SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2010; **40**:23–43
Published online 21 December 2009 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.946

Formal methods applied to high-performance computing software design: a case study of MPI one-sided communication-based locking[‡]

Salman Pervez[1], Ganesh Gopalakrishnan[2,*,†], Robert M. Kirby[2], Rajeev Thakur[3] and William Gropp[4]

[1]*Department of Computer Science, Purdue University, West Lafayette, IN, U.S.A.*
[2]*School of Computing, University of Utah, Salt Lake City, UT 84112, U.S.A.*
[3]*Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, U.S.A.*
[4]*Department of Computer Science, University of Illinois, Urbana, IL 61801, U.S.A.*

SUMMARY

There is a growing need to address the complexity of verifying the numerous concurrent protocols employed in the high-performance computing software. Today's approaches for verification consist of testing detailed implementations of these protocols. Unfortunately, this approach can seldom show the absence of bugs, and often results in serious bugs escaping into the deployed software. An approach called *Model Checking* has been demonstrated to be eminently helpful in debugging these protocols early in the software life cycle by offering the ability to represent and exhaustively analyze simplified formal protocol models. The effectiveness of model checking has yet to be adequately demonstrated in high-performance computing. This paper presents a case study of a concurrent protocol that was thought to be sufficiently well tested, but proved to contain two very non-obvious deadlocks in them. These bugs were automatically detected through model checking. The protocol models in which these bugs were detected were also easy to create. Recent work in our group demonstrates that even this tedium of model creation can be eliminated by employing dynamic source-code-level analysis methods. Our case study comes from the important domain of Message Passing Interface (MPI)-based programming, which is universally employed for simulating and predicting anything from the structural integrity of combustion chambers to the path of hurricanes. We argue that model checking must be taught as well as used widely within HPC, given this and similar success stories. Copyright 2009 John Wiley & Sons, Ltd.

Received 17 June 2008; Revised 11 September 2009; Accepted 18 September 2009

KEY WORDS: concurrent programming; formal verification; model checking; race condition; SPIN; dynamic analysis; high-performance computing (HPC); Message Passing Interface (MPI); one-sided communication

[*] Correspondence to: Ganesh Gopalakrishnan, School of Computing, University of Utah, Salt Lake City, UT 84112, U.S.A.

[†] E-mail: ganesh@cs.utah.edu

[‡] A shorter version of this paper was published in [1].

Contract/grant sponsor: NSF; contract/grant numbers: CNS-0509379, CCF-0811429
Contract/grant sponsor: Office of Advanced Scientific Computing
Contract/grant sponsor: Office of Science
Contract/grant sponsor: U.S. Department of Energy; contract/grant number: DE-FG02-08ER25835

Copyright 2009 John Wiley & Sons, Ltd.

1. INTRODUCTION

Adequately debugging concurrent protocols remains a fundamental challenge facing computing practitioners. Recent studies [2] show that many concurrency bugs can take up to a month of an expert's time to debug. As another example, consider the concurrent software employed in *high-performance computing* (HPC). This software is used for solving critical problems such as predicting the path of hurricanes using supercomputing clusters consisting of multiple (often thousands of) computing nodes. Each node in a cluster simulates portions of the physical space, exchanging data with other computing nodes that simulate adjacent spaces. These communications are almost always carried out using the Message Passing Interface (MPI) library [3], which is considered to be the *lingua franca* of parallel processing. One expects such large-scale critical simulations to yield reliable data (e.g. to avoid evacuating the wrong shorelines) and run without crashes (e.g. so that the predictions arrive before the hurricane itself does).

Unfortunately, the state of the art in software debugging in the HPC arena consists of conventional software testing methods. While testing methods do not produce false alarms (i.e. they report only genuine bugs) they are not designed to provide any guarantees pertaining to the *absence* of bugs. Testing methods are especially ineffective for concurrent systems because a concurrent programmer—even an experienced one—finds it difficult to cultivate intuitions about selecting effective tests. This is because of the vast number of potential interleavings possible even for extremely short concurrent programs. For instance, a program consisting of five processes P_1 to P_5 where each process i carries out five computing steps $a_{i,1}$ to $a_{i,5}$ has over 10 billion potential schedules (interleavings). This number derives from the following formula: for k processes, each doing n steps, there are potentially $(n \cdot k)!/(n!)^k$ interleavings (concurrent schedules). For k approaching one thousand and for n approaching a million (these are on the lower side in practice), the number of schedules are just astronomical! In addition to interleavings, the state space of a typical concurrent program is very large also due to the sheer size of its state elements.

According to Rushby [4] , experience shows that exhaustively analyzing a down-scaled model of a piece of software is often more effective at finding concurrency bugs compared with the non-exhaustive examination (e.g. through testing) of a full-scale model. A simple but powerful idea that is aimed at this goal is known as *Model Checking*. In fact, the first proponents and some of the early developers of this approach won the 2007 ACM Turing Award [5] (highest honor in computing) for model checking. Downscaling is an age old idea: a designer simply creates a scale model, much like an aircraft designer creates 'wind-tunnel flight models'. In our work, we do not offer any specific approach to downscaling, other than the suggestion that a programmer apply his/her best judgements in deciding what to eliminate and what to keep. Luckily, this step is far easier to teach/learn than developing intuitions about which interleavings actually matter.

