

Toward Reliable and Efficient Message Passing Software Through Formal Analysis*

Ganesh Gopalakrishnan and Robert M. Kirby

University of Utah
School of Computing
Salt Lake City, UT, 84112 USA
{ganesh,kirby}@cs.utah.edu

Abstract

The quest for high performance drives parallel scientific computing software design. Well over 60% of the high-performance computing (HPC) community writes programs using the MPI library; to gain performance, they are known to perform many manual optimizations. Even tools that accept high level descriptions often generate MPI code, due to its eminent portability. However, since the overall performance of a program does not usually port (due to variations in the target architecture, cluster size, etc.), manual changes to the code are inevitable in today's approaches to MPI programming and optimization. This, together with the vastness and evolving nature of the MPI standard, and the innate complexity of concurrent programming introduces costly bugs.

Our research addresses these challenges through specific efforts in the following broad areas: (i) high level expression of the parallel algorithm and compilation thereof into optimized MPI programs, (ii) optimizations of user-written detailed MPI programs through localized transformations such as barrier removal, (iii) formal modeling of complex communication standards, such as the MPI-2 standard and a facility for answering putative queries (this need arises when standard documents are impossibly difficult to manually study in order to answer questions that are not explicitly addressed in the standard), (iv) formal modeling of new (and hence relatively less well understood) features of communication libraries, such as the one-sided communication facility of MPI-2, and (v) formal modeling of intricate control algorithms in these libraries such as the progress engine for TCP and/or shared memory in MPICH2 (a formal model can explicate commonalities, help formally verify, as well as help create better future implementations). Our research gains focus through

numerous collaborations.

1. Introduction

Progress in scientific high-performance computing (HPC) is fundamental to scientific discovery in virtually all walks of life. The quest for high performance drives parallel scientific computing software design. A programmer developing such software considers several factors ranging from algorithm selection, communication mechanisms, all the way down to the expected performance of caches and network switches. There is clearly tension between such performance-centric approaches and the quest to scale quickly towards petaflop computing regimes as, say, captured in HPC road maps [1]. To scale up along the HPC path, some of the approaches taken include virtualization as in the Charm+ system [2], automatic techniques to map graphical descriptions of the intended computation to programs as in the Uintah system [3], and automatic communication optimizations as in IBM's BlueGene/L system [4].

Despite this dichotomy of views, programmers in both camps most often end up using the Message Passing Interface (MPI, [5, 6]) library. It is estimated that well over 60% of the HPC community writes programs using the MPI library. In fact, MPI has steadily evolved from MPI-1, which contained about 130 calls, through intermediate versions to the present MPI-2 standard which contains over 190 calls. As Gropp observes [7], MPI's popularity stems from several fundamental reasons:

- portability (also considering that applications can outlive the hardware),
- smooth mapping of primitives to hardware architectures,

*Supported in part by NSF award CNS-0509379

- the variety of calls that allows each user to find subsets that are well-matched to their needs, and not suffer from a minimal but ‘one size fits all’ approach,
- and its orthogonality in terms of what combinations of features are allowed.

However, MPI is a portable standard for overall behavior, and not performance. Therefore, hand-tuned codes are sometimes manually re-tuned when ported to another hardware platform or another implementation of the MPI standard. Even the proponents of the high-level modeling approach who may automatically generate MPI programs are known to manually examine the generated code and improve its performance through manual code modifications. All these activities, coupled with the vastness and evolving nature of the MPI standard, the fact that many implementations of MPI are at various stages of implementing the standard, and the innate complexity of concurrent programming itself result in costly bugs. These bugs are found late, cost months of wasted programmer time, or often produce inexplicable results.

Formal methods are enjoying an explosive growth precisely to help eliminate these kinds of bugs, as evidenced by the abundance of papers on model-checking and assertion-based verification in conferences such as POPL, PLDI, and OSDI (e.g., [8, 9, 10, 11]) and also in hardware verification (e.g., [12]). Formal methods based on program logics and decision procedures are being used to verifying optimizing compiler transformations [13], formally analyzing device driver codes [14], and solidifying industrial standards [15]. The HPC community is facing daunting challenges with respect to large-scale concurrency, reliability, and standardization. Yet, from the relative lack of formal methods papers in HPC indicates that the use of mathematical logic, assertions, and model-checking is not widely accepted or even known to be possible in this arena.

