

Formal Verification of Programs That Use MPI One-Sided Communication

Salman Pervez¹, Ganesh Gopalakrishnan¹, Robert M. Kirby¹,
Rajeev Thakur², and William Gropp²

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 methods based on model checking are applied to analyze the correctness properties of one existing and two new distributed locking protocols implemented using MPI's one-sided communication. Model checking exposed an overlooked correctness issue with the first of these protocols which was developed relying only on manual reasoning. Model checking helped confirm the basic correctness properties of the two new protocols, while also identifying the remaining problems in them. Our experience is that MPI based programming, especially the tricky and relatively poorly understood one-sided communication features, stand to gain immensely from model checking. Considering that many other areas of concurrent hardware and software design are now routinely employing model checking, our experience confirms that the MPI community can greatly benefit from the use of formal verification.

1 Introduction

Concurrent protocols are notoriously hard to design and verify. Experience has shown that virtually all non-trivial protocol implementations contain bugs such as deadlocks, livelocks, and memory leaks, despite extensive care taken during design and testing. It has also been realized that most of these bugs are basic *design* errors due to “unexpected” (untested) concurrent behaviors. Therefore, it stands to reason that if finite state models of these protocols are created, and *exhaustively* analyzed for the desired formal properties, robust protocol implementations would result. The know-how for such finite state modeling, property description, and exhaustive analysis developed over the last three decades – known as *model checking* [2] – has, indeed, been successfully applied to numerous software and hardware systems. It is now an integral part of the Windows Device Driver Development Kit [1]. Virtually *all* cache coherence protocols developed and deployed in modern microprocessors have been verified using model checking. However, despite the overall nature of concurrency as well as concurrent programming bugs being similar in scientific programming as with these

other areas, we find very little evidence of model checking being applied in this area (related work is in Section 6).

In this paper, we conduct case studies that show the promise of the application of model checking in the area of parallel scientific programming using MPI. In particular, we focus on MPI one-sided communication [?]. Being (relatively) recently introduced, one-sided communication is insufficiently understood as well as documented. One-sided communication involves shared memory concurrency, which is known to be inherently harder to reason about than the message passing concurrency of traditional MPI. One-sided communication exacerbates verification complexity because it only guarantees weak ordering semantics with respect to loads and stores which can freely reorder within a given synchronization epoch. This paper demonstrates that using model checking, bugs in MPI programs that use one-sided communication can be caught easily, while expending only modest amounts of human and computer-time. After presenting background on MPI one-sided communication in Section ??, we provide an overview of model checking in Section 2. We then describe the design of an existing distributed byte-range locking protocol [12] and its formal verification through model checking (Section 3). Model checking helped uncover the serious problem of a potential deadlock, of which the authors of the algorithm were unaware. Model checking also found a more benign problem of extra (zero-byte) sends in the algorithm, which might lend itself to an implementation dependent correction using `MPI_Iprobe` and posted receives. However, it is entirely possible for this problem to turn into a memory leak. We then present two alternate designs of the same protocol, formally verify them using model checking, and also provide empirical observations to interpret these model checking results (Section 4). Discussions, related work, and concluding remarks are presented in Section 6.

1.1 MPI One-sided Communication

MPI one-sided communication features [?] allow processes to gain exclusive access to communication windows in a block of code bracketed by the `MPI_Win_lock` and `MPI_Win_unlock` calls. Read and write accesses can be performed by a process holding exclusive access to a window through `MPI_Put` and `MPI_Get`. The main semantic difficulty stems from these `Put` and `Get` calls being not required to obey their syntactic program order in terms of when they are performed. It is known from any elementary textbook (e.g., [10]) that such ordering guarantees are crucial to the correctness of even simple concurrent protocols such as Peterson’s mutual exclusion. The specification of one-sided communication in MPI further exacerbates the issue by introducing a complex set of informally stated rules that can easily lead to contradictory interpretations.³ Common mistakes seen include nesting synchronization epochs on the same window object (such as a `win_lock/unlock` within a fence), doing read-modify-writes via a `Get-modify-Put` in the same synchronization epoch (even though gets and puts are defined