Model checking [6–8] is widely regarded as the primary technique of choice for debugging a significant number of concurrent protocols. After 25 years of research [9] 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 HPC software about three years ago, we were rather surprised to discover that designers in HPC either did not know about model checking or its potential to help them debug concurrent protocols of the kind they are now developing. After the initial dialog was established between the Argonne National Laboratory (ANL) authors of this paper (primarily interested in HPC, and the principal developers of the MPICH2 MPI library [10]) and the Utah authors of this paper (two of whom have a model checking background, and one is primarily in HPC but with a strong awareness of model checking), the Utah authors were given a modest-looking challenge problem by the ANL authors: see what model checking can do in terms of analyzing a then recently published *byte-range locking* algorithm [11] developed by some ANL researchers (including one of the authors of this paper). The challenge problem held (and still holds) considerable appeal. It uses MPI's shared-memory extension called Remote-Memory Access (RMA) [12] (also known as *one-sided communication*), which allows a process to directly read from or write to another process's memory. Unfortunately,

Copyright 2009 John Wiley & Sons, Ltd.

the weakly synchronized shared-memory behavior of MPI's one-sided operation comes with the associated problems of the shared-memory programming style, namely, the potential for race conditions. At the time the challenge problem was given to us by the ANL, it was known that one-sided communication had proved to be rather slippery territory for the previous MPI users. Last but not the least, the challenge was not a contrived example. It was developed to enable an MPI Input/Output (MPI-IO) implementation to acquire byte-range locks in the absence of POSIX [13] `fnctl()` (performs file control command) file locks, which some high-performance file systems do not support. These details are not important for our purposes: we can summarize the challenge problem as follows. Imagine processes wanting read/write accesses to contiguous ranges of bytes were 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. for Peterson's algorithm [14], it would be 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 [11] had devised a solution involving backoffs and retries. Unlike 'textbook' locking protocols whose behaviors have been studied under weak memory models (e.g. [15]), a part of the appeal of the locking protocol (to be detailed in Section 3) from a model-checking perspective stems from its use of realistic features from a library that is considered the *de facto* standard of distributed programming.

In this setting, 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 [11]) that result in deadlocks. These errors were found relatively easily using model checking. Curiously, neither the authors of the algorithm nor the reviewers of the originally published paper [11] were aware of the race condition, and conventional testing had missed the associated bugs. In this paper, we then go on to discuss two alternative algorithms. These algorithms have been analyzed using model checking, and found to be free of bugs such as deadlocks. However, the algorithms have remaining resource issues (as did the original algorithm) that required addressing on pragmatic grounds. An initial assessment, which we will present in the current work, 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 has to, in the end, employ performance analysis hand-in-hand with model-checking tools. In this regard, our performance assessment of the two alternative protocols on a 128-node cluster reveals that Alternative 2 performs far better than Alternative 1, both under low lock contention as well as high lock contention.

We then detail all our contributions, discuss a few limitations of our current model-checking approach, and briefly discuss our future work in overcoming these limitations. We also discuss the important lesson of having to build techniques and tools that help engineers to 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 survey. Even those efforts directed at the use of finite-state model checking [5, 8] for concurrent and distributed program verification are too vast to survey; applications in telecommunication software design (e.g. [16]), aerospace software (e.g. [17]), device driver design (e.g. [18]), and operating system kernels (e.g. [19]) 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. The earliest use of model checking in this area is by Matlin *et al.* who used the SPIN model checker [16] to verify parts of the MPD process manager used in MPICH2 [20]. Subsequently, Siegel and Avrunin used model checking to verify MPI programs that employ a limited set of two-sided MPI communication primitives [21]. Siegel subsequently

Copyright 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

published several techniques for efficiently analyzing MPI programs [22–24] and covering the use of their MPI-SPIN tool.

Some of the earlier publications of our group in this area pertained to the use of model checking to analyze MPI programs [25, 26], an executable formal semantic specification of MPI [27, 28] and an efficient model-checking algorithm for MPI [29]. 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 have developed several *in situ* model checkers that work directly on the parallel MPI program and hence avoid the need to create verification models. We reported on the first such dynamic model-checking algorithm for MPI in [30]. Techniques to enhance the efficiency of this algorithm were reported in [31]. In recent work, we reported on an approach to model check MPI programs directly using dynamic verification methods [32–35], introducing our algorithm 'ISP' in the process. We also employed our ISP algorithm for detecting the presence of functionally irrelevant barriers in MPI programs [36]. Other groups have approached the formal verification of MPI programs through schedule perturbation techniques [37, 38], data flow analysis [39], and by detecting bug patterns [40].

Any new application area for model checking tends to have its own details that have to be addressed in a domain-specific manner. In this vein, one must carry out numerous case studies that push the existing algorithms beyond their intended roles, thus triggering the discovery of new algorithms. In [35], we report such large case studies recently handled by ISP. In their most recent work, Siegel *et al.* apply model checking to a mature, MPI-based scientific program consisting of approximately 10k lines of code. The program, BlobFlow, implements a high-order vortex method for solving the two-dimensional Navier–Stokes equations. Despite the complexity of the code, they verify properties including freedom from deadlock and the functional equivalence of sequential and parallel versions of the program using the methods they present in [41].

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. It is also an accurate record of one of our earliest case studies that helped us to gain traction in this new and important application area of applying model checking to analyze real-world HPC software. A shorter version of this paper appeared in [1]. As for other tools verifying MPI programs employing one-sided communication, the Marmot tool has been extended to detect incorrect usages of MPI one-sided communication commands [42]. Their work does not consider the concurrency semantics of one-sided communication or how it can impact the correctness of concurrent protocols developed using them.

Note: In this paper, the usage of the word 'correctness' connotes the ability to algorithmically demonstrate the absence of common flaws such as deadlocks and livelocks. In our Promela experiments, we employed what are known as Büchi automata [8] to state the absence of starvation. From a mathematical logic stance, the notion of correctness that we employ corresponds to the ability to check a catalog of temporal logic properties.

Roadmap: The remainder of this paper is organized as follows. We begin with a brief introduction to model checking in Section 2. We describe the byte-range locking algorithm and how we model checked it in Sections 3 and 4. In Section 5, we present two alternative 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 7, we conclude with a discussion of future work.

2. 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 the desired properties. A model can be anything from a simple finite-state machine modeling the concurrent system to the 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* [8] bugs), and starvation scenarios (*liveness* [8] 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 [16], 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 to establish the 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 of, for example, billions of distinct states). A model checker that has received wide attention among the computer science community is SPIN [16]. 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.

