Formal Methods Applied to HPC Software Design: A Case Study of Locking Based on MPI One-Sided Communication

Salman Pervez, Ganesh Gopalakrishnan, Robert M. Kirby,
Rajeev Thakur, and William Gropp

Abstract

There is growing need to address the complexity of modeling and verifying the numerous concurrent protocols involved in high-performance computing (HPC) software design. Finite-state modeling and formal verification (FV) using model-checking technology have made impressive inroads into debugging concurrent protocols. However, there remains a dearth of research in applying model-checking methods to HPC software design. This situation can be attributed to the lack of awareness by the HPC community about the potentials of FV in the HPC arena, as well as lack of awareness by the FV community of the challenges that HPC application domains pose. In this paper, we demonstrate the utility of finite-state modeling and model checking in detecting race conditions and deadlocks in concurrent protocols that arise while developing HPC software. In particular, we detail a case study that develops a distributed byte-range locking algorithm using MPI's one-sided communication. Our model-checking effort detected a race condition that can cause deadlock, of which the authors of the algorithm were unaware. We describe two designs to fix the deadlock problem, and we present their formal analysis using model checking, and their performance analysis on a 128-node cluster. Our objective is to give practitioners, especially those developing parallel programs using libraries, the opportunity to accurately assess the costs and benefits of finite-state modeling and model checking.

S. Pervez, G. Gopalakrishnan, and R. M. Kirby are with the School of Computing, University of Utah. R. Thakur and W. Gropp are with the Mathematics and Computer Science Division, Argonne National Laboratory.

Index Terms

Parallel programming, race condition, deadlock, formal verification, model checking, SPIN, Message Passing Interface (MPI), one-sided communication

I. Introduction

Debugging concurrent protocols is one of the fundamental challenges facing the computing community today. Accurate protocols are especially needed given the increasing importance of making advances in high-performance computing (HPC) and the sheer number of concurrent protocols involving many concurrency libraries that will be used in realizing these advances. Model checking [1], [2], [3] is widely regarded as the primary technique of choice for debugging a significant number of concurrent protocols. After 25 years of research [4] that has resulted in many important theoretical discoveries as well as practical applications and tools, designers in many application areas now possess a good understanding of how and to what extent model checking can help in their work. Yet, when we began our research on applying formal methods for debugging high-performance computing software about three years ago, we were surprised to discover that designers in HPC did not know about model checking or its potential to help them debug concurrent protocols of the kind they are now developing.

After an initial dialog was established between Argonne National Laboratory (interested primarily in HPC, and the principal developers of the MPICH2 MPI library [5]) and the University of Utah (interested primarily in model checking), the Utah researchers were given a modest-looking challenge problem by their Argonne colleagues: See what model checking can do in terms of analyzing a then-recently published *byte-range locking* algorithm [6] developed by some Argonne researchers. The challenge problem held (and still holds) considerable appeal. It uses MPI's recent extensions that involve shared memory, called *one-sided communication* (or remote-memory access) [7], which allows a process to directly read from or write to another process's memory. In other words, it is more like shared memory than message passing and comes with

the associated problem of a shared-memory programming style, namely, the potential for race conditions. At the time the challenge was issued, one-sided communication had proved to be rather slippery territory for previous MPI users. We emphasize that the challenge problem was not a contrived example. It was developed to enable an MPI-IO implementation to acquire byte-range locks in the absence of POSIX `fcntl()` file locks, which some high-performance file systems do not support.

The challenge problem was as follows. Imagine processes wanting read/write accesses to contiguous ranges of bytes maintained in some storage device. In order to maintain mutual exclusion, the processes must detect conflicts between their byte ranges of interest through a suitable concurrent protocol. Unfortunately, the protocol actions of various processes can occur only through one-sided operations that, in effect, perform "puts" and "gets" within one-sided synchronization epochs. Since these puts and gets are unordered, popular paradigms of designing locking protocols that depend on sequential orderings of process actions (e.g., Peterson's algorithm [8] involves setting one's own "interested" bit and then seeing which other process has its turn) do not work for the stated byte-range problem. The authors of [6] had devised a solution involving back offs and retries. Unlike "textbook" locking protocols whose behaviors have been studied under weak memory models (e.g., [9]), the principal appeal of the locking protocol (detailed in Section III) from a model-checking perspective stems from its use of realistic features from a library (MPI) that is considered the de-facto standard of parallel programming.

This paper makes the following contributions. It demonstrates that by using finite-state modeling and model checking early during the design of concurrent protocols, many costly mistakes can be avoided. In particular, it discusses the race conditions latent in our challenge problem (the algorithm of [6]) that result in deadlocks. These errors were found relatively easily by using model checking. Curiously, neither the authors of the algorithm nor the reviewers of the

published paper [6] were aware of the race condition, and conventional testing had missed the associated bugs. To solve the deadlock problem, we present two other algorithms, which we analyze by using model checking and show to be free of race conditions and other problems affecting correctness. However, the algorithms have remaining resource issues (as did the original algorithm) that must be addressed on pragmatic grounds. An initial assessment we present in this paper is that these resource issues are no worse than in the original buggy protocol. In addition, since performance is a crucial aspect of any HPC design, an engineer must ultimately conduct performance analysis in addition to verifying correctness. In this regard, our performance assessment of the alternative protocols on a 128-node cluster reveals that Alternative 2 performs far better than Alternative 1, under both low and high lock contention.

This paper details our contributions, discusses a few limitations of our current model-checking approach, and briefly describes our planned work to overcome these limitations. It also discusses the important lesson of having to build techniques and tools that help engineers iterate through the option space in search of optimal choices that balance correctness, performance, and resource issues.

