Gauss: A Framework for Verifying Scientific Computing Software

Robert Palmer,   Steve Barrus,   Yu Yang,
Ganesh Gopalakrishnan,   Robert M. Kirby

School of Computing, The University of Utah, Salt Lake City, Utah

Abstract

High performance scientific computing software is of critical international importance as it supports scientific explorations and engineering. Software development in this area is highly challenging owing to the use of parallel/distributed programming methods and complex communication and synchronization libraries. There is very little use of formal methods to debug software in this area, given that the scientific computing community and the formal methods community have not traditionally worked together. The Utah Gauss project strives to make a difference by involving a domain expert in scientific computing and one in formal methods. We currently focus on MPI programs which are the kind that run on over 60% of world's supercomputers. These are programs written in C / C++ / FORTRAN employing message passing concurrency supported by the *Message Passing Interface* (MPI) library. Large-scale MPI programs also employ shared memory threads to manage concurrency within smaller task sub-groups, capitalizing on the recent availability of small-scale (e.g. single-chip) shared memory multiprocessors; such mixed programming styles can result in additional bugs. MPI libraries themselves can be buggy as they strive to implement complex requirements employing aggressive techniques such as multi-threading. We have built a model extractor that extracts from MPI C programs a formal model consisting of communicating processes represented in Microsoft's Zing modeling language. MPI library functions are also being modeled in Zing. This allows us to run formal analysis on the models to detect bugs in the MPI programs being analyzed. Our preliminary results and future plans are described; in addition, our contribution is to expose the special needs of this area and suggest specific avenues for problem-driven advances in software model-checking applied to scientific computing software development and verification.

1   Context and Motivation

Scientific supercomputing enables scientists and engineers to conduct experiments and design new products using parallel simulation. Since it plays such

a crucial role, supercomputing software must run correctly, give predictable results within predictable execution times, and not waste valuable researcher- or supercomputer time: even the energy bills of supercomputers quickly add up, let alone all the other costs! Parallel simulation is typically accomplished by capturing the system under study through a model (such as triangular meshes), subdividing the model, and distributing the pieces across multiple computers. These computers (often consisting of tens of thousands of processors) then simulate their pieces, exchanging information either through messages (usually at the macroscopic level) or shared memory (usually at finer grains). After decades of considerable turmoil, the community of supercomputer programmers has settled on a few choices with regard to the communication libraries: the Message Passing Interface (MPI) [1] for message passing and Posix [2] / OpenMP [3], etc., for shared memory. In fact, it has been estimated that MPI is used by well over 60% of world's supercomputer programmers, with this number rapidly growing. The community of programmers interested in MPI includes researchers in physics, chemistry, computational finance, and drug discovery (to name a few areas). Programmers coming from various application domains often do not have the training to employ advanced programming methods nor capabilities (or even awareness) with regard to formal verification tools. Thus they grapple with concurrency issues that, on one hand, can be understood and attacked with today's relatively mature software model-checking methods. On the other hand, one truly has to build tools and confront these issues, as the devil is in the detail. Most importantly, one often makes serendipitous advances by working in new domains. With these goals in mind, the Utah Gauss project hopes to contribute to advancements in the verification of MPI- and thread-based codes employed in scientific programming. Our initial focus will be on MPI; this will be followed by focus on thread-level modeling, mixed-style programming, and library modeling (all in some scheduling order yet to be determined).

Roadmap: We now provide a description of some of the complexity issues in this area that we are aware of (Section 1.1), an overview of MPI (Section 1.2, the design of the Gauss framework and justification of our design choices (Section 2), results (Section 3), and conclusions (Section 4).