3. THE BYTE-RANGE LOCKING ALGORITHM

Often, processes must acquire exclusive access to a range of bytes, such as a portion of a file. As mentioned on Page 2, in [11], 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 RMA) [12]. 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.

3.1. Overview of MPI

MPI evolved in the early 1990s, borrowing ideas from APIs of the previous era. MPI programs can be run with a specific number of processes (also called *ranks*). Each process can query and determine how many ranks there are, and also determine its own rank in the population. Typically, this is done early in the code; thereafter the computation of the MPI processes branches out into disjoint code segments based on process ranks. The purpose of such branching in MPI programs is to allow each MPI process to work on its own region of data (hence MPI programs are termed 'SPMD' or single-program multiple-data). Processes exchange computational results pertaining to their parts of the data. High-performance MPI libraries have highly optimized versions of *collective* MPI functions such as MPI_Barrier (for barrier synchronization), MPI_Bcast (for broadcasting data), and MPI_Reduce (for reductions such as global summation of the individual data). Whenever they apply, these collective functions are highly recommended for use in lieu of point-to-point operations.

MPI is an API that is designed to cater to a wide variety of programmers, cluster machines, and applications. As a result, MPI has been equipped with over 300 functions. Most users employ under two dozens of these calls; however, it tends to be a different subset for each application/cluster/user. Just referring to sends and receives, MPI has at least a dozen variants all the way from rendezvous style sends/receives (the sender/receiver wait for each other) to ready sends (the receive must be posted to avoid failure of sends). In addition, MPI sends and receives can specify how messages must be matched (based on process ranks and/or tags), and whether the receive is 'wildcard' (`MPI_ANY_SOURCE`) that allows an eligible send from any MPI process to match.

Copyright 2009 John Wiley & Sons, Ltd.

MPI was designed without any shared-memory features or threading. Modern MPI implementations support different levels of threading and thread safety from ‘none’ to full thread safety.

3.2. Overview of MPI’s one-sided communication

RMA was added as a part of MPI-2 [3] to provide a put/get programming model. Earlier programming models that supported these one-sided operations include the Bulk Synchronous Programming (BSP) model [43] and the SHMEM model [44] developed by Cray. Unlike the MPI 2-sided operations (regular MPI message passing commands), MPI-2 RMA allows one process to directly access the memory of another (in the same MPI job) without the tag matching and other overheads associated with the two-sided (or send/receive) operations. Because RMA operations are one-sided, they allow for more weakly synchronized algorithms, and are particularly appropriate when processes in a parallel program need to share data in dynamic ways.

MPI-2 added one-sided communication functions so as to offer a different programming model from the traditional 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* (write) and *get* (read) operations. Prior to invoking a one-sided communication operation, a process must specify the memory region that it wishes to allow other processes to directly access. This memory region is called a *window* and is specified via the *collective function* `MPI_Win_create`. An MPI collective function is one that must be called from every process in order to accomplish its desired effect in each of the processes. In this case, the `MPI_Win_create` collective call allows each process to perceive the memory effects of the put/get operations carried out by all processes.

Once `MPI_Win_create` has been called, the one-sided communication itself is achieved primarily through the `MPI_Put` and `MPI_Get` operations. (There are also additional one-sided communication operations, such as `MPI_Accumulate`, that we do not consider in this paper.) `MPI_Put` writes the data resident at the specified location of the local memory into the target location of the remote memory. `MPI_Get` reads the data resident at the specified location of the remote memory into the target location of the local memory.

Both `MPI_Put` and `MPI_Get` are non-blocking: 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 different synchronization methods. For simplicity, we consider only one of them, namely lock–unlock, which is the one used in the byte-range locking algorithm.

We explain the basic ideas of the lock–unlock method with the help of Figure 1 where Process 1 is the *target* for the `MPI_Win_lock` actions executed by Processes 0 and 2. In the lock–unlock synchronization method, the originating process calls `MPI_Win_lock` specifying the kind of lock requested (`excl`, standing for ‘exclusive’ in Figure 1) and the target (`1` in the figure). `MPI_Win_lock` is not required to block until the lock is acquired (the only exception to this rule is when the originating process and the target process are one and the same). After issuing the one-sided operations, the originating process calls `MPI_Win_unlock`. When this call returns, the one-sided operations are guaranteed to have been completed at the origin as well as at the target. The target process does not make any synchronization call in our illustration.

```
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)
```

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

Copyright 2009 John Wiley & Sons, Ltd.

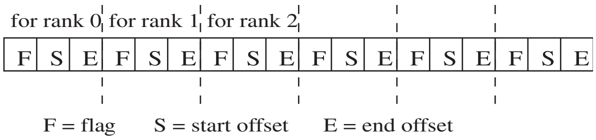

Figure 2. Window layout for the original byte-range locking algorithm.

Note that MPI puts and gets are non-blocking 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 [45]. 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.

3.3. The algorithm

Below we describe the byte-range locking algorithm of [11] together with the snippets of the code for acquiring and releasing a lock.

3.3.1. 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.

3.3.2. Acquiring the lock. The process wanting to acquire a lock calls `MPI_Win_lock` with the `lock_type` as `MPI_LOCK_EXCLUSIVE` (shown `excl` in Figure 1), followed by an `MPI_Put`, an `MPI_Get`, and then `MPI_Win_unlock` as shown in Figure 3. The usage of `MPI_LOCK_EXCLUSIVE` guarantees that the final effect on the window's state will be as if the processes accessed the target window in an exclusive (non-overlapping) manner. (The alternative would be to use `MPI_LOCK_SHARED`, which gives the effect of accessing the window in an overlapped manner.) The exact arguments of `MPI_Put` are, from left to right, as follows: `&val` is the source address, `3` is the number of `MPI_INT`s to be put, `homerank` is the target process rank, `3*(myrank)` specifies the target displacement for subsequent `MPI_INT`s, `3` is the target count, `MPI_INT` is the target datatype, and `lockwin` is the window being written into. The details of the remaining MPI operations are similar, and may be gathered from [3].