³ A collaborative project between Utah and Argonne is addressing the issue of elucidating as well as formalizing this specification.

to be nonblocking), and doing a put and a get to/from the same memory location in the same synchronization epoch. For example, the broadcast algorithms in Appendix B and C of [3] are incorrect because they rely on `MPI_Get` being a blocking function, which it is not. In implementations that take advantage of the nonblocking nature of `MPI_Get`, such as MPICH2 [11], the code will, indeed, deadlock. Since MPI one-sided communication can be implemented in a variety of ways [5], the result of making such mistakes is often implementation dependent: the program may work fine on some implementations and not on others.

2 Model Checking

Model checking is a term that has acquired an overloaded meaning. It essentially is the activity of exhaustively examining all possible behaviors of a *model* of a concurrent program (akin to wind tunnel testing of scale models of airplanes). We consider *finite state* model checking where the model of the concurrent system is expressed in a modeling language – Promela [6] in our case (all the pseudo-codes expressed in this paper have an almost direct Promela encoding). By exhaustively executing the concurrent system model, a model checker reveals its entire state transition structure, and is able to establish temporal properties such as “always P,” “A implies eventually Q,” etc, with respect to this structure. The state graphs we generate in our work are a result of the *interleaved* execution of various processes/threads. A fundamental problem with model checking is that reachable state graphs are exponential in the number of concurrent processes. The last three decades of research has, essentially, been about getting a good handle on this exponential growth, so much so that astronomically large finite state spaces – or often even many classes of infinite state spaces – can be handled by model checkers. Despite the very large state spaces of the MPI models discussed in this paper, our model checking runs finished within acceptable durations (often in minutes) on standard workstations.

3 Formal Verification of Byte-Range Locking

In [12], Thakur *et al.* present an algorithm implemented using MPI one-sided communication (with passive-target synchronization lock-unlock) for coordinating a collection of parallel processes contending for byte-range locks. We first describe the algorithm briefly (details in [12]), followed by a description of how we model checked it. Due to space limits, we cannot present the full pseudocode of the original protocol; the reader may kindly refer to the original paper [12] for details.

3.1 The Byte-Range Locking Algorithm

The algorithm is split into two phases. The lock acquire phase and the lock release phase. Each process in the system keeps a global state consisting of a

`flag` bit (initialized to 0) and its `start` and `end` values (initialized to -1) in a shared memory window `lockwin`. A flag value of 0 indicates that the process is not currently contending for the lock, while 1 indicates that it either has acquired the lock or is contending for the lock. A process updates its global state and reads others' by acquiring an exclusive lock to `lockwin` and making `MPI_Put` and/or `MPI_Get` calls. The semantics of MPI one-sided guarantee that these actions on `lockwin` are atomic. This means that if two processes simultaneously try to write into and read from `lockwin`, one will succeed while the other will have to wait.

In order to acquire the lock, a process sets its `flag` bit to 1 and updates its `start` and `end` values. It then checks to see if the lock is held/wanted by a process whose byte-range conflicts with its own. If it finds such a process, it blocks on an `MPI_Recv` call. Otherwise, it successfully acquires the lock. The process holding the lock must do a similar check when it releases the lock. If it finds a process with a conflicting byte-range that tried to do an acquire while it was holding the lock, it makes an `MPI_SEND` call to that process.

3.2 Checking the Byte-Range Locking Algorithm

To model the algorithm, we were first required to model the MPI one-sided communication constructs used in the algorithm, and capture their semantics precisely as specified in the MPI Standard. For example, the MPI-2 Standard specifies that if a communication epoch is started with `MPI_Win_lock`, it must end with `MPI_Win_unlock` and that the put/get/accumulate calls made within this epoch are not guaranteed to complete before `MPI_Win_unlock` returns. Furthermore, there are no ordering guarantees of the puts/gets/accumulates within an epoch. Therefore, in order to obtain adequate execution space coverage, *all permutations of put/get/accumulate calls in the epoch must be examined*. However, the byte-range locking algorithm uses the `MPI_LOCK_EXCLUSIVE` lock type, which means that while a certain process has entered the synchronization epoch, no other process may enter until that process has left. This makes the synchronization epoch an atomic block and renders all permutations of the calls within it equivalent from the perspective of other processes. Modeling the byte-range locking algorithm itself was relatively straightforward.⁴

