 0) {
  /* assert backtrack.size() > 0 */
  current choice = pick randomly from backtrack
  get readable envelopes for all runnable processes
  current envelope = envelope for current choice

  /* let the chosen MPI process know that it may run */
  servers[current choice] << goahead

  update block/unblock info for all processes based on current choice and current envelope

  if(chosen process executed MPI_Finalize) {
    specify that current choice is DONE
    close(servers[current choice])
    decrement active procsses
  }
  if(active processes != 0) {
    /* if we have reached a depth where backtrack shows no runnable processes. */
    if(depth+1 >= backtrack.size()) {
      rprocs = <all currently runnable processes>
          if(rprocs.size() == 0) {
            /* POSSIBLE DEADLOCK */
            close all socket connections
            report deadlock and print trace
          }
          else {
            /* the current interleaving has not been explored to current depth */
            S.add_last(<all runnable procs>)
            /* randomly choose a proc to run at each depth, e.g. last proc in S.last */
            backtrack.add_last(<S.last.last>)
            done.add_last(<empty>)
          }
    }
    depth++
  }
  else {
    /* we have gone through one interleaving of the program. Remove all choices from S
    as well as backtrack until the last decision point. A decision point is where we
    had more than 1 choice of MPI processes. */

    while(backtrack.size() > 0 && backtrack.last.size() == 1) {
      updateBacktrackInfo()
      S.remove_last()
      backtrack.remove_last()
      done.remove_last()
    }
    /* make sure search is not over */
    if(backtrack.size() > 0) {
      remove most recent choice from backtrack at current depth
      /* In the next interleaving we will be forced to take an alternate route at this point */

      reset checker state for next interleaving
      if(!fork())
        execlp(MPI program)

      make all server connections
      depth = 0
      active procs = n
    }
  }
}
```

Fig. 5. DPOR-based scheduling algorithm