Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 193 (2007) 19–27
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Formal Methods for MPI Programs

Ganesh Gopalakrishnan and Robert M. Kirby [1],[2]

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

Abstract

High-end computing is universally recognized to be a strategic tool for leadership in science and technology. A significant portion of high-end computing is conducted on clusters running the Message Passing Interface (MPI) library. MPI has become the *de facto* standard in high performance computing (HPC). Our research addresses the need to avoid bugs in MPI programs through a combination of techniques ranging from the use of formal specifications to the use of *in-situ* model checking techniques. This paper details an assessment of the efficacy of these techniques, as well as our future work plans.

Keywords: MPI, distributed programs, message passing, formal methods, model checking, formal specifications

1 Introduction

The importance of high-end computing (sometimes called high performance computing, or HPC) needs very little motivation in a modern context. HPC is universally recognized to be a strategic tool for leadership in science and technology. A significant portion of high-end computing is conducted on clusters running the Message Passing Interface (MPI [1]) library. In these applications, system models occurring across a broad range of real-world applications—anywhere from chemical boilers to weather models—are studied through simulations. MPI has become the *de facto* standard in HPC. As an expert observes [2], MPI's popularity is due to several resons:

[1] Email: ganesh@cs.utah.edu
[2] Email: kirby@cs.utah.edu

doi:10.1016/j.entcs.2007.10.005

- Portability: The need for portability is high, because code tends to outlive hardware.
- The ability to smoothly map primitives to hardware architectures.
- The large variety of calls that allows each user to find subsets that are well-matched to their needs.
- Its orthogonality in terms of what combinations of features are allowed.

Just as is the case with parallel programs in general, MPI programs in particular can contain bugs. Specifically, the sources of MPI program bugs have been identified to include the following. First of all, the large number of functions in MPI libraries can overwhelm developers. Second, MPI is most commonly taught or learned at an informal level. As such, programmers writing advanced MPI applications may overlook corner cases. Finally, MPI programs are not static; they are sometimes manually re-tuned when ported to a new hardware platform.

This paper is about our ongoing research that emphasizes the use of formal methods for making the creation of bug-free MPI programs easier. Our approach consists of (i) developing a formal model of the MPI library, (ii) developing *in-situ* (run-time) model checking tools, and (iii) developing static analysis support for enhancing the efficacy of model checking. This paper briefly describes our ongoing work in these areas, as well as our future plans. The authors wish to acknowledge the impact that Professor Gary Lindstrom had in the parallel computing research conducted at the University of Utah, and thank him for his feedback and encouragement of the research reported here.

The rest of the paper is organized as follows. In Section 2, we present an overview of our work underway in developing a formal specification for MPI. In Section 3, we describe our work on developing an *in-situ* model checker for MPI. *In-situ* model checking was introduced in VeriSoft [13] in the context of directly model checking C/C++ programs. Ours is believed to be the first realization of this idea for MPI programs. In Section 4, we present an assessment of our work so far, and draw conclusions for the future.

Related Work:

As far as we now, there is only one other group – namely that of Siegel and Avrunin – that has actively investigated the use of formal methods for high performance computing using MPI. Some of their past publications include [7,8,9]. The main differences between these works and ours are the following: (i) in Siegel and Avrunin's work, a declarative formal specification for MPI has not been proposed; (ii) they employ traditional model checking using SPIN

- not an *in-situ* approach, as we follow. The similarities between these two efforts are in terms of both efforts aiming to impact a highly important area of concurrent programming.

2 Formal Specification of MPI

It is well documented that even experienced programmers misunderstand complex libraries such as MPI. Unfortunately, none of the current recourses they have to clarify their understanding are satisfactory: (i) the mixture of natural language and semi-formal notations used in standard documents can be misinterpreted, (ii) the behavior they observe through “experiments” conducted on actual platforms only reveal how someone else has implemented the API, and (iii) formal specifications, as they are written and made available, are of little direct help to practitioners. With the move to multicores and other novel platforms, API specifications that emphasize “what” and not “how” may lead to more efficient implementations. Program analysis, verification, and platform testing of API implementations all can benefit from formal specifications.

In our previous work [16], we captured around 10% of the MPI-1.0 primitives (mainly for point-to-point communication) in TLA+ [18]. TLA+ enjoys wide usage within (at least) Microsoft and Intel by practicing engineers. We built a C front-end in the Microsoft Visual Studio (VS) parallel debugger environment through which users can submit short (but tricky) MPI programs, with embedded assertions. The embedding C statements as well as the MPI calls are turned into TLA+, run through the TLC model checker [18]. When the assertions fail, the error traces turn into the VS debugger stepping commands. This approach gives the practitioner a tool for understanding MPI based on a formal semantic description which may, as a one-time activity, be validated by experts. Our formal specification maintains cross-reference tags with the MPI reference document, citing line and page numbers of pertinent references. Another project with similar motivations (but not a C front-end) is [17], which formalizes the kernel threads procedures of the Win32 API.