When we model checked our model with three processes, our model checker SPIN [6] discovered an error indicating an “invalid end state.” Upon deeper probing, the following error scenario was identified (explained through an example, which assumes that P1 tries to lock byte-range $\langle 1, 2 \rangle$, P2 tries to lock $\langle 3, 4 \rangle$, and P3 tries to lock $\langle 2, 3 \rangle$):

- P1 and P3 successfully pass the acquire phase, and are using their respective byte-range locks.

⁴ This augurs well for the checking of other algorithms in the area of MPI one-sided communication, as one of the significant challenges in model checking lies in the ease of modeling constructs in the target domain using modeling primitives in the modeling language.

- P2 enters now, notices conflict with respect to both P1 and P3, and blocks on the MPI_Recv.
- P1 and P3 carry out their release phases, both noticing conflicts with P2, and both performing MPI_Send, when only one send is needed.

Hence, while P2 ends up successfully waking up and acquiring the lock, the extra MPI_Sends may accumulate in the system. This was the problem identified by SPIN. This is a subtle error whose severity depends on the MPI implementation being used. Recall that the MPI standard allows implementors the freedom to decide whether or not to block on an MPI_Send call.

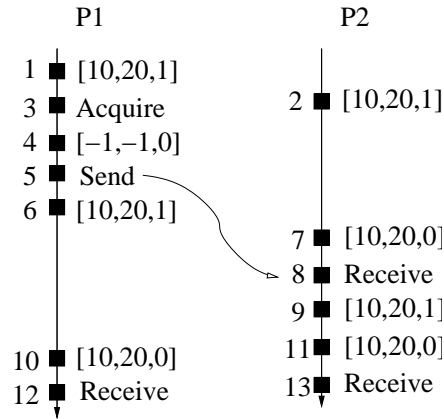


Fig. 1. A Deadlock Scenario Found Through Model Checking

We then proceeded to model the system as if these extra MPI_Sends do not exhaust the system resources, and hence do not cause processes to block. In this case, model checking detected a far more serious deadlock situation summarized in Figure 1, and as follows. P1 expresses its intent to acquire a lock in the range $\langle 10, 20 \rangle$ (1), with P2 following suit (2). P1 acquires the lock (3), finishes using it and relinquishes it (4), and performs a **Send** to unblock P2 (5). Before P2 gets a chance to change its global state, P1 tries to re-acquire the lock (6). P1 reads P2's current flag value as 1, so it decides to block by carrying out events (10) and (12). At this point, P2 changes its global state, receives the message sent by P1 (8), and proceeds to re-acquire the lock (9). P2 reads P1's current flag value as 1, so it decides to block by carrying out events (11) and (13). Both processes now block on Receive calls and the result is deadlock. This paper proposes two solutions that avoid this deadlock, and describes our experience with model checking on these solutions.

4 Correcting the Byte-Range Locking Algorithm

We present two alternative approaches to correcting the byte-range locking algorithm.

Alternative 1: One way to eliminate deadlocks is to add a third state to the “flag” used in the algorithm. This is shown in the pseudo code in Figure 2. In the algorithm of [12] a flag value of ‘0’ indicates that the process does not have the lock, while a flag value of ‘1’ indicates that it either has the lock or is in the process of determining whether it can acquire the lock. In other words, the ‘1’ state is overloaded. In the proposed fix, we add a third state of ‘2,’ with ‘0’ denoting the same as before, ‘1’ now denoting that the process has the lock, and ‘2’ denoting that it is in the process of determining whether it can acquire the lock. There is no change to the lock-release algorithm, but the lock-acquire algorithm changes as follows.