Following the `MPI_Win_lock` call, the process uses `MPI_Put` to set 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. Thereafter, using `MPI_Get`, the process 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. After `MPI_Win_unlock` returns, the process goes through the list of values returned by `MPI_Get`. Now, with respect to all other processes, the process first checks whether the flag is 1 and, if so, 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 (line 24 of Figure 3), the process resets its flag in the window to 0 by doing an `MPI_Win_lock`–`MPI_Put`–`MPI_Win_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_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 there is no conflict, we acquired the lock; common case */
        if (conflict == 0) {
          break; // lock is acquired.
        }
        /* Otherwise reset flag to 0, wait for notification, then retry 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 */
    }
}
```

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

lock; see the lock–release algorithm below). After receiving the message, it tries again to acquire the lock by using the same `Lock_acquire` algorithm.

3.3.3. 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 in this comparison. If there is a conflict with the byte-range set by another process—meaning that the process is waiting to acquire a conflicting lock—it sends a zero-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 zero-byte messages to all other processes waiting for a lock that conflicts with its own, the process returns.

4. MODEL CHECKING THE BYTE-RANGE LOCKING ALGORITHM

To model the byte-range locking 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 [12]. 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

Copyright 2009 John Wiley & Sons, Ltd.

```
Lock_release(int start, int end)
{
    val[0] = 0; val[1] = -1; val[2] = -1;

    /* set start and end offsets to -1, flag to 0, and get everyone else's status */
    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, locktype2,
            lockwin);
    MPI_Win_unlock(homerank, lockwin);

    /* check if anyone is waiting for a conflicting lock. If so, send them a
       0-byte message, in response to which they will retry the lock. For
       fairness, we start with the rank after ours and look in order. */

    i = myrank;  /* ranks are off by 1 because of the derived datatype */
    while (i < (nprocs - 1)) {
        /* the flag doesn't matter here. check only the byte ranges */
        if (byte ranges conflict) MPI_Send(NULL, 0, MPI_BYTE, i+1, WAKEUP, comm);
        i++;
    }
    i = 0;
    while (i < myrank) {
        if (byte ranges conflict) MPI_Send(NULL, 0, MPI_BYTE, i, WAKEUP, comm);
        i++;
    }
}
```

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

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.

4.1. Concise overview of Promela

Promela is a language for *formally modeling* concurrent protocols with a view to model check them. Promela strikes a good balance between expressiveness (that assists in writing clear and intuitive protocol descriptions) and facilitation of efficient model checking (by not including constructs that are expensive to handle during model checking). Promela descriptions tend to have a much clearer semantics than the corresponding programming constructs: for instance, the atomicity of a Promela construct such as `x++` itself is not in doubt. In addition, the Promela compiler avoids making optimizations that alter the semantics of the input description (unlike traditional compilers that may perform many code rearrangements subject only to the constraint of preserving data dependencies).

A Promela file consists of a collection of process descriptions that together define the behavior of interest. The general use of Promela is to describe non-deterministic finite-state machines. For specification writers, the use of non-determinism to abstract away from implementation details is a powerful tool. In Promela, it is possible to recursively spawn processes and thereby create infinite-state descriptions. Most Promela descriptions are, however, finite-state descriptions in the sense that all their executions consist of moves between states belonging to a finite set of reachable states.

The language Promela goes hand-in-hand with the model checker SPIN. The SPIN model checker accepts a Promela description, viewing it as a non-deterministic Büchi automaton (a non-deterministic finite-state automaton whose infinite behaviors are of interest). When using SPIN, one can also specify temporal logic properties, either in Promela as a `never` automaton or in

```
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;
}
```

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

linear time temporal logic (LTL). The `xspin` interface of SPIN allows LTL properties to be automatically translated into `never` automata and included in the Promela description being verified. An example of an LTL property that a user may wish to establish is 'Henceforth, whenever *a* happens, *b* should never be found to happen'. Such a property corresponds to all behavioral traces in which after the occurrence of an event *a*, an event *b* is never found to happen.

For the purposes of this paper, one may view model checking as the process of verifying that *all* executions of a finite-state machine satisfy a safety assertion (e.g. an `assert` statement) or a liveness assertion (e.g. 'after every occurrence of *a*, there will eventually be an occurrence of *b*'). In Figure 5, we describe the behavior of `MPI_Put` as follows. `MPI_Put` is an inlined macro. It declares one variable `i` initialized to `0`. It then offers a non-deterministic selection (indicated by the separator `::`) within an infinite loop `do..od`. If one of the conditions written between the `::` and `->` (called the *guard*) evaluates to true, the ensuing behavior (described after the `->`) will be considered by SPIN as being one legitimate execution to be verified. The `else` clause is invoked when none of the guards is true. The `break` construct breaks out of the immediately enclosing block. For details, the reader may refer to [16].

4.2. Promela modeling of the locking protocol

As an example, our Promela implementation of `MPI_Put` is shown in Figure 5. 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, 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.) The complete Promela code used in our model checking can be found online [46].

When we model checked our model with three processes, our model checker, SPIN [16], 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_Sends` 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 implementers to decide whether to block on an `MPI_Send` call. In practice, a zero-byte send will rarely block. Nonetheless, an implementation of

Copyright 2009 John Wiley & Sons, Ltd.

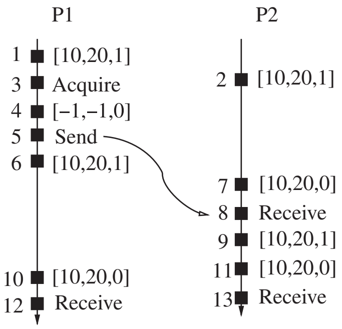

Figure 6. A deadlock scenario found through model checking.

the byte-range locking algorithm can address this problem by periodically calling `MPI_Iprobe` and matching any unexpected messages with `MPI_Recvs`.

We then modeled 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 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; hence 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 reacquire the lock (9). P2 reads P1's current flag value as 1; hence 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 neither detected by the authors of the original algorithm during their design and testing nor by the reviewers of the paper describing the algorithm [11]. Of course, after the model checker pointed out the problem, it appeared obvious and could be reproduced in practice. This is again an example of how easy it is for humans to miss the potential race conditions and the deadlock scenarios, whereas a model-checking tool can easily catch them.

5. 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.

5.1. 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, whereas 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. Hence, it resets its flag value to '0' and blocks on an `MPI_Recv`. If no such process

Copyright 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