With the vast investments in MPI, it behooves the scientific community to move towards formal methods, bridge the cultural gap between the HPC and traditional Computer Science communities, and achieve the dream of peta-flop computing through principled approaches. Even though MPI is widely regarded as a standard, it has had a healthy period of evolution where different interpretations and implementations did exist. Programming languages such as C also went through periods where compilers produced different behaviors for code fragments such as `a[i++] = i++`; with the ANSI-C standardization, things have become uniform. We believe it is time now to solidify the consensus among MPI users. One difference between C and MPI is that the latter is a concurrent

library, and humans are inherently deficient at anticipating the corner cases in concurrent executions. This elevates the importance of a formal reference specification for MPI. We do not intend this specification to dictate how MPI should be viewed; it will be mainly used to support formal reasoning, and the creation of MPI library validation test suites that complement existing suites. We will closely interact with MPI experts and formalize their understanding as well as published descriptions. We will also develop analysis tools that extract finite-state models from MPI programs (programs that use MPI calls; we plan to use MPI C-programs). We will build model-checking tools that analyze these models and automatically detect deadlocks and race conditions. We will take advantage of our MPI reference specification to enhance model-checking. Those properties that cannot be established statically will be done so at run-time through embedded assertions within MPI C-programs.

Our work brings together the domain expertise of a formal methods researcher (PI) and an active researcher in parallel scientific computing who is also coauthor of a recent book on parallel scientific computing using C++ and MPI (Co-PI). Our research is in the context of actual scientific computing applications. In particular, Kirby is associated with the Utah Scientific Computing and Imaging (SCI) Institute that has been involved in the development of HPC software [3, 16, 17, 18, 19]. We also have numerous external collaborations, the most notable being a collaboration that has recently started with the principal developers of MPICH2 (ANL). The ANL collaboration has already given us two focused problems on which to work, namely: (i) formally understanding MPI-2 constructs that achieve one-sided communication, and (ii) formally modeling some of the intricate algorithms used in MPI progress engines. These problems directly stem from the experience that the ANL group has gained in the process of creating the widely deployed MPICH2 implementation of MPI-2.

In addition to publications (two to date that are directly attributable to our grant) and research software (several in progress), the main outcomes will be student dissertations and theses (three PhD dissertations, one MS thesis, and one undergraduate research project are supported in part by this award).

Section 2 details our preliminary work on model extraction and verification applied to MPI [20]. Section 3 details work in progress and expected outcomes in the area of level expression of the parallel algorithm and compilation thereof into optimized MPI programs. Section 4 describes optimizations of user-written detailed MPI programs through localized transformations. Sec-

tion 5 describes our formal modeling of a widely used subset of the MPI-2 standard and a facility for answering putative queries. Section 6 details what we plan to achieve in modeling new features such as one-sided communication. Section 7 details how we approach the formal analysis of intricate algorithms such as used in progress engines.

Case studies are going to be a central part of our work; an important case study conducted to date is the design and performance evaluation of a parallel *and* distributed model checker written using MPI and PThreads [21]. This model checker has been extensively experimented on numerous clusters and, in fact, released for general use on our website due to its immediate applicability in modeling and model-checking large cache coherence protocols (the domain in which the sequential version of this model checker has, traditionally, been used). This case study gives us direct experience with some advanced features of MPI and PThreads and in a sense also represents how MPI might be used in aggressive asynchronous styles in scientific computing problems. A few details of this case study are presented in Section 8. Numerous other case studies stemming from the Co-PI’s research projects will also be employed as driving problems. Our research gains focus through numerous collaborations, including one with the Argonne group responsible for developing MPICH2. Section 9 has concluding remarks.