Related Work. The area of formal methods applied to concurrent-program design has a history of over 50 years of research and hence is too vast to present. Even those efforts directed at the use of model checking for concurrent and distributed program verification are vast; applications in telecommunication-software design (e.g., [10]), aerospace software (e.g., [11]), device-driver design (e.g., [12]), and operating-system kernels (e.g., [13]) are four examples of recent efforts. Focusing on the use of formal methods for HPC software design, and in particular for MPI-based parallel/distributed program design, one finds an increasing level of activity from a handful of researchers. Siegel et al. have used model checking to verify MPI programs that employ a limited set of two-sided MPI communication primitives [14], [15], [16], [17]. We have also employed model checking to formally analyze as well as verify MPI programs that use two-sided constructs

[18], [19], [20], [21]. Kranzlmüller used a formal event-graph based method to help understand MPI program executions [22]. Matlin et al. used the SPIN model checker to verify parts of the MPD process manager used in MPICH2 [23]. The case study in this paper involves one-sided communication, applies to a published algorithm, and assesses the solution in terms of resources as well as performance.

Roadmap. The rest of this paper is organized as follows. In Section II, we briefly introduce model checking. In Sections III and IV, we describe the byte-range locking algorithm and how we model checked it. In Section V, we present two designs of the algorithm that avoid the race condition, formally verify them using model checking, provide empirical observations to interpret the model-checking results, and study their performance on up to 128 processors on a Linux cluster. In Section VI, we conclude with a discussion of future work.

II. Overview of Model Checking

Model checking consists of two steps: creating *models* of concurrent systems and traversing these models exhaustively while checking the truth of desired properties. A model can be anything from a simple finite-state machine modeling the concurrent system to actual deployed code. Model checking is performed by creating—either manually or automatically—simplified models of the concurrent system to be verified, recording states and paths visited to avoid test repetitions (an extreme case of which is infinite looping), and checking the desired correctness properties, typically on the fly. Given that the size of the reachable state space of concurrent systems can be exponential in the number of concurrent processes, model checkers employ a significant number of algorithms as well as heuristics to achieve the effect of full coverage without ever storing entire state histories. The properties checked by a model checker can range from simple state properties such as *assert*s to complex temporal logic formulas that express classes of desired behaviors. Model checkers are well known for their ability to track down deadlocks, illegal states (safety [3] bugs), and starvation scenarios (liveness [3] bugs) that may survive years of intense testing. We

consider *finite-state* model checking, where the model of the concurrent system is expressed in a modeling language—Promela [10], in our case. (All the pseudocodes expressed in this paper have an almost direct Promela encoding once the MPI constructs have been accurately modeled.) By (in effect) exhaustively traversing the concurrent-system automaton, a model checker helps establish desired temporal properties, such as "always P" and "A implies eventually Q," or generates concrete error traces when such properties fail.

The capabilities of model checkers have been steadily advancing, with modern model checkers being able to handle astronomically large state spaces (consisting, for example, of billions of distinct states). A model checker that has received wide attention in the computer-science community is SPIN [10]. Despite the very large state spaces of the SPIN MPI models discussed in this paper, our model-checking runs finished within acceptable durations (often in minutes) on standard workstations.

III. The Byte-Range Locking Algorithm

Often, processes must acquire exclusive access to a range of bytes, such as a portion of a file. In [6], Thakur et al. presented an algorithm by which processes can coordinate among themselves to acquire byte-range locks, without a central lock-granting server. The algorithm uses MPI one-sided communication (or remote-memory access) [7]. Since it is essential to understand the semantics of MPI one-sided communication in order to understand the algorithm, we first briefly explain the relevant features of MPI one-sided communication.

A. Overview of MPI One-Sided Communication

MPI-2 added one-sided communication functions that offer a different programming model from the regular MPI-1 point-to-point operations. In one-sided communication, a process can directly write to or read from the memory of a remote process via *put* and *get* operations. Prior to that, a process must specify the memory region that it wishes to allow other processes

Process 0	*Process 1*	*Process 2*
`MPI_Win_create(&win)`	`MPI_Win_create(&win)`	`MPI_Win_create(&win)`
`MPI_Win_lock(excl,1)`		`MPI_Win_lock(excl,1)`
`MPI_Put(1)`		`MPI_Put(1)`
`MPI_Get(1)`		`MPI_Get(1)`
`MPI_Win_unlock(1)`		`MPI_Win_unlock(1)`
`MPI_Win_free(&win)`	`MPI_Win_free(&win)`	`MPI_Win_free(&win)`

Fig. 1. The lock-unlock synchronization method for one-sided communication in MPI. The numerical arguments indicate the target rank.

to directly access. This memory region is called a *window* and is specified via the collective function `MPI_Win_create`. The three functions for one-sided communication are `MPI_Put`, `MPI_Get`, and `MPI_Accumulate`. `MPI_Put` writes to remote memory; `MPI_Get` reads from remote memory; and `MPI_Accumulate` does a reduction operation on remote memory. All three functions are nonblocking: They initiate but do not necessarily complete the one-sided operation. These functions are not sufficient by themselves because one needs to know when a one-sided operation can be initiated (that is, when the remote memory is ready to be read or written) and when a one-sided operation is guaranteed to be completed. To specify these semantics, MPI defines three synchronization methods. For simplicity, we consider only one of them, lock-unlock, which is the one used in the byte-range locking algorithm.

In the lock-unlock method, a process can gain exclusive access to the window of a target process by calling `MPI_Win_unlock`. In the lock-unlock synchronization method, the origin process calls `MPI_Win_lock` to obtain either shared or exclusive access to the window on the target, as shown in Figure 1. `MPI_Win_lock` is not required to block until the lock is acquired, except when the origin and target are the same process. After issuing the one-sided operations, the origin calls `MPI_Win_unlock`. When `MPI_Win_unlock` returns, the one-sided operations are guaranteed to be completed at the origin and the target. The target process does not make any synchronization call.