While our previous work demonstrated the promise of our approach, we left out many details including those pertaining to data transfers. Our current project addresses MPI-2.0 (which has over 300 API functions, as opposed to 128 for MPI-1.0). In addition, our new work: (i) refines our earlier work; (ii) considers many more functions; (iii) provides a rich collection of tests that help validate our specifications; and (iv) modularizes the specifications, permitting reuse. The approximate sizes (without including comments and blank lines) of the major parts in the current TLA+ specification are shown in Table 1. We overlook deprecated functions. Also, those primitives requiring us to formalize the underlying operating system have been omitted. The framework plus our TLA+ models can be downloaded from [15]. In Section 4, we describe our future plans regarding the MPI formalization work.

Main Module	**#primitives(#lines)**
Point to Point Communication	35(800)
Userdefined Datatypes	27(450)
Group and Communicator Management	34(600)
Intracommunicator Collective Communication	16(450)
Topology	18(250)
Environment Management in MPI 1.1	10(100)
Process Management	10(200)
One sided Communication	15(600(*est.*))
Intercommunicator Collective Communication	14(300)
I/O	upcoming
Interfaces and Environments in MPI 2.0	35(700)

Table 1. Size of the Specification

3 In-Situ Model Checking of MPI Programs

Bugs in MPI programs arise due to many reasons, for example: (i) during manual optimizations that turn blocking sends/receives into their non-blocking counterparts, (ii) in the context of using wild-card communications (and the resulting non-determinism), and (iii) in programs that use one-sided communication [3]. Programmers are acutely aware of the need to test parallel programs over all their interleavings. They also know that this is impossible, thus confining testing to interleavings that are guessed to be adequate – very often incorrectly, as it turns out. Model checking [5] has helped debug many concurrent systems by relying on an *exhaustive* search of all possible interleavings – albeit on a suitably abstracted system model. For example, the SPIN [6] system which recently won ACM's prestigious software award has been used to verify many real-world protocols. For model checking to be successful in practice: (i) Designers must have modestly expensive techniques to create *models* of the protocols to be verified. (ii) They must be able to examine a fraction of the interleavings in a concurrent system, and be able to claim that the remaining interleavings are equivalent and hence need not be examined, using a suitable *partial order reduction* [5,6] algorithm.

We have developed a tool called ISP (*in-situ* partial order) which aspires to grow into a practical MPI model checker. In the only other active research effort on developing a model checker for MPI programs [9], the following approach is taken: (i) they model MPI primitives directly in Promela (SPIN's modeling language), and uses SPIN for verification; (ii) they use the C extension features of SPIN to model advanced features such as non-blocking communication commands. While these efforts ride on the success of SPIN, (i) they incur a non-trivial amount of work to accurately model MPI primitives in either Promela or C; (ii) an MPI program may be buggy because of a faulty library function; (iii) accurately modeling the C++ (usually) code that embeds MPI primitives is tedious and error-prone; and (iv) many MPI programs are written with layers of user-level libraries; understanding and accurately modeling them is tedious upfront effort.

In ISP, we employ dynamic (run-time) model checking methods [13,14] based on dynamic partial order reduction (DPOR). Our first paper on ISP [4] showed how to make DPOR work for MPI on a few small examples. It showed how we can derive the advantages of partial order reduction, as well as completely avoid user effort in modeling MPI primitives, the embedding C++ code, or user level libraries – all advantages compared to [7,8,9]. This paper provides vital measurements missing in [4]. It describes our new implementation, which is a total rewrite of our old implementation, facilitating easy experimentation. Our new implementation employs the formal semantics of MPI (described in [12]) more faithfully in formulating the partial order reduction algorithm. It implements more MPI primitives, including wild-card and many collectives. It presents improvements possible due to a search optimization resembling the Chinese Postman tour [11]. Last but not least, it suggests ways to combine static analysis and ISP-based model checking.