When a process wants to acquire a lock, it writes its flag value as ‘2’ and its start and end values in global memory indicating its intent to try for the lock. It now reads the state of the other processes from global memory. If it finds a process with a conflicting byte range and a flag value of ‘1’, then the process knows that it does not have the lock, and it resets its flag value to ‘0’ and blocks on an `MPI_Recv` call. If no such process (with conflicting byte range and flag=1) is found, but there is another process with a conflicting byte-range and a flag value of ‘2’, the process resets its flag to ‘0’, its start and end offsets to -1, and retries the lock from scratch. If neither of these cases is true, the process sets its flag value to ‘1’ and safely acquire the lock. An assessment of this protocol using model checking is presented later in this section.

Alternative 2: In this approach, when a process tries to acquire a lock and determines that it does not have the lock, it picks a process (that currently has the lock) to wake it up and then blocks on the receive. For this purpose, we add a fourth field (the pick field) to the values for each process in locklist. This change is shown in lines 23-28, Figure 3. The process about to block must now make a decision whether or not to block. This decision is based on two factors: (i) has the process selected to wait upon already released the lock, and (ii) is there a possibility of a deadlock. The former can be easily determined by reading the updated locklist. If the selected process has left, a new process must be picked in its place. We simply traverse the list of conflicting processes until we find one that has not yet left. If no such process is found, the algorithm tries to re-acquire the lock. Note the added complexity of going through the list of conflicting processes and each time doing a Put/Get from shared memory. However, if this loop is successful and it is possible to block, we can save considerable processor time in a highly contentious locking system.

Now we discuss how a potential deadlock is avoided. For a deadlock to occur in this algorithm, there must be a cycle of processes that wait on each other to be woken up. We avoid this deadlock following the algorithm in Figure 4 which, essentially, looks for a cycle among all the processes that are about to wait on each other.

```

1  Lock_acquire ( int start , int end)
2  {
3      val [0] = 2; /* flag */ val [1] = start ; val [2] = end ;
4      while (1) {
5          /* add self to locklist */
6          MPI_Win_lock (MPI_LOCK_EXCLUSIVE , homerank , 0, lockwin);
7          MPI_Put (& val , 3, MPI_INT , homerank , 3*( myrank ), 3, MPI_INT ,
8                  lockwin );
9          MPI_Get (locklistcopy , 3*( nprocs -1) , MPI_INT , homerank , 0, 1,
10                 locktype1 , lockwin);
11         MPI_Win_unlock ( homerank , lockwin );
12         /* check to see if lock is already held */
13         conflict = 0;
14         flag1 = 0;
15         flag2 = 0;
16         for (i=0; i < ( nprocs - 1); i ++ ) {
17             if (( flag == 1) && ( byte ranges conflict with lock request )) {
18                 flag1 = 1;
19                 break;
20             }
21             if (( flag == 2) && ( byte ranges conflict with lock request )) {
22                 flag2 = 1;
23                 break;
24             }
25         }
26         if ( flag1 == 1) {
27             /* reset flag to 0, wait for notification, and then retry the lock */
28             MPI_Win_lock ( MPI_LOCK_EXCLUSIVE , homerank , 0, lockwin );
29             val [0] = 0;
30             MPI_Put (val , 1, MPI_INT , homerank , 3*( myrank ), 1, MPI_INT ,
31                     lockwin );
32             MPI_Win_unlock ( homerank , lockwin );
33             /* wait for notification from some other process */
34             MPI_Recv (NULL , 0, MPI_BYTE , MPI_ANY_SOURCE , WAKEUP , comm ,
35                      MPI_STATUS_IGNORE );
36             /* retry the lock */
37             Lock_acquire(start, end);
38         }
39         else if ( flag2 == 1) {
40             /* reset flag to 0, start/end offsets to -1, and then retry the lock */
41             MPI_Win_lock ( MPI_LOCK_EXCLUSIVE , homerank , 0, lockwin );
42             val [0] = 0; /* flag */ val [1] = -1 ; val [2] = -1 ;
43             MPI_Put (val , 3, MPI_INT , homerank , 3*( myrank ), 3, MPI_INT ,
44                     lockwin );
45             MPI_Win_unlock ( homerank , lockwin );
46             /* retry the lock */
47             Lock_acquire(start, end);
48         }
49         else {
50             MPI_Win_lock ( MPI_LOCK_EXCLUSIVE , homerank , 0, lockwin );
51             val [0] = 1;
52             MPI_Put (val , 1, MPI_INT , homerank , 3*( myrank ), 1, MPI_INT ,
53                     lockwin );
54             MPI_Win_unlock ( homerank , lockwin );
55             /* lock is acquired */
56             break ;
57         }
58     }
59 }