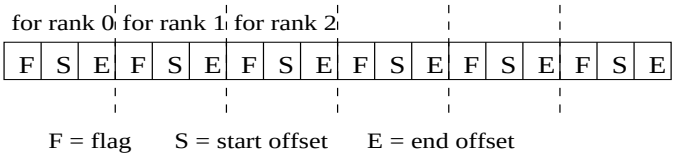

Fig. 2. Window layout for the original byte-range locking algorithm.

Note that MPI puts and gets are nonblocking operations, and an implementation is allowed to reorder them within a lock-unlock synchronization epoch. They are guaranteed to be completed, both locally and remotely, only after the unlock returns. In other words, a get operation is not guaranteed to see the data that was written by a put issued before it in the same lock-unlock epoch. Consequently, it is difficult to implement an atomic read-modify-write operation by using MPI one-sided communication [24]. One cannot simply do a lock-get-modify-put-unlock because the data from the get is not available until after the unlock. In fact, the MPI Standard defines such an operation to be erroneous (doing a put and a get to the same location in the window in the same synchronization epoch). One also cannot do a lock-get-unlock, modify the data, and then do a lock-put-unlock because the read-modify-write is no longer atomic. This feature of MPI complicates the design of a byte-range locking algorithm.

B. The Algorithm

Below we describe the byte-range locking algorithm of [6] together with snippets of the code for acquiring and releasing a lock.

1) Window Layout: The window memory for the byte-range locking algorithm comprises three values for each process, ordered by process rank, as shown in Figure 2. (The window is allocated on any one process, say rank 0.) The three values are a flag, the start offset for the byte-range lock, and the end offset.

2) Acquiring the Lock: The process wanting to acquire a lock calls `MPI_Win_lock` with the `lock_type` as `MPI_LOCK_EXCLUSIVE`, followed by an `MPI_Put`, an `MPI_Get`, and then

MPI_Win_unlock, as shown in Figure 3. With the MPI_Put, the process sets its own three values in the window: It sets the flag to 1 and the start and end offsets to those needed for the lock. With the MPI_Get, it gets the three values for all other processes (excluding its own values) by using a suitably constructed derived datatype, for example, an indexed type with two blocks. (The derived datatype is used because the memory accessed by a put and a get in the same synchronization epoch must not overlap as per the MPI specification.) After MPI_Win_unlock returns, the process goes through the list of values returned by MPI_Get. For all other processes, it first checks whether the flag is 1; if so, it checks whether there is a conflict between that process's byte-range lock and the lock it wants to acquire. If there is no such conflict with any other process, it considers the lock acquired. If a conflict (flag and byte range) exists with any process, it considers the lock as not acquired.

If the lock is not acquired, the process resets its flag in the window to 0 by doing a lock-put-unlock and leaves its start and end offsets in the window unchanged. It then calls a zero-byte MPI_Recv with MPI_ANY_SOURCE as the source and blocks until it receives such a message from any other process (that currently has a lock; see the lock-release algorithm below). After receiving the message, it tries again to acquire the lock by using the above algorithm.

3) Releasing the Lock: The process wanting to release a lock calls MPI_Win_lock with the lock_type as MPI_LOCK_EXCLUSIVE, followed by an MPI_Put, an MPI_Get, and then MPI_Win_unlock, as shown in Figure 4. With the MPI_Put, the process resets its own three values in the window: It resets its flag to 0 and the start and end offsets to -1. With the MPI_Get, it gets the start and end offsets for all other processes (excluding its own values) by using a derived datatype. This derived datatype could be different from the one used for acquiring the lock because the flags are not needed. After MPI_Win_unlock returns, the process goes through the list of values returned by MPI_Get. For all other processes, it checks whether there is a conflict between the byte range set for that process and the lock it is releasing. The flag is ignored

```
Lock_acquire(int start, int end)
{
    val[0] = 1; /* flag */  val[1] = start; val[2] = end;

    while (1) {
        /* add self to locklist */
        MPI_Win_lock(MPI_LOCK_EXCLUSIVE, homerank, 0, lockwin);
        MPI_Put(&val, 3, MPI_INT, homerank, 3*(myrank), 3, MPI_INT, lockwin);
        MPI_Get(locklistcopy, 3*(nprocs-1), MPI_INT, homerank, 0, 1, locktype1,
                lockwin);
        MPI_Win_unlock(homerank, lockwin);

        /* check to see if lock is already held */
        conflict = 0;
        for (i=0; i < (nprocs - 1); i++) {
            if ((flag == 1) && (byte ranges conflict with lock request)) {
                conflict = 1; break;
            }
        }

        if (conflict == 1) {
            /* reset flag to 0, wait for notification, and then retry the lock */
            MPI_Win_lock(MPI_LOCK_EXCLUSIVE, homerank, 0, lockwin);
            val[0] = 0;
            MPI_Put(val, 1, MPI_INT, homerank, 3*(myrank), 1, MPI_INT, lockwin);
            MPI_Win_unlock(homerank, lockwin);

        /* wait for notification from some other process */
        MPI_Recv(NULL, 0, MPI_BYTE, MPI_ANY_SOURCE, WAKEUP, comm,
                     MPI_STATUS_IGNORE);
            /* retry the lock */
        }
        else {
            /* lock is acquired */
            break;
        }
    }
}
```

Fig. 3. Pseudocode for obtaining a byte-range lock in the original algorithm.

