Formal Analysis of Message Passing

Stephen F. Siegel* and Ganesh Gopalakrishnan**

[1] Verified Software Laboratory, Department of Computer and Information Sciences
University of Delaware, Newark, DE 19716, USA
siegel@cis.udel.edu http://vsl.cis.udel.edu
[2] School of Computing, University of Utah, Salt Lake City, UT 84112, USA
ganesh@cs.utah.edu http://www.cs.utah.edu/fv

Abstract. The message passing paradigm underlies many important families of programs—for instance programs in the area of high performance computing that support science and engineering research. Unfortunately, very few formal methods researchers are involved in developing formal analysis tools and techniques for message passing programs. This paper summarizes research being done in our groups in support of this area, specifically with respect to the Message Passing Interface. We emphasize the need for specialized varieties of many familiar notions such as deadlock detection, race analysis, symmetry analysis, partial order reduction, static analysis and symbolic reasoning support. Since these issues are harbingers of those being faced in multicore programming, the time is ripe to build a critical mass of researchers working in this area.

1 Introduction

Ever since Dijkstra introduced the notion of semaphores [10], shared memory concurrent programming has been a familiar research topic for computer scientists. Shared memory programming allows the deployment of parallel activities (threads or tasks) that access pieces of shared data. A variety of mechanisms, ranging from static scheduling to runtime schedule management using locks, ensure the integrity of shared data. The underlying hardware maintains the shared memory view by employing cache coherency protocols.

Shared memory concurrency now dominates the attention of computer science researchers—especially those interested in formal analysis methods for correctness [46, 2]. The purpose of this article, however, is to bring focus sharply onto the predicament of application scientists who had long ago realized the need for parallelism. A fairly significant milestone was reached about seventeen years ago when these scientists and computer manufacturers interested in large scale scientific computing standardized parallel programming around the Message Passing Interface, which soon became the *de facto* standard in this area.

* Supported by the U.S. National Science Foundation grants CCF-0733035 and CCF-0953210, and the University of Delaware Research Foundation

** Supported by Microsoft, and NSF CCF-0903408, 0935858.

Development of MPI has continued, with the latest edition of the MPI Standard, version 2.2, published in 2009 [26].

Message passing has long been realized as "the other dominant paradigm for parallel programming." As opposed to shared memory, message passing makes it the responsibility of programmers to explicitly move data between participant processes/threads. The semantics of message passing programs has been a popular research topic, with two notable publications being Hoare's theory of Communicating Sequential Processes [15] and Milner's Calculus of Communicating Systems [27]. However, the brand of message passing that has truly succeeded—namely MPI—is a far cry from notations such as CCS and CSP, and even their embellished programming language versions, Occam [18] and Erlang [1]. MPI 2.2 specifies over 300 primitives, including dozens of functions for *point-to-point* sending and receiving of messages, *collective* operations such as broadcast and reduce, and many other functions for structuring large-scale simulation codes using abstractions such as *communicators* and *topologies.*

It is without doubt that MPI has succeeded in a practical sense. It is the single notation that *all* application scientists around the world use for performing large-scale "experiments" on expensive supercomputers. It has enabled cutting-edge research on numerous fronts: how chemical reactions occur; how black holes evolve; how weather systems work; how new theories in physics are put to test; and how we may one day build efficient and safe nuclear reactors. This paper asks the following fair question: *what are formal methods researchers doing to help MPI programmers?* The answer unfortunately is *next to nothing!* We describe some of the research directions being pursued in our groups. We close by reiterating the importance of developing tools for message passing concurrency, both because MPI continues to be relevant for the coming decade and because other APIs and languages inspired by the message passing ideas incubated in MPI are becoming important in the upcoming era of concurrent and parallel computing.