```
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 */
}}
```

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

(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 considers the lock acquired. (The algorithm is described recursively in Figure 7 only for convenience. It is implemented and modeled as an iteration.)

5.1.1. 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

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

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

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 in one particular situation 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 in the state of trying to acquire the lock both block in order so 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 backoff 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 5.5 show the possibility of these livelocks happening in practice. The possibility of livelock can be reduced by having each process backoff for a random amount of time before retrying, thereby avoiding the likelihood of the same sequence of events occurring each time. Alternative 2 presented below eliminates the possibility of livelock.

5.2. Alternative 2

This approach 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: (i) Has the process selected to wake it up already released the lock? and (ii) 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.

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 is shown in Figure 12. 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 5.4.

```
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 */
}
```

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

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

Figure 10. Avoiding circular loops among processes selected to awaken in Alternative 2.

5.3. Formal modeling and verification

Both alternative algorithms were prototyped by using Promela. The entire code is available online [46]. We employ Promela channels to model MPI communication. While the creation of such Promela models requires some expertise, accumulated evidence (described at the homepage of the SPIN tool [47]) shows that Promela can be easily taught to engineers. Our Promela models occupy nearly the same number of lines of code and are structured similar to the pseudocode we

```
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...
```

Figure 11. Excerpts from Promela pseudo-code encoding of Lock_acquire.