in this comparison. If there is a conflict with the byte range set by another process—meaning that process is waiting to acquire a conflicting lock—it sends a 0-byte message to that process, in response to which that process will retry the lock. After it has gone through the entire list of values and sent 0-byte messages to all other processes waiting for a lock that conflicts with its own, the process returns.

```
1  Lock_release(int start, int end)
2  {
3      val[0] = 0; val[1] = -1; val[2] = -1;
4
5      /* set start and end offsets to -1, flag to 0, and get everyone else's status */
6      MPI_Win_lock(MPI_LOCK_EXCLUSIVE, homerank, 0, lockwin);
7      MPI_Put(val, 3, MPI_INT, homerank, 3*(myrank), 3, MPI_INT, lockwin);
8      MPI_Get(locklistcopy, 3*(nprocs-1), MPI_INT, homerank, 0, 1, locktype2,
9              lockwin);
10     MPI_Win_unlock(homerank, lockwin);
11
12     /* check if anyone is waiting for a conflicting lock. If so, send them a
13        0-byte message, in response to which they will retry the lock. For
14        fairness, we start with the rank after ours and look in order. */
15
16     i = myrank;  /* ranks are off by 1 because of the derived datatype */
17     while (i < (nprocs - 1)) {
18         /* the flag doesn't matter here. check only the byte ranges */
19         if (byte ranges conflict) MPI_Send(NULL, 0, MPI_BYTE, i+1, WAKEUP, comm);
20         i++;
21     }
22     i = 0;
23     while (i < myrank) {
24         if (byte ranges conflict) MPI_Send(NULL, 0, MPI_BYTE, i, WAKEUP, comm);
25         i++;
26     }
27 }
```

Fig. 4. Pseudocode for releasing a lock in the original algorithm.

IV. Model Checking the Byte-Range Locking Algorithm

To model the algorithm, we first needed to model the MPI one-sided communication constructs used in the algorithm and capture their semantics precisely as specified in the MPI standard [7]. For example, the MPI 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; therefore, while a certain process has entered the

```
inline MPI_Put(flag, start, end, pick, proc_id) {
  int i=0;
  do
    :: i==NUM_PROCS -> break;
    :: put_requests[i].used -> i++;
    :: else ->
        put_requests[i].used = 1; put_requests[i].req_flag = flag;
        put_requests[i].req_start = start; put_requests[i].req_end = end;
        put_requests[i].req_pick = pick; put_requests[i].req_id = proc_id;
        break;
  od;
}
```

Fig. 5. Promela implementation of `MPI_Put`. Put requests are simply saved in a list, and the list is processed in `MPI_Win_unlock`.

synchronization epoch, no other process may enter until that process has left. This approach makes the synchronization epoch an atomic block and renders all permutations of the calls within it equivalent from the perspective of other processes.

As an example, our Promela implementation of `MPI_Put` is shown in Figure 5. In order to be consistent with the MPI Standard, a put request is simply stored in a list of requests, and the list is processed in `MPI_Win_unlock`. Modeling the byte-range locking algorithm itself was relatively straightforward. (This experience augurs well for the checking of other algorithms that use MPI one-sided communication, since 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.) The complete Promela code used in our model checking can be found online [25].

When we model checked our model with three processes, our model checker, SPIN [10], discovered an error indicating an “invalid end state.” Deeper probing revealed the following error scenario (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 acquire their byte-range locks.

- P2 then tries to acquire its lock, notices conflict with respect to both P1 and P3, and blocks on the `MPI_Recv`.
- P1 and P3 release their locks, both notice conflicts with P2, and both perform an MPI_Send, when only one send is needed.

Hence, while P2 ends up successfully waking up and acquiring the lock, the extra `MPI_Send`s may accumulate in the system. This is a subtle error whose severity depends on the MPI implementation being used. Recall that the MPI Standard allows implementors to decide whether to block on an `MPI_Send` call. In practice, a zero-byte send will rarely block. Nonetheless, an implementation of the byte-range locking algorithm can address this problem by periodically calling `MPI_Iprobe` and matching any unexpected messages with `MPI_Recv`s.

We then modeled the system as if these extra `MPI_Send`s 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 6. 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 reacquire 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 reacquires 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 deadlock is caused by a classic race condition, namely, a particular timing of events that caused P1 to attempt to reacquire the lock (6) before P2 could reset its state (7). We note that the possibility of this race condition was not detected by the authors of the original algorithm during their design and testing nor by the reviewers of the paper describing the algorithm [6]. Of course, after the model checker pointed out the problem, it appeared obvious and could be

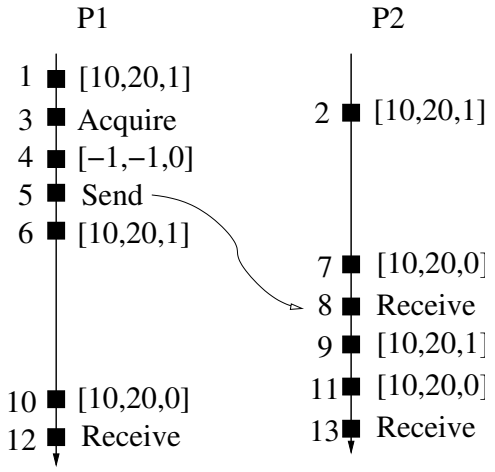

Fig. 6. A deadlock scenario found through model checking.

reproduced in practice. This is again an example of how easy it is for humans to miss potential race conditions and deadlock scenarios, whereas a model-checking tool can easily catch them.

V. Correcting the Byte-Range Locking Algorithm

We propose two approaches to fixing this deadlock problem, describe our experience with using model checking on these solutions, and study their relative performance on a Linux cluster.

A. Alternative 1

One way to eliminate the deadlock is to add a third state to the "flag" used in the algorithm. This is shown in the pseudocode in Figure 7. In the original algorithm, 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 acquired the lock or is in the process of determining whether it has acquired 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 acquired the lock, and '2' denoting that it is in the process of determining whether it has acquired 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 the memory window. It also reads the state of the other processes from the memory window. If it finds a process with a conflicting byte range and a flag value of ‘1,’ it knows that it does not have the lock. So it resets its flag value to ‘0’ and blocks on an `MPI_Recv`. 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’ and 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 considers the lock acquired. (The algorithm is described recursively in Figure 7 only for convenience. It is implemented and modeled as an iteration.)

1) Dealing with Fairness and Livelock: One problem with this algorithm is the issue of fairness: a process wanting to acquire the lock repeatedly may starve out other processes. This problem can be avoided if the MPI implementation grants exclusive access to the window fairly among the requesting processes.