Related Work: A plethora of bugs possible in MPI programs is discussed in [4]. The parallel programming community employs several conventional debugging tools such as TotalView [5], Parallel DBX [6], and MpiGdb [7] to debug MPI programs. More advanced tools such as Umpire [8] examine bus traces and infer the sequences of MPI calls executed. The recently proposed tool MPICHECK [9] makes several advances over traditional approaches, including algorithms to detect deadlocks. By analyzing sequences of these events, erroneous scenarios can often be quickly detected. While these tools help visualize program execution as well as tune their performance to

some extent, they do not help exhaustively search through the state-space of abstracted models, which is what model-checkers are good at.

The only prominent formal methods activity in this area (that we are aware of) is described in several papers written by Siegel, Avrunin and Mironova [10,11,12,13,14,15]. We discuss only two papers due to lack of space. In [13], the authors identify a formal subset of MPI that, essentially, omits wild-card receive statements. They prove that if MPI programs can be shown to be deadlock-free under this assumption, then the introduction of buffering does not introduce any deadlocks. In [12], the work closest to ours in terms of thrust is presented. They illustrate the power of SPIN to model MPI program control skeletons.

1.1 An overview of complexity issues

Over the years, the MPI library has steadily evolved from MPI-1, which contained 128 calls, through intermediate versions to the present MPI-2 standard which contains 194 calls. People are known to misuse the MPI library as they fail to understand it well enough. In addition, most platforms support a version of MPI that is somewhere in-between version 1 and 2. MPI libraries are written in various languages (C, C++, FORTRAN, etc.) with the host program calling the MPI functions also (usually) written in the same languages. It has also been observed that true shared memory interactions achieved through the API supported by thread libraries (such as POSIX or OpenMP) can be faster than those interactions achieved through MPI libraries *even if* the "message passing" in MPI is implemented through shared memory variables; thus, it is customary to find thread programming mixed with MPI programming, especially if the hardware realities (e.g., availability of multiple CPU-core chips) encourage this trend. Even "pure" MPI programs become quite involved, as their organization responds to the underlying hardware organization—e.g., the number of message adapters provided per CPU. Therefore, when the computational profile changes (e.g., a car being simulated suddenly crumples, sending significant quantities of the car's mass to one processor to handle), techniques such as *adaptive mesh refinement* [16] are invoked to re-define the task distribution through dynamic load balancing.

As real-world examples of problems in day-to-day usage of MPI, in the Uintah [17] Computational Framework (UCF) research project at Utah, a set of software components and libraries are being developed to facilitate the simulation of Partial Differential Equations on Structured AMR grids using hundreds to thousands of processors. UCF is regularly used on hundreds of processors on several platforms. This code has exposed bugs (deadlocks) in at least two MPI implementations, and it was very difficult to determine whether the problem was with the UCF or the MPI library. In one instance, one deadlock took *one person-month* to ultimately track down using conventional

3

debugging methods.

It is important to recall that supercomputer users are inherently performance-driven. However, MPI is intended to be a portable standard only for the overall semantics - *not* performance. This means that MPI programs are almost always modified after porting. This can introduce the following kinds of bugs: (i) the new platform may not implement an exactly compatible MPI library, (ii) the original implementation may have worked due to liberal resource availabilities, which may not be true of the (still legal) new platform; hence deadlocks are likely, and (iii) the user may rearrange the computational structures for more "balanced" performance, which can cause a new crop of bugs to emerge.

1.2   An Overview of MPI

It is our frequent experience that people in software model-checking have not seen even one MPI program; to redress this, we describe a simple MPI C program in detail (Figure 1). Like many (most?) MPI programs, this program also follows the Single Program Multiple Data (SPMD) paradigm. After initializing the MPI system via `MPI_Init`, a query for the number of processes (`MPI_Comm_size`) is made. Each process then finds out its rank in the pool of processes (`MPI_Comm_rank`). Following that, the even-numbered processes (`if mynode%2==0`) perform two sends while the odd-numbered processes perform two (hopefully) matching receives. A `barrier` is then executed by all the processes. Thereafter, the odd-numbered processes (`if mynode%2==1`) perform the sends with the even-numbered processes performing receives. The arguments of various MPI calls differ; consider one example:

```
MPI_Send(&mynode, CNT,   MPI_INT, (mynode-1+totalnodes)%totalnodes, TAG, MPI_COMM_WORLD);
```

Here, `&mynode` is the buffer from which the data originates, `CNT` is the number of bytes sent, `MPI_INT` is the datatype, `(mynode-1+totalnodes)%totalnodes` is the destination process, `TAG` is the tag, and `MPI_COMM_WORLD` is the communicator. The tag and communicator must match in order for a `MPI_Receive` to obtain the data sent. Further details about MPI may be understood from references such as [1].