```

Fig. 2. Pseudocode for the deadlock-free byte-range locking algorithm (alternative 1)

```

1 Lock_acquire ( int start , int end)
2 {
3     val [0] = 1; /* flag */ val [1] = start ; val [2] = end ; val [3] = -1 /* pick */ ;
4     int picklist[num_procs];
5     while (1) {
6         /* add self to locklist */
7         MPI_Win_lock ( MPI_LOCK_EXCLUSIVE , homerank , 0, lockwin );
8         MPI_Put (& val , 4, MPI_INT , homerank , 4*( myrank ), 4, MPI_INT ,
9             lockwin );
10        MPI_Get (locklistcopy , 4*( nprocs-1 ) , MPI_INT , homerank , 0, 1,
11            locktype1 , lockwin );
12        MPI_Win_unlock ( homerank , lockwin );
13        /* check to see if lock is already held */
14        cprocs_i = 0;
15        for (i=0; i < ( nprocs - 1); i ++ ) {
16            if (( flag == 1) && ( byte range conflicts with Pi's request )) {
17                conflict = 1; picklist[cprocs_i] = Pi; cprocs_i++; }
18        }
19        if (conflict == 1) {
20            for(j=0; j < cprocs_i; j++) {
21                MPI_Win_lock ( MPI_LOCK_EXCLUSIVE , homerank , 0, lockwin );
22                val [0] = 0; val [3] = picklist [j];
23                /* reset flag to 0, indicate pick and pick_counter */
24                MPI_Put (&val , 4, MPI_INT , homerank , 4*( myrank ), 4, MPI_INT ,
25                    lockwin);
26                MPI_Win_unlock ( homerank , lockwin );
27                if(picklist [j] has released the lock) || detect_deadlock() {
28                    /* repeat for the next process in picklist */
29                    j++;
30                }
31                else {
32                    /* wait for notification from picklist [j] */
33                    MPI_Recv (NULL , 0, MPI_BYTE , MPI_ANY_SOURCE , WAKEUP , comm ,
34                        MPI_STATUS_IGNORE );
35                    /* retry the lock */
36                }
37            }
38            /* if the entire list has been traversed, retry the lock */
39        }
40        else {
41            break ; /* lock is acquired */
42        }
43    }
44 }

```

Fig. 3. Pseudocode for the deadlock-free byte-range locking algorithm (alternative 2)

```

1 detect_deadlock() {
2     cur_pick = locklistcopy[4 * myrank + 3];
3     while(i < num_procs) {
4         /* picking this process means a cycle is completed */
5         if(locklistcopy[4 * cur_pick + 3] == my_rank)
6             return 1;
7         /* no cycle can be formed */
8         else if(locklistcopy[4 * cur_pick + 3] == -1)
9             return 0;
10        else
11            cur_pick = locklistcopy[4 * cur_pick + 3];
12    }
13 }

```

Fig. 4. Avoiding circular loops among processes picked to wake up other processes in alternative 2

An Assessment of the Alternative Algorithms: Model checking using SPIN has helped establish the following formal properties of these algorithms:

- Absence of deadlocks (both alternatives)
- Communal progress (if a collection of processes request a lock then someone will eventually obtain it): Alternative 2 satisfies this under all fair schedules (all processes are scheduled to run infinitely often), while Alternative 1 places a few additional restrictions to rule out a few rare schedules (details on our website in our technical report [?]).