2 An Overview of MPI and its Correctness Issues

Structure of an MPI program. An MPI program comprises some number $n \geq 1$ of MPI processes. MPI does provide for dynamic process creation, but this feature is not widely used, and we will assume n is fixed for the runtime of the program. Each MPI process is specified as a C/C++ or Fortran program which uses the types, constants, and procedures defined in the MPI library. While there is no requirement that the processes are generated from the same source code, or even written in the same language, in practice one almost always writes and compiles a single generic program and specifies n when the program is executed. The MPI runtime system instantiates n processes from this code. Though generated from the same code, the processes can behave differently because each can obtain its unique PID and the code can contain branches on this value.

MPI provides an abstraction—the *communicator*—which represents an isolated communication universe. Almost every MPI function takes a communicator as an argument. Messages sent using one communicator can never be received or have any impact upon another communicator. A set of processes is associated to each communicator. If the size of this set is m, the processes are numbered from 0 to $m - 1$; this number is the *rank* of the process with respect to the communicator. One process may take part in many communicators and have a different rank in each. MPI defines a type `MPI_Comm` for communicators, and a number of functions to create and manipulate them. The predefined communicator `MPI_COMM_WORLD` consists of all n processes; the rank of a process with respect to `MPI_COMM_WORLD` may be thought of as the process's unique PID. The function `MPI_Comm_size` is used to obtain the number of processes in a communicator and `MPI_Comm_rank` is used to obtain the rank of the calling process. For examples of these, see Fig. 2(b).

Point-to-point operations. MPI's *point-to-point* functions are used to send a message from one process to another. There are many variants, but the most basic are `MPI_Send` and `MPI_Recv`, and many useful MPI programs can be written with just these two communication operations. The sending process specifies the rank of the destination process as well as an integer *tag* which the receiver can use in selecting messages for reception. The receiving process may specify the rank of the source and the tag, but may instead use the *wildcard* values `MPI_ANY_SOURCE` and `MPI_ANY_TAG` for either or both of these arguments, indicating that a message from any source and/or with any tag can be received.

A message *matches* a receive if the communicators agree, the sources agree (or the receive uses `MPI_ANY_SOURCE`), and the tags agree (or the receive uses `MPI_ANY_TAG`). A receive cannot accept a message x from process i if there is an earlier matching message from process i that has not yet been received [26, §3.5]. This "non-overtaking" requirement means that point-to-point messaging may be modeled by a system of FIFO queues—one for each ordered pair of processes—with the exception that message tags may be used to pull a message from the middle of a queue. Early approaches to the verification of MPI programs used the model checker Spin [16] and took exactly this approach; see [23, 39].

Neither the type nor the number of data elements is used to match messages with receives. It is up to the programmer to "get these right." If the types are incompatible or the receive buffer is not large enough to contain the incoming message, anything could happen: the Standard does not require the MPI implementation to report an error. Implementations might interpret floating-point numbers as integers, or overwrite the receive buffer (perhaps resulting in a segmentation fault). If error messages are issued, they are often cryptic.

What about buffering? Unlike standard channel models, which assign a fixed capacity to each channel, the MPI model makes no assumptions about the availability of buffer space. At any time, a message sent by `MPI_Send` *may* be buffered, so the sender can proceed even if the receiving process has not reached a corresponding receive, *or* the sender may be blocked until the receiver arrives at a matching receive and the message can be copied directly from the send buffer to the receive buffer. The Standard places no restrictions on how the MPI implementation makes this decision, though in practice most implementations will base the decision on factors such as the amount of buffering space available and the size of the message. A correct MPI program must behave as expected no matter how these decisions are made. In particular, a correct program should never reach a state in which progress is only possible if a message is buffered. Even though such an action may succeed, it is also possible that the program deadlocks, depending on the choices made by the MPI implementation. This undesirable state is known as *potential deadlock*, and much developer effort is expended on avoiding, detecting, and eliminating potential deadlocks.