3.1 Why Dynamic Dependence?

Partial order reduction attempts to eliminate redundant traces generated through commuting actions. Consider two C statements `a[j++]` and `a[k--]` that are concurrently enabled in two threads. These actions commute if `j !=k` – a fact best known at run-time (static analysis can, in general, not determine such information accurately). This *dependence information* is exploited by DPOR at run-time, thus helping to eliminate needless interleavings. What about the commuting behavior of MPI primitives such as wild-card communications? It turns out, as we show in [12], that treating them requires a DPOR approach. This is because the potential senders that match a wild-card is also most accurately known only at run-time. If this information is not accurately determined, say through ad hoc static analysis, either a conservative approach is needed ("all senders that potentially target the wild-card receive may match") or a dependency may be overlooked.

3.2 How ISP Works

Currently ISP checks for deadlocks and local assertion violations. For every MPI function (*e.g.* MPI_send), ISP provides a replacement function (a modest, one-time effort). When invoked, these replacement functions first consults a central scheduler (an `N+1`st process) through TCP sockets. If the scheduler gives permission, the replacement function invokes PMPI_send (that is provided with every MPI implementation, and has the same functionality as MPI_send). This allows the scheduler to march the processes of a given MPI program according to one arbitrary interleaving, till all processes hit MPI_finalize. ISP examines the resulting trace of actions, and records at each of its choice points, where a different process could have been selected. Such alternative choices are deemed necessary based on the dynamic dependence between actions in the current trace (see [14] for details). If, at choice point i, a different process p' is deemed necessary to have been run, ISP (which is implemented using "stateless search" [6]) re-executes the entire MPI program till it comes to choice point i, and picks p' to run. Our studies in [4] show that the DPOR algorithm in ISP can considerably reduce the number of interleavings than without DPOR.

3.3 Results and Assessment

We ran three simple examples: (i) the parallel trapezoidal rule computation (Trap), (ii) Monte-carlo evaluation of Pi (MC), and (iii) A byte-range locking protocol using one-sided MPI communication (BR). These programs are still "small," nowhere near the size of real-world MPI programs; however, to be able to handle real-world programs, improvements similar to those identified here must be made to ISP. The byte-range protocol was actually a realistic protocol [3]. These codes as well as additional details (including ISP's code with documentation) are available from our website [3].

• For Trap, a deliberately introduced deadlock was found instantaneously. Following correction, the program was exhaustively searched over 33 total interleavings, taking 8.94 seconds. Of this time, 8.44 seconds were spent in restarting the MPI system!

• For MC, three deadlocks (which were *not evident* upon casual inspection) were automatically detected (detailed on our website). After correcting the code, ISP ran over 3,427 interleavings, taking 15.52 minutes, of which 15.077 minutes were spent restarting the MPI system.

• For BR, one deadlock was found after exploring 62 interleavings. After correction, the code ran over 11,000 interleavings, and the run was killed

[3] `http://www.cs.utah.edu/formal_verification/ppopp08-isp/`.

(pending further improvements to ISP before we exhaust the execution space of BR).

3.4 Improvements to ISP

3.4.1 Eliminate Restart Overhead

The restart time of the MPI system is clearly a dominant overhead. This price is being paid because as opposed to existing model checkers which maintain state hash-tables, we cannot easily maintain a hash-table of visited states including the state of the MPI program as well as the MPI run-time system! In resorting to re-execution, we are, in effect, banking on deterministic replay. We considered using existing MPI checkpointing systems, but presently have assigned a low priority to this path, in lieu of the following directions.

Chinese Postman Tour: Before MPI_Finalize, how about re-initializing the user level state variables followed by a "go to" in each process to the label immediately following MPI_Init? The resulting execution will resemble a Chinese Postman tour. Preliminary experiments show that the execution time of MC reduces from 15.52 minutes to 63 seconds, finding the same deadlocks. For Trap, the time reduces from 8.94 seconds to 0.347 seconds.

Since MPI programs are "data independent" (often have variables that do not influence control flow), we will study whether (and how) static analysis may be able to help avoid re-initializing some user level variables. Also, we do not re-initialize the MPI run-time state. This may actually help detect resource leaks; e.g., many users write code sloppily relying on MPI_finalize to free-up dynamically created communicators.

Conservative State Recording: Even with DPOR, it is possible (because of the stateless search) to be repeating common "tail sequences" in the different traces explored. We will investigate conservative methods to estimate (and record) visited states; unlike in a model checker, full state recording and re-visit checking is expensive in a tool similar to ISP.

3.4.2 Hover Windows of Precision

Barrier Insertion: Can the user insert a pair of MPI_barrier instructions, requesting ISP to perform interleavings only within their confines? These barriers can, later, be moved to other parts of the MPI program. This may help users localize model checking, following their own insights into their program.