```
Lock_release (int start, int end) {
  val[0] = 0; val[1] = -1; val[2] = -1; val[3] = -1;
  /* set start and end offsets to -1, flag to 0,
  pick to -1 and get everyone else's status */
  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);
  i = 0; /* ranks are off by 1 because of the derived datatype, adjust i accordingly*/
  while (i < (nprocs - 1)) {
    /* the byte ranges don't matter here, just check the pick */
    if (locklistcopy[4 * i + 3] == myrank)
      MPI_Send (NULL, 0, MPI_BYTE, i, WAKEUP, comm);
    i++;
  }
}
```

Figure 12. Excerpts from Promela pseudo-code encoding of Lock_release.

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. In Section 6, we discuss many related issues, including the growing success of our direct dynamic verification approach for MPI programs [32–35].

5.4. *Assessment of the alternative algorithms*

Neither of these alternatives eliminates the extra sends, but, as described in Section 3, an implementation can deal with them by using MPI_Iprobe. We model checked these algorithms using

Copyright 2009 John Wiley & Sons, Ltd.

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

- Absence of deadlocks (both alternatives). (Even if an MPI implementation blocks on a zero-byte send, the extra sends need not cause deadlock because they can be handled by using `MPI_Iprobes`.)
- 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) [48].

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). There are 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 anyway sent a message toward 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 [14] to be the basis for the *entire* byte-range locking protocol! The drastic inefficiency of such overkill solutions will not be tolerated. This point underscores the trio of concerns introduced in Section 1, namely that a programmer must, in the end, 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, equally depend 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.

5.5. *Performance results*

To measure the relative performance of the two algorithms, we wrote three test programs: one in which all processes try to acquire non-conflicting 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 the tests, each 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 using MPICH2 over GM.

Figure 13 shows the results for non-conflicting 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 14 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 although the specified time is small, it can take up to 10 ms longer than specified until the process becomes

Copyright 2009 John Wiley & Sons, Ltd.

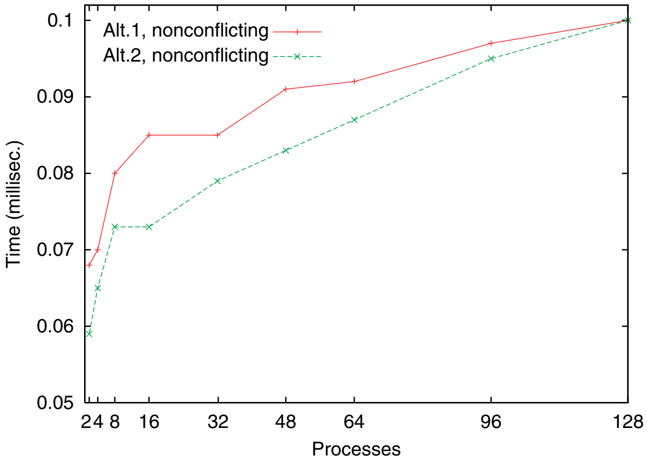

Figure 13. Average time per process for acquiring and releasing a non-conflicting lock.

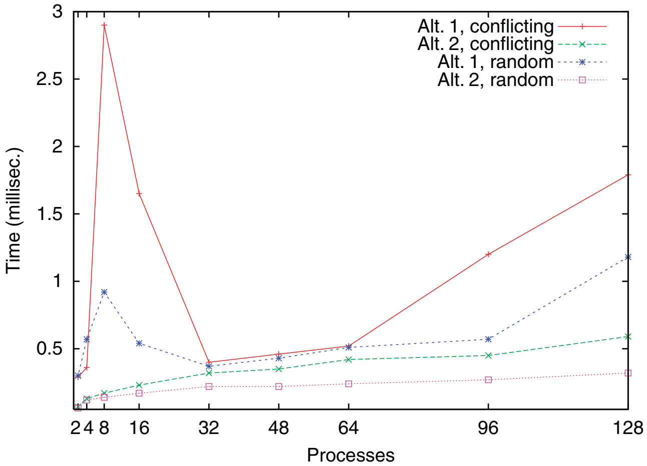

Figure 14. Average time per process for acquiring and releasing a conflicting lock and a random lock. The spike in the small process range for Alternative 1 is because the effect of the backoff with nanosleep is more visible here as wasted time, whereas on larger number of processes, that time gets used by some other process trying to acquire the lock.

runnable again. This causes the process to wait longer than needed and slows the algorithm. In addition, 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 × 500) ns, 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.

6. DISCUSSIONS

We now address several important questions concerning the viability of applying model checking for HPC software.

Copyright 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

Effort required to model applications: The approach presented in this paper is ideally suited for modeling protocols at a high level. It is ill suited for modeling large-scale applications after the applications have been developed in sufficient detail. This is because of the extreme tedium of understanding and modeling an already built application.

Effort to model MPI library functions: This paper's attempt of modeling MPI's one-sided functions in Promela is tractable, given the relatively small number of these functions. However, for the full MPI library, given that MPI-2 has over 300 functions, the effort becomes tedious very quickly. MPI-SPIN [49] includes a description of many MPI functions. In [27], we wrote a description for many MPI functions in TLA+ [50]; this was expanded into a description of nearly 150 MPI functions in [28]. The ISP tool [32, 33] bypasses the need to model MPI functions by relying on the native semantics of an actual MPI library implementation. This of course runs the risk of relying on the semantics of an actual MPI library implementation. In short, there appears to be no 'best approach' when it comes to modeling the semantics of complex APIs such as MPI. Each verification methodology has to adopt its own approach, and the use of multiple models is also justified, provided one can reconcile the differences between these models.

Expected scalability of the proposed techniques: Model checking relies on the maxim that many bugs are best caught on a much smaller scale than the typically deployed applications. In contrast, many present-day practices are reluctant to approach software development by going through a succession of progressively refined models. The approach in this paper is best suited for verification after downscaling, and is not suited for verification of fully configured system deployments with a large number of processes and runtime memory.

Techniques for high-end HPC applications: This point is related to the previous one: high-end HPC applications suffer from many bugs that are outside the realm of today's formal analysis methods. Bugs could arise due to incorrectly written applications, incorrectly behaving MPI libraries, or resource consumptions in excess of the (often unstated) usage constraints. Given that testing is the only approach for debugging, localizing faults becomes an extremely difficult proposition. Therefore, development frameworks that provide multiple intuitive views as well as source code navigation facilities are important even for formal tools. Examples include the GUIs of the CHESS tool [51] or the ISP tool [52].

In conclusion, the methods presented in this paper are about as realistic as any other alternative available in today's cornucopia of available debugging methods. They underscore the importance of finding bugs at the small scale through formal analysis. Future research can support effective debugging by addressing the tedium of modeling as well as maintaining consistency between various models.

7. 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 alternative algorithms for byte-range locking that avoid the race condition. We also analyzed the characteristics of these protocols including their correctness guarantees, and also their performance.