2. Model Extraction and Verification

In [20], we report our first attempt at formal analysis applied to MPI programs. Figure 1 shows the conceptual flow of a tool path that we set out to build. We employed the Berkeley CIL tool suite and used it to extract finite state models of MPI C programs. The MPI calls in these programs were modeled by finite state machines that capture how MPI calls proceed through various events, including those that mark when the process of writing into the send buffer finishes, when the message transfer begins and ends, and when the send buffer may be overwritten again. While this approach represents a “classic” approach to software model checking, numerous problems prevent this approach from scaling: (i) model extraction may not work smoothly when embedded programming pragmas or esoteric (non-MPI) constructs are employed, (ii) model extraction of large programs is hard, (iii) model checking of large extracted models will suffer from state explosion, and (iv) since MPI programs will often be written in C/C++ and at other times in FORTRAN, writing model extractors for all these languages will cause us too much engineering work up-front without com-

mensurate rapid research benefits. These experiences have turned our attention to high impact research avenues that promise better overall benefit on real-world MPI programs and MPI libraries. The directions chosen based on this experience constitute the rest of this document.

3. High Level Transformations

Many MPI programs implement expressions involving simple higher order functions such as *map*, *filter*, and *fold*. In these cases, it is attractive to allow a user to express the intent in such high level terms and automatically generate MPI programs that employ communications in lieu of function applications to “weave values” across various functions. Thus, the overall net effect would be achieved in a distributed computation setting.

MPI programs are often written at a very low level in the hope of achieving high performance when in fact doing so increases the likelihood of producing buggy and unmaintainable code. To avoid this danger, experienced MPI programmers get the code working with a semi-conservative communications pattern, and then to try to relax the pattern. Examples would be using Send/Recv’s first (to get a code working) and then to transition to ISend/Recv’s (or ISend/IRRecv’s). We are working on such an approach to semi-automated (automation, coupled with expert designer guidance) code optimization. We already have numerous examples of manually optimized MPI programs, and plan to apply our techniques to show how these examples can be re-derived using our transformation system. Our past experience in cache coherence protocol synthesis [22] has given us some background experience.

In embarking on this project, we also realize that having a formal semantics for MPI is essential in order to justify the transformations. This is addressed in Section 5.

4. Local Transformations

As opposed to transformations that affect an entire MPI process, local transformations attempt to relax the rigid synchrony and/or inefficient implementation caused by a few localized constructs. The main optimization to be researched is the removal of redundant barrier constructs. Many `barrier` calls are inserted by programmers to assist their own mental vision of the computation. We will research correctness criteria for justifying barrier removal. We plan to symbolically execute the unoptimized and optimized versions and

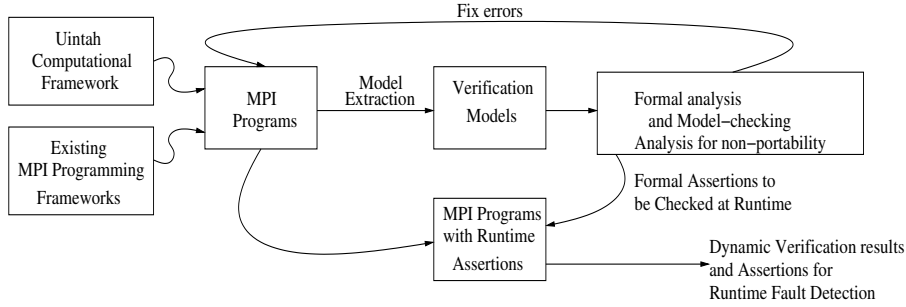


Figure 1. Conceptual Flow Diagram of Proposed Framework

develop suitable criteria for comparing the symbolic execution results. In other cases, utilizing existing optimized constructs (e.g., broadcasts using `MPI_Bcast` instead of send-receives) allows the applications programmer to attain more transparent coding, and also allows the MPI version to optimize the program for the particular architecture.