A formal model of programs that use a subset of MPI (including the functions described above) is described in [39,40]. Using this model, several theorems facilitating formal verification can be proved. In particular, programs that do not use `MPI_ANY_SOURCE` exhibit a number of desirable deterministic properties. For example, absence of potential deadlocks can be established by examining only synchronous executions (those in which every send is forced to take place synchronously)—this is true even in the presence of local nondeterminism within a process. If in addition each process is deterministic, absence of potential deadlocks and in fact any property of the terminal state of the program can be verified by examining any single interleaving.

All of these theorems fail for programs that use `MPI_ANY_SOURCE`. Even for these programs, however, it is not necessary to explore every possible interleaving and behavior allowed by the MPI Standard in order to verify many desirable properties, such as absence of potential deadlock. MPI-specific *partial order reduction* approaches have been developed to determine precisely when it is safe to restrict attention to a smaller classes of behaviors. The *urgent* POR scheme, for example, defines a reduced state space in such a way that when control is away from an any-source receive only a single interleaving needs to be examined, but when at such a receive multiple interleavings might have to be explored [35]. This reduction is safe for any property of potentially halted states. MPI-specific *dynamic* POR schemes are another approach [47,48].

Collectives. MPI provides a number of higher-level communication operations that involve all processes in a communicator, rather than just two. These *collective* functions include barrier, broadcast, and reduction operations. The syntax follows an SPMD style. For example, to engage in a broadcast, all processes invoke the same function, `MPI_Bcast`, with the same value for argument `root`, the rank of the process that is sending. On the root process, argument `buf` is a pointer to the send buffer, while on a non-root process, it points to the buffer that will be used to receive the broadcast message. The MPI Standard requires that all processes in the communicator invoke the same collective operations on the communicator, in the same order, and that certain arguments (such as `root`) have the same value on every process. If these conditions are violated, the behavior of the MPI implementation is undefined.

The synchronization semantics of the collective operations are also loosely defined. Certain operations, such as `MPI_Barrier` or an `MPI_Allreduce` using addition, must necessarily create a synchronization barrier: no process can leave the operation until every process has entered it. Others do not *necessarily* impose a barrier: it is possible for the root to enter and leave a broadcast operation before any other process arrives at the broadcast, because the messages it sends out could be buffered. The MPI Standard allows the implementation to choose the degree of synchronization. As in the case with `MPI_Send`, the degree of synchronization can change dynamically and unpredictably during execution. A correct program cannot assume anything.

Every MPI collective operation is functionally equivalent to a routine that can be written using point-to-point operations. Indeed, in many places the Standard describes a collective operation by giving an equivalent sequence of point-to-point operations. One might wonder why MPI specifies the collectives, since the programmer could just implement them using the point-to-points. The answer is that the collective may be functionally equivalent, but is expected to give better performance in most cases than anything that could be expressed on top of point-to-points. In the IBM BlueGene series, for example, many collective operations are mapped directly to a tree-based network optimized for communication involving all nodes in a partition. Point-to-point operations use a separate 3d-torus network. However, if one is only interested in functional correctness, this does mean that many verification techniques can be extended to the collectives "for free." The theorems mentioned above, for example, all apply to programs using collectives. (Technically, this only holds for reduction operations for which the reduction operator is commutative and associative. Since floating-point addition and multiplication are not associative, it is possible for a reduction using either operator to return different values when invoked twice from the same state. This is because the Standard does not insist that the operation be applied to the processes in any particular order, such as by increasing rank. However, this is the only source of nondeterminism arising from the use of collectives.)

Nonblocking Operations. MPI provides ways for the programmer to specify how computational and communication actions associated to a process may take place concurrently. Modern high-performance architectures can take advantage of this information by mapping these actions to separate, concurrently executing hardware components. This capability is often credited with a significant share of the high level of performance obtained by state-of-the-art simulations.