2   Model Extraction and Verification in Gauss

MPI programs developed within the Uintah framework or other existing frameworks will be input by a model-extractor (Figure 2). Currently, we use CIL [18] and generate Zing [19] as the result. The extracted models will be model-checked for errors as well as potential non-portability that may result from MPI platform variations (e.g., if the MPI program assumes implementation-provided buffering which the MPI standard does not guarantee, it could deadlock when ported). We envisage having to deal with unprovable assertions via runtime checking methods.

4

```
#include<mpi.h>
#define CNT 1
#define TAG 1

int main(int argc, char ** argv){

  int mynode = 0;
  int totalnodes = 0;
  MPI_Status status;
  int recvdata0 = 0;
  int recvdata1 = 0;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
  MPI_Comm_rank(MPI_COMM_WORLD, &mynode);

  if(mynode%2 == 0){
    MPI_Send(&mynode,CNT,MPI_INT,(mynode+1)%totalnodes,TAG,MPI_COMM_WORLD);
    MPI_Send(&mynode,CNT,MPI_INT,(mynode-1+totalnodes)%totalnodes,
                TAG,MPI_COMM_WORLD);
  }
  else{
    MPI_Recv(&recvdata0,CNT,MPI_INT,(mynode-1+totalnodes)%totalnodes,
                TAG,MPI_COMM_WORLD,&status);
    MPI_Recv(&recvdata1,CNT,MPI_INT,(mynode+1)%totalnodes,
                TAG,MPI_COMM_WORLD,&status);
  }

  MPI_Barrier(MPI_COMM_WORLD);

  if(mynode%2 == 1){
    MPI_Send(&mynode,CNT,MPI_INT,(mynode+1)%totalnodes,TAG,MPI_COMM_WORLD);
    MPI_Send(&mynode,CNT,MPI_INT,(mynode-1+totalnodes)%totalnodes,
                TAG,MPI_COMM_WORLD);
  }
  else{
    MPI_Recv(&recvdata0,CNT,MPI_INT,(mynode-1+totalnodes)%totalnodes,
                TAG,MPI_COMM_WORLD,&status);
    MPI_Recv(&recvdata1,CNT,MPI_INT,(mynode+1)%totalnodes,
                TAG,MPI_COMM_WORLD,&status);
  }

  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Finalize();
}
```

Fig. 1. An Example MPI Program

2.1 Choice of Zing

Zing modeling language is designed for expressing concurrent models of software. Some of the most relevant (for us) differences between Zing and other modeling languages such as Promela are that Zing follows an object-oriented style, supports dynamic process creation, dynamic data allocation, and exceptions. The basic structure of a model in Zing is a class. Data and procedures that have operations related to these data are encapsulated within each class. Currently Zing does not support multiple inheritance.

The data types in Zing includes both primitive types, such as boolean,
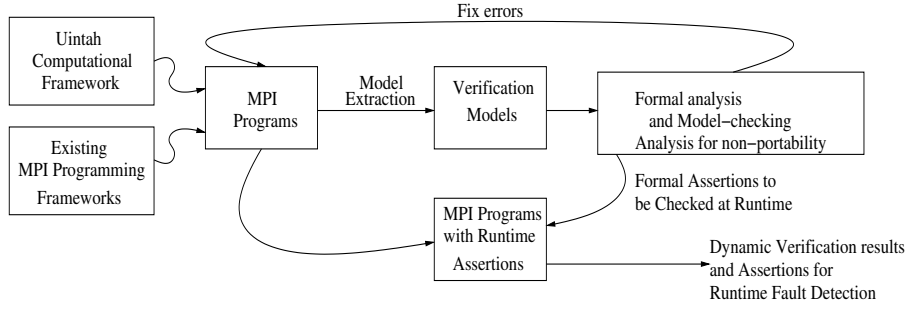
5