5. Formal Modeling of MPI

A formal semantics for MPI has not been written by any group - at least at a sufficient degree of detail that permits formal analysis and transformation. Our attempt in this area will result in a formal semantic description that assists in many ways, including these ways: (i) help understand the standards document as an augment to the already existing English descriptions, (ii) help justify the correctness of transformations done at the MPI program level, (iii) allow users to conclude facts about MPI that are not explicitly stated in the standards document, and (iv) help develop MPI validation suites that can stress test corner-cases and also allow one to test formal properties with respect to implementations.

In our work, the formal semantics will be captured in a formal state transition notation similar to TLA+ [23] or B [24]. Queries will be specified in a notation similar to *test automata* [25] which are finite state observers of computations. We will generate Boolean satisfiability instances out of putative queries, and using Boolean satisfiability tools [26] arrive at either an answer in the affirmative (“SAT”) or explain why certain scenarios are impossible using unsatisfiability cores [27].

We have recently finished such a formal specification for the Intel Itanium shared memory and built a tool that helps users query the specification using specific (short) shared memory concurrent assembly programs that have embedded assertions [25]. If these assertions can be satisfied, our tool emits an interleaving

of the concurrent assembly language instructions that can satisfy the assertions. If the assertions cannot be satisfied, our tool emits a precedence graph of shared memory events that cannot be simultaneously satisfied. This effort gives us valuable background experience.

6. Modeling Relatively Less-Understood MPI calls

MPI-2 has a one-sided communication construct that helps enhance the efficiency of many MPI programs - the price paid being the proclivity to bugs both due to the intricate shared memory nature of the construct as well as the potential to misread its documentation (which is also sketchy). R. Thakur of ANL has offered a mini-challenge to us by the way of requesting (i) a formal modeling approach to one-sided communication, and (ii) formally verifying a distributed locking algorithm implemented using one-sided communication. This focused project will allow us to develop formal solutions that have the potential to readily transition into practice.

7. Modeling Intricate Control Protocols

In terms of the complexity, the *progress engine* of MPI is one of the complex reactive pieces of software that exists within MPI. This complexity arises due to multiple reasons: (i) not all connections are opened statically, as the cluster may be extremely large. Thus, dynamically setting up connections, and responding to failures becomes an issue, (ii) progress engines that employ TCP sockets and shared memory are apparently ridden with complex corner cases. It is not easy to tell, for instance, whether these codes are thread-safe. One of our PhD students has chosen this problem to be his main focus.

We propose to formalize the progress subsystem and apply model checking methods as well as other program

analysis methods to find bugs within it. Such formal models can also help future implementors of progress engines avoid redundant work and learn from past implementations. Again, with R. Thakur’s involvement we will put some of our PhD students to work on these problems.

As background experience, one of our students has, through a very ambitious class project, created a formal model for the progress engine of the MPI LAM implementation in Promela [28, 29]. We will continue this line of work by generating tests for the MPI LAM system based on this formal model.

8. Case Study: a Parallel and Distributed Model Checker

Model checking of safety properties [30] can be scaled up by pooling the CPU and memory resources of multiple computers. As compute clusters containing 100s of nodes, with each node realized using multi-core (e.g., 2) CPUs will be widespread, a model checker based on the *parallel* (shared memory) and *distributed* (message passing) paradigms will more efficiently use the hardware resources. Such a model checker can be designed by having each node employ *two* shared memory threads that run on the (typically) two CPUs of a node, with one thread responsible for state generation, and the other for efficient communication, including (i) performing overlapped asynchronous message passing, and (ii) aggregating the states to be sent into larger chunks in order to improve communication network utilization. We have designed and implemented such a model checking architecture called *Eddy*. In [21], we describe the design rationale, details of how the threads interact and yield control, exchange messages, as well as detect termination. We have realized an instance of this architecture for the Murphi modeling language. Called Eddy_Murphi, we report its performance over the number of nodes as well as communication parameters such as those controlling state aggregation. Nearly linear reduction of compute time with increasing number of nodes is observed. Our thread task partition is done in such a way that it is modular, easy to port across different modeling languages, and easy to tune across a variety of platforms.

Figure 2 shows the speedup obtained by Eddy_Murphi w.r.t. Murphi (i.e. $\frac{\text{Eddy_Murphi time}}{\text{Murphi time}}$) as a function of the number of compute nodes. This figure shows that we obtain a nearly linear speedup on almost all the examples, and that on all examples we are considerably faster than standalone Murphi.