The MPI mechanism for specifying such overlap is *nonblocking communication.* The idea is to decompose the *blocking* send and receive operations discussed above into two distinct phases: the first *posts* a communication request; the second *waits* until that request has completed. Between the posting and waiting, the programmer may include any code (including other communication operations) that does not modify (or, in the case of a nonblocking receive, read) the buffer associated to the communication.

The nonblocking function `MPI_Isend` posts a send request, creates a request object, and returns a handle to that object (a value of type `MPI_Request`). This call always returns immediately, before the data has necessarily been copied out of the send buffer. (The `I` in `MPI_Isend` stands for "immediate.") A subsequent call to `MPI_Wait` on that handle blocks until the send operation has completed, i.e., until the data has been completely copied from the send buffer—either into some temporary buffer (if the send is buffered) or directly into the matching receive buffer (if the send is executed synchronously). In particular, the return of MPI_Wait does not mean the message has been received, or even that a matching receive operation has been posted. The call to MPI_Wait also results in the request object being deallocated. After that call returns, it is again safe to modify, re-use, or deallocate the send buffer. MPI_Irecv posts a nonblocking receive request and behaves similarly. These functions generalize the blocking send and receive: MPI_Send is equivalent to an MPI_Isend followed immediately by an MPI_Wait; MPI_Recv to an MPI_Irecv followed immediately by MPI_Wait.

A formal model of the nonblocking semantics, as well as a description of their realization in a model checking tool, can be found in [36]. Extensions of the theorems discussed above to the nonblocking case are given in [41].

Nonblocking operations provide a powerful mechanism to the programmer, but also a number of dangers. For example, the programmer must take care to not write to a send buffer involved in a nonblocking operation until after the call to MPI_Wait returns. As with all the other pitfalls discussed above, the behavior of the MPI implementation in the case of a violation is undefined.

Properties. In Fig. 1, we summarize a number of correctness properties that any MPI program should satisfy. The programmer cannot count on the compiler or MPI runtime to check any of these, or even to report errors if they are violated. Violations can lead to erroneous results, or to a crash several days in to a long-running simulation on an expensive supercomputer. Given the stakes, the need for tools that can verify such properties before execution is clear.

In addition to these generic correctness properties, developers have expressed interest in a number of properties that may be applicable only in certain cases, or that bear more on performance than correctness. A sampling follows:

1. *The program contains no unnecessary barriers.* (Barriers can take a huge toll on performance, but it is often difficult to decide when a particular one is required for correctness.)
2. *The number of outstanding communication requests never exceeds some specified bound.* (With most MPI implementations, performance can degrade sharply when the number of such requests becomes excessive.)
3. *Every nonblocking communication request is issued as early as possible; the completion operation is issued as late as possible.* (The goal is to maximize the overlap to get the best performance from the runtime.)
4. *A specific receive operation is always issued before the corresponding send is issued.* (Dealing with "unexpected" messages can lead to expensive memory copies and other slow-downs.)
5. *The program is input-output deterministic.* (I.e., the final output is a function only of the input, and does not depend on the interleaving or any other choices made by a compliant MPI implementation.)
6. *The program is input-output equivalent to some other given (sequential or MPI) program.* (Often, a simple sequential program is used as the starting point and serves as the specification for the optimized MPI version.)

1. For each process, no MPI function is invoked before `MPI_Init`; if `MPI_Init` is invoked then `MPI_Finalize` will be invoked before termination; no MPI function will be invoked after `MPI_Finalize`.
2. Absence of potential deadlock.
3. In MPI functions involving "count" arguments, such arguments are always non-negative; if a "count" argument is positive, the corresponding buffer argument is a valid non-null pointer.
4. Any rank argument used in an MPI function call (e.g., `source`, `dest`, `root`) lies between 0 and $m-1$ (inclusive), where m is the number of processes in the communicator used in that call. (Exceptions: `source` may be `MPI_ANY_SOURCE`, `source` or `dest` may be `MPI_PROC_NULL`.)
5. The element type of any message received is compatible with the type specified by the receive statement.
6. Assuming weak fairness, every message sent is eventually received.
7. Any message received does not overflow the specified receive buffer.
8. For any communicator, all processes belonging to the communicator issue the same sequence of collective calls on that communicator, in the same order, and with compatible arguments (`root`, `op`, etc.).
9. Every nonblocking communication request is eventually completed, by a call to `MPI_Wait` or similar function.
10. The receive buffer associated to a nonblocking receive request is never read or modified before the request completes; the send buffer associated to a nonblocking send request is never modified before the request completes.