Even in the absence of this problem, model checking revealed the potential for livelock when processes try to acquire the lock multiple times. The situation is similar to the one that caused deadlock in the original algorithm, where two processes trying to acquire the lock both block in order that the other can go ahead. To avoid deadlock, we introduced the intermediate state of 2, which ensures that instead of blocking on an `MPI_Recv`, the process backs off and retries the lock. Figure 8 shows an example of how the back off and retry could repeat forever if events get scheduled in a particular way. This is another example of a race condition that is hard for humans to detect but can be caught by a model checker. The performance graphs presented in Section V-E shows the possibility of these livelocks happening in practice. The possibility of livelock can be reduced, however, by having each process back off for a random amount of time before retrying, thereby avoiding the likelihood of the same sequence of events occurring each time.

```
Lock_acquire (int start, int end)
{
  val[0] = 2; /* flag */
  val[1] = start; val[2] = end;
  /* add self to locklist */
  MPI_Win_lock(MPI_LOCK_EXCLUSIVE, homerank, 0, lockwin);
  MPI_Put(&val, 3, MPI_INT, homerank, 3*myrank, 3, MPI_INT, lockwin);
  /* use derived datatype to get others' info into a contiguous buffer */
  MPI_Get(locklistcopy, 3*(nprocs-1), MPI_INT, homerank, 0, 1,
          locktype1, lockwin);
  MPI_Win_unlock(homerank, lockwin);
  /* check to see if lock is already held */
  flag1 = flag2 = 0;
  for (i=0; i < (nprocs-1); i ++) {
    if ((flag == 1) && (byte ranges conflict with lock request)) {
      flag1 = 1;
      break;
    }
    if ((flag == 2) && (byte ranges conflict with lock request)) {
      flag2 = 1;
      break;
    }
  }
  if (flag1 == 1) {
    /* reset flag to 0, wait for notification, and then retry */
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, homerank, 0, lockwin);
    val[0] = 0;
    MPI_Put(val, 1, MPI_INT, homerank, 3*myrank, 1, MPI_INT, lockwin);
    MPI_Win_unlock(homerank, lockwin);
    /* wait for notification from some other process */
    MPI_Recv(NULL, 0, MPI_BYTE, MPI_ANY_SOURCE, WAKEUP, comm, &status);
    /* retry the lock - iteration shown as tail recursion */
    Lock_acquire(start, end);
  }
  else if (flag2 == 1) {
    /* reset flag to 0, start/end offsets to -1 */
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, homerank, 0, lockwin);
    val[0] = 0; /* flag */
    val[1] = -1; val[2] = -1;
    MPI_Put(val, 3, MPI_INT, homerank, 3*myrank, 3, MPI_INT, lockwin);
    MPI_Win_unlock(homerank, lockwin);
    /* wait for small random amount of time (to avoid livelock) */
    /* then retry the lock - iteration shown as tail recursion */
    Lock_acquire(start, end);
  }
  else {
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, homerank, 0, lockwin);
    val[0] = 1;
    MPI_Put(val, 1, MPI_INT, homerank, 3*myrank, 1, MPI_INT, lockwin);
    MPI_Win_unlock(homerank, lockwin);
    /* lock is acquired */
  }
}
```

Fig. 7. Pseudocode for the deadlock-free byte-range locking algorithm (Alternative 1).

P1 sets flag=2
P2 sets flag=2, sees P1's 2, and decides to retry
P1 acquires the lock
P1 releases the lock and retries for the lock
P1 sets flag=2, sees P2's 2, and decides to retry

P2 sets flag=0
P2 sets flag=2, sees P1's 2, and decides to retry
P1 sets flag=0
P1 sets flag=2, sees P2's 2, and decides to retry

Above sequence repeats

Fig. 8. A potential livelock scenario in Alternative 1 found through model checking.

B. Alternative 2

Alternative 2 uses the same values for the flag as the original algorithm; but 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 awaken it and then blocks on the receive. For this purpose, we add a fourth field (the pick field) to the values for each process in the memory window (see Figure 9). The process trying to acquire the lock must now decide whether to block and wait for notification from picklist[j]. This decision is based on two factors: Has the process selected to wake it up already released the lock? and Is there a possibility of a deadlock caused by a cycle of processes that wait on each other to wake them up? The latter can be detected and avoided by using the algorithm in Figure 10. The former can be easily determined by reading the values returned by the `MPI_Get` on line 24. If the selected process has already released the lock, 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 released the lock. If no such process is found, the algorithm tries to reacquire the lock. Note the added complexity of going through the list of conflicting processes and doing put and get operations each time. However, if this loop is successful and the process blocks on `MPI_Recv`, we can save considerable processor time in the case of highly contentious lock requests as compared with Alternative 1.

```
Lock_acquire (int start, int end)
{
  int picklist[num_procs];
  val[0] = 1; /* flag */
  val[1] = start; val[2] = end; val[3] = -1; /* pick */
  /* add self to locklist */
  MPI_Win_lock(MPI_LOCK_EXCLUSIVE, homerank, 0, lockwin);
  MPI_Put(&val, 4, MPI_INT, homerank, 4*myrank, 4, MPI_INT, lockwin);
  MPI_Get(locklistcopy, 4*(nprocs-1), MPI_INT, homerank, 0, 1,
          locktype1, lockwin);
  MPI_Win_unlock(homerank, lockwin);
  /* check to see if lock is already held */
  cprocs_i = 0;
  for (i=0; i <(nprocs-1); i++)
    if ((flag == 1) && (byte range conflicts with Pi‘s request)) {
      conflict = 1; picklist[cprocs_i] = Pi; cprocs_i++;
    }
  if (conflict == 1) {
    for (j=0; j < cprocs_i; j++) {
      MPI_Win_lock(MPI_LOCK_EXCLUSIVE, homerank, 0, lockwin);
      val[0] = 0; val[3] = picklist[j];
      /* reset flag to 0, indicate pick and pick_counter */
      MPI_Put(&val, 4, MPI_INT, homerank, 4*myrank, 4, MPI_INT, lockwin);
      MPI_Get(locklistcopy, 4*(nprocs-1), MPI_INT, homerank, 0, 1,
              locktype1, lockwin);
      MPI_Win_unlock(homerank, lockwin);
      if (picklist[j] has released the lock || detect_deadlock())
        /* repeat for the next process in picklist */
      else {
        /* wait for notification from picklist[j], then retry the lock */
        MPI_Recv(NULL, 0, MPI_BYTE, picklist[j], WAKEUP, comm,
                 MPI_STATUS_IGNORE);
        break;
      }
    }
    Lock_acquire(start, end); /* Iteration shown as tail recursion */
  }
  /* lock is acquired here */
}
```