We note that neither of these alternatives eliminates the extra sends. That said, Alternative 2 considerably reduces these extra sends, as it restricts the number of processes that can wake up a particular process to 1. The exact performance tradeoffs will be determined as part of our future work. We are still seeking algorithms that: (i) avoid deadlock, (ii) avoid extra sends, and (iii) are efficient.

5 Related Work

Excellent tutorials as well as research papers on model checking are widely available, and we omit a survey here. In [7], the author surveys the area of parallel scientific programming and techniques to portray executions through intuitive graphical means. He offers a formally-based tool that can help understand MPI program executions. In [9], the authors present the use of Promela and SPIN for modeling and verifying simple MPI C programs (2D diffusion) that employ basic MPI message passing constructs. To our knowledge, nobody has applied model checking to analyze one-sided communication constructs. As remarked earlier, the application of formal verification methods can often have significant impact when applied to relatively new (and hence relatively less understood) parallel programming primitives, as well as their applications.

6 Conclusions and Future Work

We have shown how formal verification based on model checking can be used to find actual deadlocks in published algorithms that use the MPI one-sided communication primitives. We also discuss how this technology can help shed light on a number of related issues such as forward progress, as well as the possibility of having unconsumed messages. We presented and analyzed two alternative algorithms, and analyzed their characteristics.

Our work is still in its early stages. Capitalizing on the maxim that formal methods can have their biggest impact when applied to constructs that are relatively new or are under development, we plan to formalize the entire set of MPI one-sided communication primitives. This can help develop a comprehensive approach to verifying programs that use the MPI one-sided constructs. As future case studies, we plan to analyze numerous other algorithms, including the scalable fetch-and-increment algorithm described in [4]. We plan to explore the use

of automated tools to extract models from MPI programs, instead of creating them by hand. We plan to explore the advantages of using other modeling languages such as +CAL [8], and also investigate the possibility of directly model checking MPI programs, instead of their extracted formal models.

Acknowledgments

This work was supported by NSF award CNS-0509379 and by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38.

References

1. Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside microsoft. In *IFM 04: Integrated Formal Methods*, pages 1–20. Springer-Verlag, April 2004.
2. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, December 1999.
3. Marina Kraeva Glenn R. Luecke, Silvia Spanoyannis. The performance and scalability of SHMEM and MPI-2 one-sided routines on a SGI Origin 2000 and a cray T3E-600. *Concurrency and Computation: Practice and Experience*, 16(10):1037–1060, 2004.
4. William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.
5. William Gropp and Rajeev Thakur. An evaluation of implementation options for MPI one-sided communication. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 12th European PVM/MPI Users' Group Meeting*, pages 415–424. Lecture Notes in Computer Science 3666, Springer, September 2005.
6. Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003. ISBN 0-32122-862-6.
7. Dieter Kranzlmler. *Event Graph Analysis for Debugging Massively Parallel Programs*. PhD thesis, John Kepler University Linz, Austria, September 2000. <http://www.gup.uni-linz.ac.at/~dk/thesis>.
8. Leslie Lamport, 2006. <http://research.microsoft.com/users/lamport/tla/pluscal.html>.
9. Stephen F. Siegel and George Avrunin. Verification of mpi-based software for scientific computation. In *Proceedings of the 11th International SPIN Workshop on Model Checking Software*, pages 286–303, April 2004. LNCS volume 2989.
10. A. S. Tanenbaum. *Distributed Operating Systems*, chapter 6, pages 289–375. Prentice-Hall, Inc., 1995.
11. Rajeev Thakur, William Gropp, and Brian Toonen. Optimizing the synchronization operations in MPI one-sided communication. *International Journal of High-Performance Computing Applications*, 19(2):119–128, Summer 2005.
12. Rajeev Thakur, Robert Ross, and Robert Latham. Implementing byte-range locks using MPI one-sided communication. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 12th European PVM/MPI Users' Group Meeting*, pages 120–129. Lecture Notes in Computer Science 3666, Springer, September 2005.