Fig. 1. Generic correctness properties applicable to all MPI programs

3 Symbolic Execution and Reachability Analysis for MPI

Symbolic execution involves executing a program using symbolic expressions in place of ordinary concrete values [19]. It has been used for test generation, analysis and verification in many contexts. Its great advantage is that it may be used to reason about many possible input and parameter values at once. When combined with model checking techniques which reason about all possible interleavings and other nondeterministic behaviors, it can be a powerful tool in verifying properties of MPI programs such as those discussed above.

MPI-Spin [38, 42, 36, 43, 37, 41] was one of the first tools to combine model checking and symbolic execution to verify MPI programs. An extension to Spin, it adds to Spin's input language many of the most commonly used MPI functions, types, and constants. It also adds a library of functions supporting symbolic arithmetic, including a simple but fast theorem-proving capability.

One of MPI-Spin's most innovative features is the ability to establish that two programs are functionally equivalent. The idea is to form a model which is the sequential composition of the two programs and add an assertion at the final state that the outputs from the two programs agree. If the assertion can be shown to hold for all inputs and all possible behaviors of the MPI implementation, the property holds. Typically, bounds must be placed on certain inputs and parameters so that the model will have a finite number of states.

MPI-Spin requires a Promela model. To extract such a model by hand is labor-intensive and error-prone. In contrast, its successor, the *Toolkit for Accurate Scientific Software* [34,44,45], works directly from C/MPI source code. The front end automatically extracts a TASS *model*. Many of the most challenging programming language constructs can be represented directly in the model and are supported by the TASS verification engine. These include functions and recursion, pointers and pointer arithmetic, multi-dimensional arrays, dynamically allocated data, and of course, a subset of MPI. TASS also supports many MPI-specific optimizations (such as the urgent POR scheme discussed in §2) that are not possible to implement on top of Spin.

TASS performs two basic functions: (1) verification of a single program, in which properties such as those of Fig. 1 are checked, and (2) comparison of two programs for functional equivalence. In both cases, the user adds annotations to the program in the form of pragmas. These may be used to indicate which variables are to be considered the input or output, to place assumptions (such as bounds) on parameters or variables, or to specify special assertions [44]. To verify a program, TASS takes as input this annotated C code and a concrete value n for the number of processes. It constructs an internal model of an n-process instantiation of the program and performs an explicit, depth-first search of the model's state space, using symbolic expressions for all values. The symbolic module performs sophisticated simplifications of expressions and can also dispatch many of the queries. For those it cannot dispatch on its own, it invokes CVC3 [4]. In comparison mode, either, both, or none of the programs may use MPI, and the number of processes for each is specified separately.

Example: Matrix Multiplication. Consider the problem of multiplying two matrices. The straightforward sequential version is given in part (a) of Fig. 2 while part (b) presents a parallel MPI version adapted from [13].

The MPI version uses the manager-worker pattern. The problem is decomposed into a set of tasks. One process, the manager, is responsible for assigning tasks to workers and collecting and processing the results. As soon as a worker returns a result, the manager sends that worker a new task, and proceeds in this way until all tasks have been distributed. When there are many more tasks than processes, and the amount of time required to complete a task is unpredictable, this approach offers a practical solution to the load-balancing problem.