Fig. 9. Pseudocode for the deadlock-free byte-range locking algorithm (Alternative 2).

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

Fig. 10. Avoiding circular loops among processes picked to wake up other processes in Alternative 2.

The lock-release algorithm for Alternative 2 is similar to the original except that the releasing process sends a wake-up message only to those processes that have picked it, not to all processes that have a conflicting byte range. This procedure also reduces the number of extra sends (but does not eliminate them altogether). A discussion of these extra sends is provided in Section V-D.

C. Formal Modeling and Verification

Alternative 1 and Alternative 2 were prototyped by using Promela. The entire code is available online [25]. We employ Promela channels to model MPI sockets. While the creation of such Promela models requires some expertise, our experience is that Promela can be easily taught to most engineers. Our Promela models occupy nearly the same number of lines of code and are structured similar to the pseudocode we have presented for the algorithms. Figure 11 shows an excerpt of the `lock_acquire` function in Promela. Comparing with Figure 9, we see that the Promela code fleshes out the pseudocode by adding an iteration across `NUM_PROCS` and essentially carries out the same sequence of actions such as `MPI_Win_lock` and `MPI_Put`. We have built a support library in Promela that models these MPI primitives.

D. Assessment of the Alternative Algorithms

Neither of these alternatives eliminates the extra sends, but, as described in Section IV, an implementation can deal with them by using `MPI_Iprobe`. We model checked these algorithms

```
inline lock_acquire(start, end) {
  .......
      do
      :: 1 ->
         do
           :: cprocs_i == NUM_PROCS ->
                 break;
           :: else ->
                 cprocs[cprocs_i] = -1;
                 cprocs_i++;
         od;
         cprocs_i = 0;

         MPI_Win_lock(my_pid);
         MPI_Put(1, start, end, -1, my_pid);
         MPI_Get(my_pid);
         MPI_Win_unlock(my_pid);

         do
           :: lock_acquire_i == DATA_SIZE -> break;
           :: else ->
                 if
                   :: lock_acquire_i == my_pid ->
                              lock_acquire_i = lock_acquire_i + 4;
                   :: lock_acquire_i != my_pid &&
                      (private_data[my_pid].data[lock_acquire_i] == 1) ->
                           /* someone else has a flag value of 1,
                           check for byte range conflict */

              ...rest of the code omitted...
```

Fig. 11. Excerpts from Promela encoding of `Lock_acquire`.

using SPIN, which helped establish the following formal properties of the algorithms:

- Absence of deadlocks (both Alternatives 1 and 2). (Even if an MPI implementation blocks on a 0-byte send, the extra sends need not cause deadlock because they can be handled by using `MPI_Iprobe`s.)
- Communal progress (that is, 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), whereas Alternative 1 requires a few additional restrictions to rule out a few rare schedules (meaning, the livelock problem discussed in Figure 8) [26].

That said, Alternative 2 considerably reduces these extra sends, as it restricts the number of processes that can wake up a particular process compared with Alternative 1.

We are still seeking algorithms that would avoid the extra sends (and be efficient). We have a few reasons to suspect that finding such algorithms will not be straightforward. For instance, consider the scenario of a process P1 initially picking P2 to wake it up but, finding that it has released the lock (Line 27 of Figure 9), tries to pick the next eligible process. In Figure 9, we do not show the obvious action of P1 first "unpicking" P2 before picking the next eligible process because P1 will need access to the window before it can mark P2 as having been unpicked. This action may actually be scheduled after P2 has again acquired and released the lock and sent a message to P1, thus defeating the act of unpicking P2. Of course, theoretically, correct solutions do exist. One could, for instance, consider implementing atomic storage locations, one location per one-sided communication window, and program, say, Peterson's mutual exclusion algorithm [8] to be the basis for the *entire* byte-range locking protocol. The drastic inefficiency of such overkill solutions would not be tolerated, however. This point underscores the trio of concerns introduced in Section I, namely, that a programmer must ultimately learn to apply model checking to debug the overall correctness, and also balance the issues of resources and performance in arriving at the final solution employed. The success of formal methods applied in the HPC arena may, in the end, depend equally on the success that the research community attains pertaining to the latter two issues, over and above the success attained in the area of model checking.