Loop Peeling: Since most MPI programs have loops from which MPI operations are invoked, it seems attractive/necessary to (through static analysis) peel out some of the iterations and have them alone be trapped and inter-leaved by ISP. The other iterations of the MPI program loops must still carry out these communications – however, without being trapped by ISP. Our preliminary implementations of this idea quickly told us that this has to be done with considerable care. In particular, at each step, ISP's current scheduler fires all the MPI processes and expects all these MPI processes to come back to it with the next instrumented MPI command (else, ISP reports a deadlock and halts). Our experience indicates that making ISP's scheduler more general is likely to complicate its code considerably. Therefore, if we were to implement loop peeling in ISP, it should be done in a manner compatible with ISP's current scheduler.

3.4.3 Strength Reduction

Since MPI programs are often data independent, suitable static analysis methods may be able to eliminate considerable amounts of the embedding C++ code.

3.4.4 Distribute Search

ISP itself follows a highly parallelizable algorithm. The alternative interleavings may well be explored by other nodes of a large cluster. Our implementation of an ISP algorithm for PThreads [10] (a different effort with no code in common) shows that with a suitably designed heuristic, linear speed-up on large clusters is a possibility. Further plans regarding ISP are presented in Section 4.

4 Assessment, and Concluding Remarks

4.1 An Assessment

Given the importance of MPI, we argue that it is necessary to develop tools that support formal methods in the design and debugging of MPI programs. Our work addresses the issue of providing a formal semantics for MPI, and building a model checker that does not require the tedious and error-prone step of modeling. Concerning the formal semantics of MPI, our future plan regarding our formal specification is to thoroughly debug it, and to fully integrate it into our Microsoft Visual Studio environment. Concerning *in-situ* model checking, we present practical model checking methods that can detect latent deadlocks in large MPI programs by intelligently orchestrating the interleavings among MPI processes. This work depends on an adaptation of the dynamic partial order reduction work of [14] to work efficiently for MPI. Details are sketched in Section 3, as well as in [4].

4.2 Future Work

Our future work will consist of two distinct phases. First, we plan to more rigorously analyze our MPI formal specification and find ways in which to solid feedback. Our plans regarding ISP are to develop new combinations of static and dynamic analysis, obtain MPI programs that are from a class known to be difficult to debug, and measure the efficacy of our tools.

References

[1] The Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. http://www.mpi-forum.org/.

[2] W. D. Gropp. "Learning from the Success of MPI," In *8th International Conference on High Performance Computing - HiPC*, 2001.

[3] S. Pervez et.al., "Formal Verification of Programs that use MPI One-sided Communication," EuroPVM/MPI, 30-39, 2006.

[4] S. Pervez et.al., "Practical Model Checking Method for Verifying Correctness of MPI Programs," EuroPVM/MPI, 2007 (accepted).

[5] E.M. Clarke, O. Grumberg, and D. Peled, "Model Checking," MIT Press, 1999.

[6] G. J. Holzmann, "The SPIN Model Checker," Addison-Wesley, 2004.

[7] Stephen F. Siegel and George Avrunin, "Verification of MPI-based software for scientific computation," In *Proceedings of the 11th International SPIN Workshop on Model Checking Software*, pages 286–303, April 2004. LNCS volume 2989.

[8] Stephen 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)*, pages 413–429, 2005.

[9] S.F. Siegel, "Model Checking Nonblocking MPI Programs," VMCAI, 2007.

[10] , Y. Yang et.al., "Distributed Dynamic Partial Order Reduction based Verification of Threaded Software," SPIN 2007.

[11] http://mathworld.wolfram.com/ChinesePostmanProblem.html

[12] , R. Palmer et.al., "Semantics Driven Dynamic Partial-Order Reduction of MPI-based Parallel Programs," PADTAD 2007.

[13] , P. Godefroid, "Model Checking for Programming Languages using VeriSoft," POPL 1997.

[14] C. Flanagan, P. Godefroid, "Dynamic partial-order reduction for model checking software," POPL 2005.

[15] www.cs.utah.edu/formal_verification.

[16] Robert Palmer, Michael Delisi, Ganesh Gopalakrishnan and Robert M. Kirby, "An Approach to Formalization and Analysis of Message Passing Libraries". The 12th International Workshop on Formal Methods for Industrial Critical Systems (FMICS), Berlin, Germany, July, 2007.

[17] The Win32 Threads API Specification. http://research.microsoft.com/users/lamport/tla/threads/threads.html

[18] research.microsoft.com/users/lamport/tla/tla.html

[19] John Reynolds. Separation logic: A logic for shared mutable data structures. The 17th IEEE Symposium on Logic in Computer Science (LICS), 2002.