In our example, a task is the computation of one row of the product matrix. The manager is the process of rank 0, and begins by broadcasting the second matrix `b` to all workers. The manager then sends one task to each worker; the task is encoded as a message in which the data is the row of `a` and the tag is the index of that row. (A tag of 0 indicates that there are no more tasks, so the worker should terminate.) The manager waits for a response from any worker using a wildcard receive. In the message sent by the worker, the data contains the computed values for the row of the product matrix and the tag contains the index of the row. The identity of the worker whose result was received is

```
void vecmat(double vector[L], double matrix[L][M], double result[M]) {
  int j, k;
  for (j = 0; j < M; j++)
    for (k = 0, result[j] = 0.0; k < L; k++) result[j] += vector[k]*matrix[k][j];
}
int main(int argc, char *argv[]) {
  int i, j, k; double a[N][L], b[L][M], c[N][M]; /* read a, b somehow */
  for (i = 0; i < N; i++) vecmat(a[i], b, c[i]);
  return 0;
}
```

(a) Sequential version

```
#define comm MPI_COMM_WORLD
int main(int argc, char *argv[]) {
  int rank, nprocs, i, j; MPI_Status status;
  MPI_Init(&argc, &argv); MPI_Comm_size(comm, &nprocs); MPI_Comm_rank(comm, &rank);
  if (rank == 0) {
    int count; double a[N][L], b[L][M], c[N][M], tmp[M]; /* read a, b somehow */
    MPI_Bcast(b, L*M, MPI_DOUBLE, 0, comm);
    for (count = 0; count < nprocs-1 && count < N; count++)
      MPI_Send(&a[count][0], L, MPI_DOUBLE, count+1, count+1, comm);
    for (i = 0; i < N; i++) {
      MPI_Recv(tmp, M, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);
      for (j = 0; j < M; j++) c[status.MPI_TAG-1][j] = tmp[j];
      if (count < N) {
        MPI_Send(&a[count][0], L, MPI_DOUBLE, status.MPI_SOURCE, count+1, comm);
        count++;
      }
    }
    for (i = 1; i < nprocs; i++) MPI_Send(NULL, 0, MPI_INT, i, 0, comm);
  } else {
    double b[L][M], in[L], out[M];
    MPI_Bcast(b, L*M, MPI_DOUBLE, 0, comm);
    while (1) {
      MPI_Recv(in, L, MPI_DOUBLE, 0, MPI_ANY_TAG, comm, &status);
      if (status.MPI_TAG == 0) break;
      vecmat(in, b, out);
      MPI_Send(out, M, MPI_DOUBLE, 0, status.MPI_TAG, comm);
    }
  }
  MPI_Finalize();
  return 0;
}
```

(b) Parallel MPI version using manager-worker pattern

Fig. 2. Matrix multiplication

obtained from the MPI_SOURCE field of the status object; this worker is sent the next task, if work remains. Finally, all workers are sent the termination signal.

For many programs (especially those that avoid wildcards and other sources of nondeterminism), TASS can scale to very large configurations and process counts. But manager-worker programs are notorious for the combinatorial blow-up in the state space, and by their very nature, they must contain some nondeterministic construct, such as MPI_ANY_SOURCE. This example is therefore one of the most challenging for a tool such as TASS. Nevertheless, TASS is able to verify functional equivalence of the two versions over a region of the parameter space in which the number of tasks is bounded by 10, for up to 12 processes.