E. Performance Results

To measure the relative performance of the two algorithms, we wrote three test programs: one in which all processes try to acquire nonconflicting locks (different byte ranges), another in which all processes try to acquire a conflicting lock (same byte range), and a third in which all processes acquire random locks (random byte-range between 0 and 1000). In all tests, each

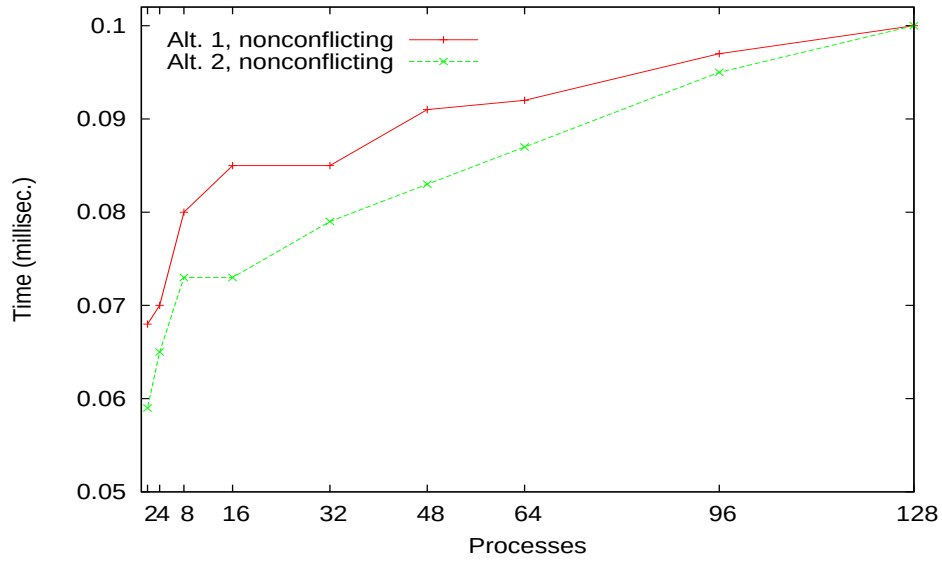

Fig. 12. Average time per process for acquiring and releasing a nonconflicting lock.

process acquires and releases the lock in a loop several times. We measured the time taken by all processes to complete acquiring and releasing all their locks and divided this time by the number of processes times the number of iterations. This measurement gave the average time taken by a single process for acquiring and releasing a single lock. We ran the tests on up to 128 nodes of a Myrinet-connected Linux cluster at Argonne. We used the latest version of MPICH2 (1.0.5) with the Nemesis channel over GM.

Figure 12 shows the results for nonconflicting locks. In this case, there is no contention for the byte-range lock; however, since the target window is located on one process (rank 0) and all the one-sided operations are directed to it, the time taken to service the one-sided operations increases as the number of processes calling them increases. Alternative 1 is slightly slower than Alternative 2 because it always involves two steps: the flag is first set to 2, and if no conflict is detected, it is set to 1. Alternative 2 requires only one step.

Figure 13 shows the results for conflicting and random locks. Even in these two cases, we find that Alternative 2 outperforms Alternative 1. Alternative 1 is hampered by the need for a process to back off for a random amount of time before retrying the lock when it detects that

another process is trying to acquire a lock (flag=2). This random wait is needed to avoid the livelock condition described earlier. We implemented the wait by using the POSIX `nanosleep` function, which delays the execution of the program for at least the time specified. However, a drawback of the way this function is implemented in Linux (and many other operating systems) is that even though the specified time is small, it can take up to 10 ms longer than specified until the process becomes runnable again. This causes the process to wait longer than needed and slows the algorithm. Also, there is no good way to know how long a process must wait before retrying in order to avoid the livelock. Based on experiments, we used a time equal to (myrank $\times$ 500) nanoseconds, where myrank is the rank of the process.

For conflicting locks on a small number of processes (4–16), we find that the time taken by Alternative 1 is substantially higher than Alternative 2. We believe this is because the effect of `nanosleep` taking longer than specified to return is more visible here as wasted time. On larger numbers of processes, that time gets used by some other process trying to acquire the lock and hence does not adversely affect the average.

VI. Conclusions

We have shown how formal verification based on model checking can be used to find actual deadlocks in a published algorithm for distributed byte-range locking. We have also discussed how this technology can help shed light on a number of related issues such as forward progress and the possibility of there being unconsumed messages. We presented and analyzed two algorithms for byte-range locking that avoid the race condition. We also verified their characteristics.

Although concurrency-related errors such as deadlocks and race conditions are hard for humans to detect but easy for a model-checking tool, the use of formal verification for parallel programming is still quite rare. Our experience provides evidence that the parallel-programming community should pay more attention to this promising technology. With the widespread emergence of multicore chips, and the need for concurrent programming to exploit the availability

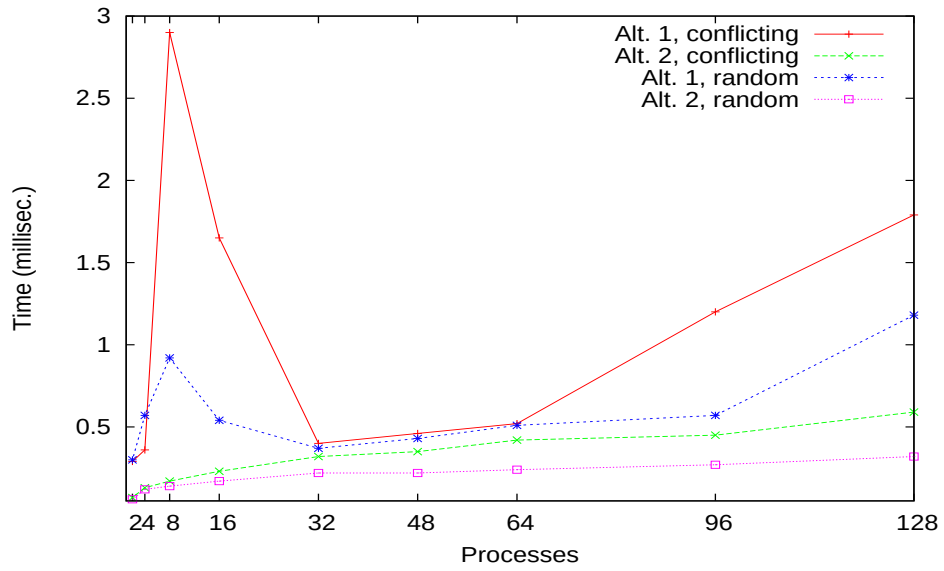

Fig. 13. Average time per process for acquiring and releasing a conflicting lock and a random lock. The spike in the small process range for Alternative 1 is because the effect of the backoff with nanosleep is more visible here as wasted time, whereas on a larger number of processes, that time gets used by some other process trying to acquire the lock.

of multiple CPUs, the need for such technology is all the more important and timely.