Fig. 2. Conceptual Flow Diagram of Proposed Framework

enumeration, integer and ranges, and user-defined complex types such as array, set, channel, etc. Unlike some other modeling languages in which the size of a channel is fixed, Zing supports channels of unbounded size. Zing also support dynamically-sized arrays. As for the control flow, Zing supports sequential control as well as procedure call and exceptional handling. When a procedure is invoked in Zing, parameters of primitive types are passed by value, and parameters of user-defined complex types are passed by reference.

To specify the entry points of a program and process creation, Zing provides keywords *activate* and *async*, with which users can specify how the program is configured and processes are dynamically created. Once a process is created, it executes concurrently with all other runnable processes. Zing processes can communicate with each other via shared variables, message queues etc. Two special statement, send and receive, are provided in Zing to express message-based communication.

Zing uses *select* statement to specify possible blocking and randomness. A list of conditional expression and corresponding actions are required for each select statement. A process in the Zing model is blocked if and only if none of the conditions is satisfied. If more than one conditions is satisfied, Zing will pick up one of them nondeterministically. Zing also provide *choose* operator to support explicit non-deterministic selection.

In summary, the choice of Zing allows us to use an expressive language to express the models, and employ Zing model-checkers developed by microsoft to get readily started. We will also develop an in-house model-checker for Zing (an effort that has been started) so that we may tune it as we see fit for this domain. Further details about Zing are described in [20,21].

2.2 Model Extraction Details

Previous discussions allow us to present our design as well as justify various choices we have made. Given the large sizes of MPI C programs, it is tempting to abstract the program and then perform model-extraction. We chose to instead extract *everything*, leaving it to a later abstraction tool to simplify

6

the extracted Zing description; in other words, we decided to keep model extraction and model abstraction separate. This is safe from the point of view of not missing out corner-cases which we are yet to fully determine (this being a new domain of work). Actually, we might keep most of the extracted Zing description abstract, and fly a "concretization lens" over the model to detail *any chosen* aspect of the model. This also gives us the ability to handle multiple host languages: they all can be subject to model-extraction to write out one common language, namely Zing.

A Zing model is a collection of classes that are similar to a Java or C# program [21,20]. Zing supports concurrency throught the presence of threads that are explicitly spawned through decorating a method with the async keyword. Thus a thread is the primary unit of sequential abstraction in a Zing model. This choice would allow us to model the primary unit of sequential abstraction in an MPI C program (namely, a process) and the unit of abstraction in thread-based code (namely, a thread) uniformly. We create a special class for the model and for each function in the program create a Zing class. Global variables become members of the class while local variables are retained within the scope of the Zing method within the class. The C language uses the main function as the primary entry point. Thus scoping and process interaction are modeled using classes and thread interaction.

The Zing language is object oriented. As such all objects are created on the Zing heap. In C, however, it is possible to take the address of an object (struct or base type) that is allocated on the stack and treat it the same as though it were allocated on the heap. It is also possible to perform pointer manipulations using arithmetic operations. Likewise it is possible to cast an integer to a pointer and then perform operations on the memory at the given address. Although from an object oriented stance these may be less desirable program constructs, in C these are widely used. To facilitate the extraction of programs that use these and other C-isms we have created a model of the base types of the C language where every memory allocation whether it be on the C stack or C heap is represented on the Zing heap as an object. This makes it possible to represent operations such as taking the address of a stack object. It will also allow us to build in safety checks to pointer arithmetic and array accesses that are not present in the actual C program.