Fig. 3 shows various statistics arising from this use of TASS. Note that after a point, increasing n only reduces the state space: this is because a greater portion

n	transitions	statesSeen	statesSaved	stateMatches	memory (MB)	time (s)
2	58537	58538	1110	0	85	3.3
3	547958	545943	20831	3249	85	14.8
4	4214154	4187479	125521	36279	196	109.9
5	24234538	23996300	561447	275823	522	1127.5
6	86671454	85436358	1545815	1304667	1353	6593.4
7	154537494	151752013	2167303	2841957	1982	8347.8
8	140695720	137779991	1605759	2938383	1242	3732.5
9	75252734	73553814	724211	1704339	699	1400.4
10	27706410	27048664	235531	658791	255	473.2
11	10801810	10543295	90147	258921	144	192.1
12	10819370	10560855	90815	258921	146	197.0

Fig. 3. TASS performance verifying equivalence of sequential and parallel matrix multiplication programs. For each number of processes n, equivalence is verified for all L, M, and N satisfying $0 \leq L, M \leq 2$ and $0 \leq N \leq 10$. The number of tasks is N. Run on a 2.8GHz quad-core Intel i7 iMac with 16GB RAM.

of the work is distributed in the initial deterministic phase of the program. After the number of workers exceeds the number of tasks (moving from $n = 11$ to $n = 12$), there is very little change in the number of states, since one is only adding processes that never do any work. The number of states saved is only a small fraction of the number of states explored. This is because TASS never saves a state that has no chance of being seen again ("matched"), one of the keys to scalability. At the worst point, $n = 7$, more than 150 million states are explored, and over 2 million saved, taking 2.5 hours. Surely symmetry or some other reduction approach could be applied to examples such as this to reduce this computational effort, though so far no one has figured out how to do this.

4 Dynamic Analysis of MPI

It is widely acknowledged that static analysis methods for concurrent programs cannot be accurate, highly scalable, and highly automated—all at the same time. Therefore it is crucially important to have efficient dynamic verification methods for MPI programs—a trend already apparent in other areas of concurrent programming [11,29,14]. We first discuss some of the highly desirable attributes of a dynamic analyzer for MPI programs, illustrating them on a simple example (Figure 4 from [47]). We then describe our dynamic formal verifier ISP which has most of these attributes. In the past we have demonstrated [51] that non-trivial MPI programs (e.g., the 15KLOC hypergraph partitioner ParMETIS) can be analyzed using ISP even on modest computational platforms such as laptop computers. While the use of a small number of MPI processes helped in these demonstrations, it was the fact that these examples were *deterministic* that helped the most. Exploiting determinism, the MPI-specific dynamic partial order reduction algorithm used in ISP can analyze these examples in seconds, generating only one interleaving.

P_0	P_1	P_2
$Isend\ (to : 1, 22);$	$Irecv\ (from : *, x)$	$Barrier;$
$Barrier;$	$Barrier;$	$Isend\ (to : 1, 33);$
	$if\ (x == 33)\ bug;$	

Fig. 4. MPI Example Illustrating Need for MPI-specific DPOR

Unfortunately, MPI applications currently of interest to practitioners tend to be much larger, and generate many nondeterministic MPI calls. Such applications can quickly exhaust the computational as well as memory resources of ISP. Even if a scaled down model of these applications can be analyzed on modestly sized platforms, bugs can be missed because both the MPI algorithms as well as the MPI library algorithms involved while executing on larger data sets will be different from those used for executing on smaller data sets. More often than not, these examples cannot be scaled down meaningfully, or are poorly parameterized, thus preventing designers from downscaling them. To address the need for highly scalable dynamic analysis methods, we have built a preliminary tool called DAMPI (distributed analyzer for MPI programs) [50]. Already, DAMPI has analyzed many large Fortran and C applications[3]on a 1000 CPU supercomputer cluster with nearly the same level of coverage guarantees as obtainable through ISP. We now proceed to describe the basics of dynamic verification algorithms for MPI, followed by ISP and DAMPI.