One difficulty in model checking is the need to create an accurate model of the program being verified. This step is tedious and error prone. If the model itself is not accurate, the verification will not be accurate. To avoid this problem, we are also working on an in-situ model checker that works directly on the parallel MPI program and avoids the need to create a model [27].

ACKNOWLEDGMENTS

This work was supported by NSF award CNS-0509379, by the Microsoft HPC Institutes program, and by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357.

REFERENCES

[1] E. A. Emerson and E. M. Clarke, “Using branching time temporal logic to synthesize synchronization skeletons,” *Science of Computer Programming*, vol. 2, no. 3, pp. 241–266, 1981.

[2] J. Queille and J. Sifakis, "Specification and verification of concurrent systems in CESAR," in *Fifth International Symposium on Programming*, M. Dezani-Ciancaglini and U. Montanari, Eds. Lecture Notes in Computer Science 137, Springer-Verlag, 1981, pp. 337–351.

[3] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. Cambridge, MA: MIT Press, 1999.

[4] "Symposium on twenty five years of model checking," Nov. 2006, `www.easychair.org/FLoC-06/25MC-preproceedings.pdf`.

[5] "MPICH2," `http://www.mcs.anl.gov/mpi/mpich2`.

[6] R. Thakur, R. Ross, and R. 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*. Lecture Notes in Computer Science 3666, Springer, September 2005, pp. 120–129.

[7] Message Passing Interface Forum, "MPI-2: Extensions to the Message-Passing Interface," July 1997, `http://www.mpi-forum.org/docs/docs.html`.

[8] G. L. Peterson, "Myths about the mutual exclusion problem," *Information Processing Letters*, vol. 12, no. 3, june 1981.

[9] M. Ahamad, R. A. Bazzi, R. John, P. Kohli, and G. Neiger, "The power of processor consistency," in *Symposium on Parallel Algorithms and Architectures*, June 1993, pp. 251–260.

[10] G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*. Boston: Addison-Wesley, 2003.

[11] W. Visser, K. Havelund, G. Brat, and S. Park, "Model checking programs," in *The Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00)*, September 2000.

[12] T. Ball, B. Cook, V. Levin, and S. K. Rajamani, "SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft," in *Proceedings of IFM 04: Integrated Formal Methods*. Springer, April 2004, pp. 1–20.

[13] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. L. Dill, "Cmc: A pragmatic approach to model checking real code," in *Proceedings of the Fifth Symposium on Operating System Design and Implementation*, December 2002.

[14] S. F. Siegel and G. S. Avrunin, "Verification of MPI-based software for scientific computation," in *Proceedings of the 11th International SPIN Workshop on Model Checking Software*. Lecture Notes in Computer Science 2989, Springer, April 2004, pp. 286–303.

[15] S. F. Siegel, "Efficient verification of halting properties for MPI programs with wildcard receives," in *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2005, pp. 413–429.

[16] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke, "Using model checking with symbolic execution to verify

parallel numerical programs," in *Proceedings of the ACM SIGSOFT 2006 International Symposium on Software Testing and Analysis*, July 2006.

[17] S. F. Siegel, "Model checking nonblocking MPI programs," in *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation*, January 2007, pp. 44–58.

[18] R. Palmer, S. Barrus, Y. Yang, G. Gopalakrishnan, and R. M. Kirby, "Gauss: A framework for verifying scientific computing software," in *Workshop on Software Model Checking (SoftMC 2005)*, vol. 144. Electronic Notes on Theoretical Computer Science, Elsevier, 2005, pp. 95–106.

[19] R. Palmer, G. Gopalakrishnan, and R. M. Kirby, "Formal specification and verification using +CAL: An experience report," in *Proceedings of Verify'06 (FLoC 2006)*, 2006.

[20] R. Palmer, M. Delisi, G. Gopalakrishnan, and R. M. Kirby, "An approach to formalization and analysis of message passing libraries," in *Proceedings of 12th International Workshop on Formal Methods for Industry Critical Systems (FMICS)*, July 2007, accepted.

[21] R. Palmer, G. Gopalakrishnan, and R. M. Kirby, "Semantics driven dynamic partial-order reduction of mpi-based parallel programs," in *Parallel and Distributed Systems: Testing and Debugging (PADTAD)*, 2007, accepted.

[22] D. Kranzlmüller, "Event graph analysis for debugging massively parallel programs," Ph.D. dissertation, John Kepler University Linz, Austria, September 2000, `http://www.gup.uni-linz.ac.at/~dk/thesis`.

[23] O. S. Matlin, E. Lusk, and W. McCune, "SPINning parallel systems software," in *Model Checking of Software: 9th International SPIN Workshop*. Lecture Notes in Computer Science 2318, Springer, 2002, pp. 213–220.

[24] W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2: Advanced Features of the Message-Passing Interface*. Cambridge, MA: MIT Press, 1999.

[25] S. Pervez, "Promela encoding of byte range locking written using MPI one-sided operations," 2007, `http://www.cs.utah.edu/formal_verification/benchmarks/byterange-locking-promela-models.tar.gz`.

[26] ——, "Byte-range locks using MPI one-sided communication," University of Utah, School of Computing, Tech. Rep., 2006, `http://www.cs.utah.edu/formal_verification/OnesidedTR1/`.

[27] S. Pervez, G. Gopalakrishnan, R. M. Kirby, R. Palmer, R. Thakur, and W. Gropp, "Practical model checking method for verifying correctness of MPI programs," University of Utah, School of Computing, Tech. Rep. UUCS-07-014, 2007, accepted for Euro PVM/MPI 2007.