The Zing modeling language has all of the basic types of C# except for **char** and **string** and adds the type of **byte**. Our model of basic C types includes **pointer**, **char**, **short**, **integer**, **floatingpoint**, and **doublefloatingpoint**. Our plan is to initially de-emphasize the handling of floating point values, as we hope to be able to determine the effect of data on control through abstraction/refinement techniques. In MPI, some of these interactions are, for instance, for convergance testing (e.g., "has the error value under L2-norm gone below ε?"), and we hope to cover these cases through non-deterministic

```
MPI_Recv(&recvdata1,1,MPI_INT,(mynode+1)%totalnodes,1,MPI_COMM_WORLD,&status);

__cil_tmp45 = integer.addressof(recvdata1);
__cil_tmp46 = integer.create(1);
__cil_tmp47 = integer.create(6);
__cil_tmp48 = integer.create(1);
__cil_tmp49 = integer.add(mynode, __cil_tmp48);
__cil_tmp50 = integer.mod(__cil_tmp49, totalnodes);
__cil_tmp51 = integer.create(1);
__cil_tmp52 = integer.create(91);
__cil_tmp53 = __anonstruct_MPI_Status_1.addressof(status);
MPI_Recv(__cil_tmp45, __cil_tmp46, __cil_tmp47, __cil_tmp50,
         __cil_tmp51, __cil_tmp52,__cil_tmp53);
```

Fig. 3. An extracted function call for MPI_Recv.

over-approximation.

Another important feature of MPI programs is the abundance of arrays and vectors. Arrays and operations on arrays are also represented in the model. Operations on each of these base types is carried out using an atomic static method call for that class. Constants in the program are not represented as constants in the model, rather an object is allocated for the constant and the value of the object is set to the constant.

The Zing modeling language requires that some operations that can be performed in a single C statement be simplified to multiple Zing statements. To do the extraction and simplification we have modified the CIL [18] tool in two important ways. First we have modified the pretty printer such that it targets Zing syntax (wich is similar to C in many ways but required a little tweaking) and the environment model described briefly above. The second modification was made to the *three address visitor* of CIL. This visitor seeks to simplify C programs to a three address representation. We modified this visitor to create a simplified Zing instead of a pure three address code.

Figure 3 shows the C code and Zing representation for an invocation of MPI_Recv from the Sideswap1 example. Constants such as 1, MPI_INT, and MPI_COMM_WORLD are represented explicitly as integers allocated on the heap. The address of, addition, and modulous operations were lifted out of the method call and performed seperately. Numerous modifications to the model extractor are currently in progress including the representation of casts, the addition of coalescing the simplified sequence of Zing statements into atomic regions thereby simplifying the extracted model, and constant protection and reuse (for example, the extracted version of Figure 3 has three heap locations to represent the number 1).
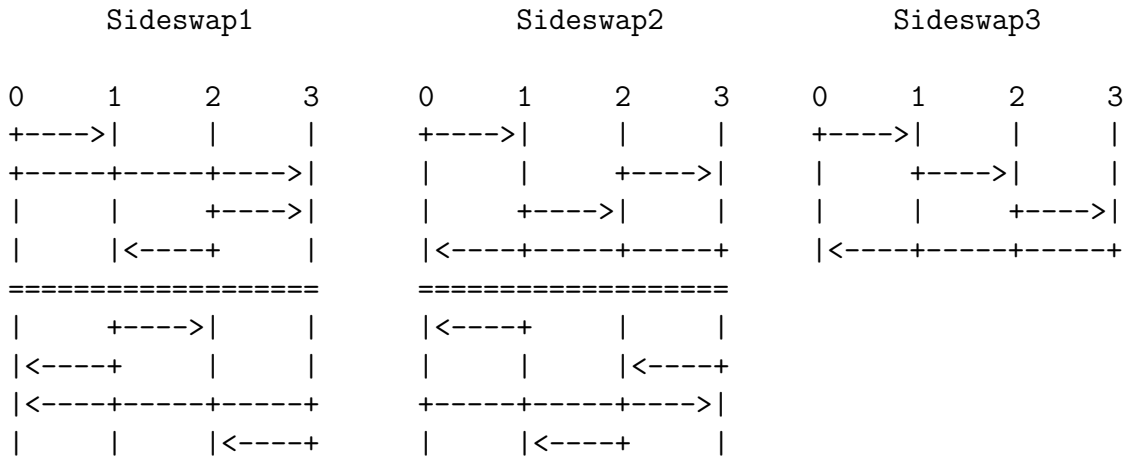
8

```
        Sideswap1                   Sideswap2                   Sideswap3

0    1     2     3         0     1     2     3         0     1     2     3
+---->|     |     |         +---->|     |     |         +---->|     |     |
+-----+-----+---->|         |     |     +---->|         |     +---->|     |
|     |     +---->|         |     +---->|     |         |     |     +---->|
|     |<----+     |         |<----+-----+-----+         |<----+-----+-----+
==================          ==================
|     +---->|     |         |<----+     |     |
|<----+     |     |         |     |     |<----+
|<----+-----+-----+         +-----+-----+---->|
|     |     |<----+         |     |<----+     |
```