We also ran a very large protocol whose verification

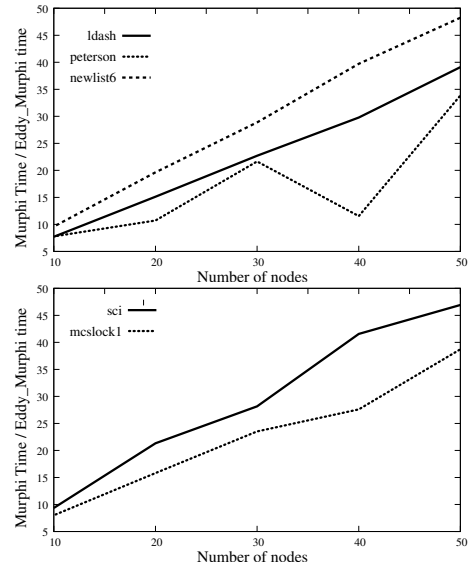


Figure 2. Eddy_Murphi linear speedup curves

is not feasible on a standalone machine. This is the case of the FLASH protocol [31] with 5 processors and 2 data values as parameters. This protocol has more than 3×10^9 states, and its verification with standard Murphi would require a huge amount of RAM memory (assuming 40 bits for each state in hash compaction, we would need 15 GB of RAM for the hash table only), as well as an unacceptable computational time. On the other hand, by using a disk version of Murphi [32], the computation lasts more than 1 week (we do not know the exact amount of time, but a projection based on the first part of the verification leads to a probable execution time of 3 weeks). However, we successfully completed the verification of this protocol with Eddy_Murphi on 60 nodes in approximately 9 hours.

9. Concluding Remarks

To summarize, within the past six months of our NSF project, we have assembled a team of capable graduate and undergraduate students, and have started working on (i) model extraction and verification, (ii) high level transformations for MPI program optimization, (iii) local transformations, and (iv) creating an executable formal semantics for MPI. In addition, we are working on two focused aspects of the MPICH2 implementation, namely (i) one-sided communication, and (ii) some progress engines employed within MPICH2. In terms of code and publications, we have published two papers, namely [20] and [21]. The

former paper involved prototype code while the latter paper developed code that has been released for general use. Considerable progress along the above lines is expected during the remaining years of our project. Our publication venues will include HPC-oriented conferences as well as formal verification and program analysis oriented conferences.