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 in its infancy. Some of the recent encouraging developments in this area include the CHESS tool of Microsoft Research [51, 53], the MODIST tool for verifying distributed systems by employing model-checking techniques in a lightweight manner [54], the Utah authors' group's tool Inspect for verifying C/Pthreads programs, and the MPI-SPIN, [49] which uses model checking for concurrency analysis and symbolic execution for verifying computations. With the widespread emergence of multicore chips and the need for concurrent programming to exploit their availability, the need for formal methods is all the more important and timely. The availability of a good model checker enables a designer to confidently embark on designing aggressively optimized protocols. Thus, formal tools can not only boost confidence but also lead to the development of efficient protocols.

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. The approach of modeling a protocol in Promela and verifying the model is worthwhile for short intricate protocols such as locking protocols. However, if the model itself is not accurate, verification results can be misleading. It may still be possible to build highly abstract models in Promela, analyze the resulting models cheaply using SPIN, find bugs, and thereby save the tedium of having to build a finely engineered but incorrect protocol!

However, for anything but the simplest of protocols, a designer may in fact wish to directly model check the protocol using native languages and libraries. It is also inevitable that designers build real protocol implementations during design space exploration, so that the performance characteristics of the protocols are well understood. We believe that this dynamic formal verification approach—as embodied in our tools ISP (discussed in Sections 6 and 1) and Inspect, as well as external tools such as CHESS and MODIST—holds considerable promise going forward.

There are still harder concurrency verification problems that are yet to be addressed by us or by others. This prominently includes the topic of parameterized verification: prove for an arbitrary number of processes that a given protocol is correct. All our formal verification exercises targeted a specific number (three or around that number) of processes. In general, parameterized verification is algorithmically unsolvable [8] as one can show that solving parameterized verification is tantamount to algorithmically solving the halting problem. Despite this, parameterized verification may be solvable in restricted cases through the use of symbolic reasoning methods. In some cases, parameterized verification can be solved through a technique known as counterexample guided abstraction/refinement. One such successful attempt by Chou *et al.* [55] is of considerable practical interest in the area of directory-based multiprocessor cache coherency protocol modeling and verification. This work achieves parameterized verification purely through interactive model checking of successively (manually) refined protocols. Whether such successes can occur in the realm of MPI programming and verification remains to be seen.

ACKNOWLEDGEMENTS

This work was supported by NSF awards CNS-0509379, CCF-0811429, 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. Department of Energy, under Contract DE-FG02-08ER25835. We also thank the reviewers for their diligent reading of our paper and their valuable comments.

REFERENCES

1. Pervez S, Gopalakrishnan G, Kirby RM, Thakur R, Gropp W. Formal verification of programs that use MPI one-sided communication. *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (*EuroPVM/MPI*), Bonn, Germany (*Lecture Notes in Computer Science*, vol. 4192), 2006; 30–39. Outstanding Paper.
2. Godefroid P, Nagappan N. Concurrency at Microsoft: An exploratory survey. *EC2*: *Workshop on Exploiting Concurrency Efficiently and Correctly*. Available at: http://www.cs.utah.edu/ec2/papers/ec2-pp1.pdf [15 December 2009].
3. MPI Forum. MPI: A message-passing interface standard, version 2.1, June 2008. Available at: http://www.mpi-forum.org [15 December 2009].
4. Rushby J. Model checking and other ways of automating formal methods, panel on model checking for concurrent programs software quality week, 1995. Available at: http://www.csl.sri.com/reports/postscript/sqw95.ps.gz [15 December 2009].
5. Clarke E, Emerson EA, Sifakis J. The 2007 ACM Turing Award given to the original proponents and developers of Model Checking for their pioneering work that began in 1981. Available at: http://awards.acm.org/homepage.cfm?awd=140 [15 December 2009].
6. Emerson EA, Clarke EM. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming* 1981; **2**(3):241–266.

Copyright 2009 John Wiley & Sons, Ltd.

7. Queille J, Sifakis J. Specification and verification of concurrent systems in CESAR. *Fifth International Symposium on Programming*, Dezani-Ciancaglini M, Montanari U (eds.). (*Lecture Notes in Computer Science*, vol. 137). Springer: Berlin, 1981; 337–351.
8. Clarke EM, Grumberg O, Peled D. *Model Checking*. MIT Press: Cambridge, MA, 1999.
9. Symposium on twenty five years of model checking, November 2006. Available at: http://www.easychair.org/FLoC-06/25MC-preproceedings.pdf [15 December 2009].
10. The MPICH2 site. Available at: http://www-unix.mcs.anl.gov/mpi/mpich2/ [15 December 2009].
11. Thakur R, Ross R, Latham R. Implementing byte-range locks using MPI one-sided communication. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, *12th European PVM/MPI Users' Group Meeting* (*Lecture Notes in Computer Science*, vol. 3666). Springer: Berlin, September 2005; 120–129.
12. Message Passing Interface Forum. MPI2: A Message Passing Interface standard. *International Journal of High Performance Computing Applications* 1998; **12**(1–2):1–299.
13. Portable operating system interface (POSIX). Available at: http://standards.ieee.org/regauth/posix/ [15 December 2009].
14. Peterson GL. Myths about the mutual exclusion problem. *Information Processing Letters* 1981; **12**(3):115–116.
15. Ahamad M, Bazzi RA, John R, Kohli P, Neiger G. The power of processor consistency (extended abstract). *Symposium on Parallel Algorithms and Architectures*, Velen, Germany, June 1993; 251–260.
16. Holzmann GJ. *The SPIN Model Checker*: *Primer and Reference Manual*. Addison-Wesley: Boston, MA, 2003.
17. Visser W, Havelund K, Brat G, Park S. Model checking programs. *The Fifteenth IEEE International Conference on Automated Software Engineering* (*ASE'00*), Grenoble, France, September 2000; 3–12.
18. Ball T, Cook B, Levin V, Rajamani SK. SLAM and static driver verifier: technology transfer of formal methods inside Microsoft. *Proceedings of IFM 04*: *Integrated Formal Methods*. Springer: Berlin, April 2004; 1–20.
19. Musuvathi M, Park D, Chou A, Engler D, Dill DL. CMC: A pragmatic approach to model checking real code. *Proceedings of the Fifth Symposium on Operating System Design and Implementation*, Boston, MA, U.S.A., December 2002.
20. Matlin OS, Lusk E, McCune W. SPINning parallel systems software. *Model Checking of Software*: *Ninth International SPIN Workshop* (*Lecture Notes in Computer Science*, vol. 2318). Springer: Berlin, 2002; 213–220.
21. Siegel SF, Avrunin GS. Verification of MPI-based software for scientific computation. *Proceedings of the 11th International SPIN Workshop on Model Checking Software* (*Lecture Notes in Computer Science*, vol. 2989). Springer: Berlin, April 2004; 286–303.