Fig. 4. Communication pattern for the Sideswap1 and Sideswap2 examples. Also the cyclic dependency (causing deadlock) for the Sideswap3 example.

2.3 Library Modeling

For this project, a simple MPI library implementation was created in Zing. This MPI library includes MPI_Init, MPI_Finalize, MPI_Comm_rank, MPI_Comm_size, MPI_Send, MPI_Recv, MPI_Bcast and MPI_Barrier. This subset of the MPI standard proved to be enough to model many small sample MPI programs. Certain data structures had to be created for this implementation. The most important one was the MPI Communicator class. An MPI communicator is a means of sending and receiving messages for a collection of processes. A general global communicator, MPI_COMM_WORLD, is predefined to include all processes. We will only consider this communicator for simplicity. The MPI_Comm class was implemented as a link-list that functions much like a common queue. Every time a message needs to be sent or received, a request is placed on this queue and can later be taken off of the queue by a matching send or receive. New requests are added to the end on the list and the search for a match is always started at the head of the list. This preserves the message ordering required by the MPI standard.

We discuss MPI_Send (others omitted). MPI_Send can be broken down into a series of smaller operations. It first needs to check to see if there is a matching receive on the message queue. If there is one that matches then the receive request is taken off the queue and the send is used to fulfill that request. The data is copied from the send buffer to the buffer provided by the receive. If no matching message was found, the send request is placed on the message queue and it remains there until a matching receive call is made.

9

Name	C LOC	Zing LOC	Processes	Time	States
Sideswap1.c	38	1254	2	4	12882
Sideswap2.c	42	1222	2	4	13339
Sideswap3.c	26	1118	24	2	2522

Fig. 5. Table of results from the Gauss Framework for MPI

3   Experimental Results

We have a preliminary implementation of the toolflow in Figure 2 through which three examples (Figure 4) have been run. The Sideswap1 program is a simple pairwise communication pattern (presented through actual MPI code in Figure 1). Every process in the computation is assigned an integer value known as it's rank. This program causes every evenly ranked process to send to the odd ranked neighbor both to the right and left in a ring. The odd ranked processes perform matching receives from their even ranked neighbors. Then all processes synchronize on the barrier. In the second half of the program the odd ranked processes send and the even ranked processes receive. The Sideswap2 program is also a simple pairwise communication pattern. Even ranked processes send to their neighbor above and then receive from their neighbor below. After a synchronization the odd ranked processes do the same. The Sideswap3 program has a deadlock in a system where the MPI framework does not provide any buffering. This is a common source of bugs in MPI programs: MPI programmers who find that their code works on an installation often find that it does not work on another installation. As pointed out earlier, this is an example of an MPI program that (perhaps inadvertently) assumes implementation-provided buffering and orchestrates "too many sends" to occur in sequence, while the MPI standard does not provide such buffering guarantees. Hence the new installation to which the code is ported can exhibit a resource deadlock. Model-checking techniques permit the amount of buffering provided by the MPI communicator to be modeleled through non-deterministic over-approximation, thus helping to shake out a whole class of bugs.