References

- [1] Daniel Reed. The roadmap for the revitalization of high-end computing. http://www.hpcc.gov/hecrtf-outreach/20040112_cra_hecrtf_report.pdf.
- [2] L.V. Kale and S. Krshnam. Charm++: Parallel programming with message-driven objects. In *Parallel Programming Using C++*, pages 175–213. MIT Press, 1996.
- [3] J. Davison de St. Germain, J. McCorquodale, S. G. Parker, and C. R. Johnson. Uintah: A massively parallel problem solving environment. In *HPDC'00: Ninth IEEE International Symposium on High Performance and Distributed Computing*, August 2000.
- [4] N.R. Adiga. An overview of the bluegene/l super-computer. In *Conference on High Performance Networking and Computing: SC2002*, pages 60–60, 2002. (with 30 co-authors).
- [5] Peter Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1996. ISBN 1-55860-339-5.
- [6] George Em Karniadakis and Robert M. Kirby II. *Parallel Scientific Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms and their Implementation*. Cambridge University Press, 2003.
- [7] W. D. Gropp. Learning from the success of mpi. In *8th International Conference on High Performance Computing - HiPC*, 2001.
- [8] G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of OSDI '96*, October 1996. Available at: <http://www.cs.berkeley.edu/~necula/osdi96.ps.gz>.
- [9] G.C. Necula, S. McPeak, and W. Weimer. Ccured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL02)*, London, January 2002.
- [10] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *Principles of Programming Languages (POPL)*, pages 232–244, January 2004.
- [11] George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of Conference on Compiler Construction (CC'02)*, March 2002. Grenoble.
- [12] Ganesh Gopalakrishnan and Warren A. Hunt Jr., editors. *Formal Methods in Systems Design*, volume 22(2). Kluwer, March 2003. Special Issue on Industrial Practice of Formal Hardware Verification: A Sampling.
- [13] Raya Leviathan and Amir Pnueli. Validating software pipelining optimizations. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 280–287, 2002.
- [14] Sriram K. Rajamani Thomas Ball. The slam project: Debugging system software via static analysis. In *POPL 2002, January 2002*, pages 1–3, January 2002. More details at: www.research.microsoft.com/slam/.
- [15] Francisco Corella, Robert Shaw, and Cui Zhang. A formal proof of absence of deadlock for any acyclic network of pci buses. In *Hardware Description Languages and their Applications*, pages 134–156. Chapman Hall, 1997.
- [16] Steven G. Parker and Christopher R. Johnson. SCIRun: a scientific programming environment for computational steering. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, 1995. Article 52, ISBN:0-89791-816-9.
- [17] Andrew S. Forsberg, David H. Laidlaw, Andries van Dam, Robert M. Kirby, George E. Karniadakis, and Jonathan L. Elion. Immersive virtual reality for visualizing flow through an artery. In *IEEE Visualization*, pages 457–460, 2000.
- [18] Robert M. Kirby, H. Marmanis, and David H. Laidlaw. Visualizing multivalued data from 2d incompressible flows using concepts from painting. In *IEEE Visualization*, pages 333–340, 1999.
- [19] R. M. Kirby, T. C. Warburton, S. J. Sherwin, A. Beskok, and G. E. Karniadakis. The $\mathcal{N}\epsilon\kappa\mathcal{T}\alpha\mathcal{R}$ code: Dynamic simulations without remeshing. In *Proc. of the 2nd International Symposium on*

- Computational Technologies for Fluid/Thermal/-Chemical Systems with Industrial Applications, ASME PVP Division, Volume 397*, 1999.
- [20] Robert Palmer, Steve Barrus, Yu Yang, Ganesh Gopalakrishnan, and Robert M. Kirby. Gauss: A framework for verifying scientific computing software. In *SoftMC: Workshop on Software Model Checking (adjoining CAV'05), Edinburgh, UK*, July 2005.
- [21] I. Melatti, R. Palmer, G. Sawaya, Y. Yang, R. M. Kirby, and G. Gopalakrishnan. Parallel and distributed model checking in eddy. In *Proceedings of the SPIN Workshop, 2006, Austria*, March 2006. To appear.
- [22] R. Nalumasu and G. Gopalakrishnan. Deriving efficient cache coherence protocols through refinement. *Formal Methods in System Design*, 20(1):107–125, January 2002.
- [23] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional (Pearson Education), 2002.
- [24] J.-R. Abrial, editor. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
- [25] Ratan Nalumasu, Rajnish Ghughal, Abdel Mokkedem, and Ganesh Gopalakrishnan. The ‘test model-checking’ approach to the verification of formal memory models of multiprocessors. In *Proceedings of the 10th International Conference on Computer-aided Verification (CAV)*, pages 464–476, 1998.
- [26] Lintao Zhang and Sharad Malik. The quest for efficient boolean satisfiability solvers. In *Computer Aided Verification*, pages 17–36, 2002. LNCS 2402.
- [27] Lintao Zhang and Sharad Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formulas. In *Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT2003)*, 2003.
- [28] Gerard Holzmann. *The Spin Model Checker Primer and Reference Manual*. Addison-Wesley, 2003.
- [29] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
- [30] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, December 1999.
- [31] J. Kuskin and D. Ofelt et al. The Stanford FLASH multiprocessor. In *SIGARCH94*, pages 302–313, May 1994.
- [32] G. Della Penna, B. Intrigila, I. Melatti, E. Tronci, and M. Venturini Zilli. Exploiting transition locality in automatic verification of finite state concurrent systems. *Software Tools for Technology Transfer*, 6(4):320–341, 2004.