22. Siegel SF. Efficient verification of halting properties for MPI programs with wildcard receives. *Proceedings of the Sixth International Conference on Verification*, *Model Checking*, *and Abstract Interpretation* (*VMCAI*), Paris, France, 2005; 413–429.
23. Siegel SF, Mironova A, Avrunin GS, Clarke LA. Using model checking with symbolic execution to verify parallel numerical programs. *Proceedings of the ACM SIGSOFT 2006 International Symposium on Software Testing and Analysis*, Portland, ME, U.S.A., July 2006; 157–168.
24. Siegel SF. Model checking nonblocking MPI programs. *Proceedings of the Eighth International Conference on Verification*, *Model Checking*, *and Abstract Interpretation*, Nice, France, January 2007; 44–58.
25. Palmer R, Barrus S, Yang Y, Gopalakrishnan G, Kirby RM. Gauss: A framework for verifying scientific computing software. *Workshop on Software Model Checking* (*Electronic Notes on Theoretical Computer Science* (*ENTCS*), vol. 953), Edinburgh, Scotland, 2005. Available at: http://www.cs.utah.edu/fv [15 December 2009].
26. Palmer R, Gopalakrishnan G, Kirby RM. Formal specification and verification using +CAL: An experience report. *Proceedings of Verify'06* (*FLoC 2006*), Seattle, WA, U.S.A., 2006. Available at: http://www.cs.utah.edu/fv/publications/conferences/pdf/verify06.pdf [15 December 2009].
27. Palmer R, Delisi M, Gopalakrishnan G, Kirby RM. An approach to formalization and analysis of message passing libraries. *Formal Methods for Industry Critical Systems* (*FMICS 2007*), Leue S, Merino P (eds.). (*Lecture Notes in Computer Science*, vol. 4916). Springer: Berlin, 2008; 164–181. Best Paper Award.
28. Li G, DeLisi M, Gopalakrishnan G, Kirby RM. Formal specification of the MPI-2.0 standard in tla+. *Principles and Practices of Parallel Programming* (*PPoPP*), Salt Lake City, UT, U.S.A., 2008; 283–284.
29. Palmer R, Gopalakrishnan G, Kirby RM. Semantics driven dynamic partial-order reduction of MPI-based parallel programs. *Parallel and Distributed Systems*: *Testing and Debugging* (*PADTAD*), London, U.K., 2007; 43–53.
30. Pervez S, Palmer R, Gopalakrishnan G, Kirby RM, Thakur R, Gropp W. Practical model checking method for verifying correctness of MPI programs. *EuroPVM/MPI* (*Lecture Notes in Computer Science*, vol. 4757). Springer: Berlin, 2007; 344–353.
31. Vakkalanka S, Sharma SV, Gopalakrishnan G, Kirby RM. ISP: A tool for model checking MPI programs. *Principles and Practices of Parallel Programming* (*PPoPP*), Salt Lake City, UT, U.S.A., 2008; 285–286.
32. Vakkalanka S, Gopalakrishnan G, Kirby RM. Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. *Computer Aided Verification* (*CAV 2008*), vol. 5123/2008, Princeton, NJ, U.S.A., 2008; 66–79.
33. Vakkalanka S, DeLisi M, Gopalakrishnan G, Kirby RM, Thakur R, Gropp W. Implementing efficient dynamic formal verification methods for MPI programs. *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (*EuroPVM/MPI*), vol. 5205/2008, Dublin, Ireland, 2008; 248–256.
34. Vakkalanka S, DeLisi M, Gopalakrishnan G, Kirby RM. Scheduling considerations for building dynamic verification tools for MPI. *Parallel and Distributed Systems—Testing and Debugging* (*PADTAD-VI*), Seattle, WA, U.S.A., July 2008. Available at: http://www.cs.utah.edu/fv/ [15 December 2009].

35. Vo A, Vakkalanka S, DeLisi M, Gopalakrishnan G, Kirby RM, Thakur R. Formal verification of practical MPI programs. *Principles and Practices of Parallel Programming* (*PPoPP*), Dublin, Ireland, 2009; 261–269.
36. Sharma S, Vakkalanka S, Gopalakrishnan G, Kirby RM, Thakur R, Gropp W. A formal approach to detect functionally irrelevant barriers in MPI programs. *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (*EuroPVM/MPI*), Dublin, Ireland, 2008; 265–273.
37. Vuduc R, Schulz M, Quinlan D, de Supinski B, Saebjornsen A. Improved distributed memory applications testing by message perturbation. *Parallel and Distributed Systems*: *Testing and Debugging* (*PADTAD-IV*), Portland, ME, U.S.A., 2006. Available at: http://vuduc.org/research/jitterbug/index.html [15 December 2009].
38. Krammer B, Bidmon K, Müller MS, Resch MM. Marmot: An MPI analysis and checking tool. *Parallel Computing 2003*, Dresden, Germany, September 2003; 493–500.
39. Strout MM, Kreaseck B, Hovland PD. Data-flow analysis for MPI programs. *International Conference on Parallel Programming* (*ICPP*), Columbus, OH, U.S.A., 2006; 175–184.
40. Quinlan D, Vuduc R, Misherghi G. Techniques for the specification of bug patterns. *Parallel and Distributed Systems: Testing and Debugging* (*PADTAD*), London, U.K., 2007; 27–35.
41. Siegel SF, Mironova A, Avrunin GS, Clarke LA. Combining symbolic execution with model checking to verify parallel numerical programs. *ACM Transactions on Software Engineering and Methodology* 2008; **17**(2):1–34, Article 10.
42. Krammer B, Resch MM. Correctness checking of MPI one-sided communication using marmot. *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (*EuroPVM/MPI*) (*Lecture Notes in Computer Science*, vol. 4192). Springer: Berlin, 2006; 105–114.
43. Hill JMD, Skillicorn DB. Lessons learned from implementing BSP. *Future Generation Computer Systems* 1998; **13**(4–5):327–335.
44. SHMEM API for parallel programming. Available at: http://www.shmem.org [15 December 2009].
45. Gropp W, Lusk E, Thakur R. *Using MPI-2*: *Advanced Features of the Message-Passing Interface*. MIT Press: Cambridge, MA, 1999.
46. Pervez S. Promela encoding of byte range locking written using MPI one-sided operations, 2007. Available at: http://www.cs.utah.edu/formal_verification/benchmarks/byterange-locking-promela-models.tar.gz [15 December 2009].
47. On-the-fly LTL model checking with SPIN. Available at: http://www.spinroot.com [15 December 2009].
48. Pervez S. Byte-range locks using MPI one-sided communication. *Technical Report*, University of Utah, School of Computing, 2006. Available at: http://www.cs.utah.edu/formal_verification/OnesidedTR1/ [15 December 2009].
49. Siegel S. The MPI-SPIN tool webpage. Available at: http://vsl.cis.udel.edu/mpi-spin/ [15 December 2009].
50. Lamport L. Available at: research.microsoft.com/users/lamport/tla/tla.html [15 December 2009].
51. Chess: Find and reproduce Heisenbugs in concurrent programs. One may download the chess tool for evaluation from this url: http://research.microsoft.com/en-us/projects/chess/ [15 December 2009].
52. Release of ISP: Tool for dynamic verification of MPI programs, School of Computing, University of Utah. Available at: http://www.cs.utah.edu/formal_verification/ISP-release [15 December 2009].
53. Musuvathi M, Qadeer S. Iterative context bounding for systematic testing of multithreaded programs. *PLDI '07*: *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*. ACM: New York, NY, U.S.A., 2007; 446–455.
54. Yang J, Chen T, Wu M, Xu Z, Liu X, Lin H, Yang M, Zhang FL, Zhou L. Modist: Transparent model checking of unmodified distributed systems. *Sixth Usenix Symposium on Networked Systems Design and Implementation* (*NSDI*), Boston, MA, U.S.A., April 2009; 213–228.
55. Chou CT, Mannava PK, Park S. A simple method for parameterized verification of cache coherence protocols. *Formal Methods in Computer Aided Design*, Austin, TX, U.S.A., 2004; 382–398.

Copyright 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/spe