4   Concluding Remarks

This paper is to encourage the software model-checking community to pay attention to an important area of international priority—namely supercomputing software development—and develop tools and techniques to support this area. We described the approach taken in the Utah Gauss framework which addresses various needs, including handling MPI and Pthreads programs, dif-

ferent host languages, and MPI/Pthreads library models. Our preliminary implementation consists of a model extractor written using Berkeley CIL and which produces Zing models. Currently we model-check the Zing models using Microsoft's Zing model-checker, and we are able to detect "textbook" bug descriptions using this tool flow.

Given the infancy of our work, we have quite a few exciting future plans. The foremost will be to develop an abstraction-refinement loop to handle very large Zing models. The second will be partial order reduction techniques that can capitalize on the semantics of MPI library functions. We envisage building static- and dynamic abstraction tools, with the project anticipated to last at least three years, producing at least two PhD students.

References

[1] MPI. http://www-unix.mcs.anl.gov/mpi/.

[2] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[3] OpenMP. http://www.openmp.org.

[4] Marc Snir and William Gropp. *MPI: The Complete Reference*. The MIT Press, 1998.

[5] TotalView. http://www.etnus.com/TotalView/.

[6] Parallel DBX Software. http://hpcf.nersc.gov.

[7] http://www-unix.mcs.anl.gov/mpi/mpich/docs/userguide/node26.htm.

[8] Jeffrey S. Vetter and Bronis R.de Supinski. Dynamic software testing of mpi applications with umpire. In *Procedings of the 2000 IEEE/ACM conference on SuperComputing*, 2000.

[9] Glenn Luecke, Yan Zou, James Coyle, Jim Hoekstra, and Marina Kraeva. Mpi-check:a tool for checking fortran 90 mpi programs. In *Concurrency and Computation:Practice and Experience*, 2003.

[10] Stephen F. Siegel and George S. Avrunin. Modeling wildcard-free mpi program for verification. In *to appear in Proceedings of the ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, 2005.

[11] Stephen F. Siegel. Efficient verification of halting properties for mpi programs with wildcard receives. In *Proceedings of Verificaiton, Model Checking,and Abstract Interpretation: 6th International Conference, VMCAI*, 2005.

[12] Stephen F. Siegel and George S. Avrunin. Verification of mpi-based software for scientific computation. In *Procedings of the 11th International SPIN Workshop*, 2004.

[13] Stephen F. Siegel and George S. Avrunin. Modeling mpi programs for verification. In *Technical Report UM-CS-2004-75*, 2004.

[14] Stephen F. Siegel and George S. Avrunin. Finite-state verification for high performance computing. In *Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications*, 2005.

[15] Stephen F. Siegel, Anastasia Mironova, George S. Avrunin, and Lori A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. Technical Report UM-CS-2005-15, Department of Computer Science, University of Massachusetts, 2005.

[16] http://flash.uchicago.edu/ tomek/AMR/.

[17] Steven G. Parker. A component-based architecture for parallel multi-physics PDE simula tion. In *Proceedings of the International Conference on Computational Scien ce-Part III*, pages 719–734. Springer-Verlag, 2002.

[18] George 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*, 2002.

[19] CIL. http://research.microsoft.com/zing/.

[20] Zing Modeling Language Specification. http://research.microsoft.com/zing/ZingLanguageSpecification.pdf.

[21] Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jacob Rehof, and Yichen Xie. Zing: A model checker for concurrent software. In *MSR Techinical Report MSR-TR-2004-10*, 2004.