Requirements of an MPI Dynamic Analyzer. An idealized MPI dynamic analyzer must possess (at least) the following features:
De-bias from the absolute speeds of the platform: Conventional execution based testing methods for MPI omit many important schedules, causing them to miss bugs even in short MPI programs [9]. This is mainly because of the fact that their executions get trapped into a narrow range of all feasible schedules [52]. To illustrate this issue, consider Figure 4. Here, a non-blocking send call is issued by P_0. The matching wait for this call is not shown, but assumed to come well after the $Barrier$ call in P_0 (and similarly for $Irecv$, the non-blocking receive from P_1 and for $Isend$ from P_2). Also note that $Irecv$ can match any sender (its argument is `MPI_ANY_SOURCE`, denoted by $*$). Therefore, after starting P_0's $Isend$ and P_1's $Irecv$, an MPI platform is actually allowed to execute the "$Barrier$" calls. This enables $Isend$ of P_2 also to be executed, thus setting up a race between the two $Isend$s within "the MPI runtime" (a distributed system) to match $Irecv$. Ordinary testing methods for MPI programs cannot influence whether $Barrier$ calls happens first or which $Isend$ matches P_1's $Irecv$. They also cannot exert control over who wins races within MPI runtimes. ISP verifies an MPI program by dynamically reordering as well as dynamically rewriting MPI calls, as described under *forcing nondeterminism coverage* (below). These

[3] Thanks to excellent profiling support developed for MPI [31], it is possible to make dynamic analysis language agnostic—an important requirement in the MPI domain.

are done with the objective of discovering the maximal extent of nondeterminism at runtime. Approaches based on 'delay padding' are unreliable for MPI and very wasteful of testing resources.

Force nondeterminism coverage: Schedule independent bugs (e.g., an allocated MPI object that is not freed) can often be caught through conventional testing. To detect schedule dependent bugs, ISP must explore the maximal extent of nondeterminism possible. Our approach will be to employ stateless model checking [11], set up a backtracking point around $Irecv(from : *)$, and *rewrite the call* to $Irecv(from : 0)$ and $Irecv(from : 2)$ in turn, pursuing these two courses. ISP can determine this set of *relevant executions* and it replays over them automatically, thus ensuring nondeterminism coverage.

Eliminate redundant tests: Ordinary schedule perturbation methods such as [52] may end up permuting the order of $Barrier$ invocations over all the $n!$ equivalent cases. Such wastage is completely eliminated in ISP which does not permute the schedules of fully deterministic MPI operations.

However, ISP can still generate redundant tests when it comes to nondeterministic operations. Patterns such as in Figure 4 where the received data is decoded tend to be somewhat rare in MPI. However, building accurate static analysis methods to detect where data decoding occurs, and to maintain such static analyzers across multiple languages requires non-trivial effort (future work). For now, we use heuristics to limit schedule explosion due to nondeterministic calls used in succession.

Base tool operation on a theory of happens-before: The decision to execute $Barrier$s before $Irecv(from : *)$ is not a one-off special case in the ISP scheduler. Special case based dynamic verification tools tend to be very brittle and formally impossible to characterize. Instead, this decision is a natural consequence of exploiting the *happens-before* order we have defined for MPI [47]. While MPI itself can be formalized in several ways (e.g., [35, 22]), we have found that it is this more "macroscopic" formal semantics of happens-before ordering that directly guides the construction of the ISP scheduler.

We conducted an *ad hoc* test of the generality of this approach by applying ISP on many simple (but fairly tricky) questions pertaining to MPI program behavior [3]. We observed that ISP's scheduler could determine program outcomes based on the happens-before relation alone.[4] The following additional analysis algorithms are based on ISP's happens-before:

- The consequences of `MPI_Send` not having adequate buffering (with respect to the message being sent) can be modeled through a happens-before edge connecting the underlying `MPI_Isend` and `MPI_Wait`.
- MPI programs can sometimes deadlock more if buffering is *increased*. We can precisely model and study whether a given MPI program has this vulnerability by analyzing the underlying happens-before structure [49].
- We include an algorithm to detect *functionally irrelevant barriers* in MPI programs [33] by analyzing the happens-before structure and seeing whether the presence of an `MPI_Barrier` alters this relation.

[4] A few flaws in ISP were also found and fixed in the process.

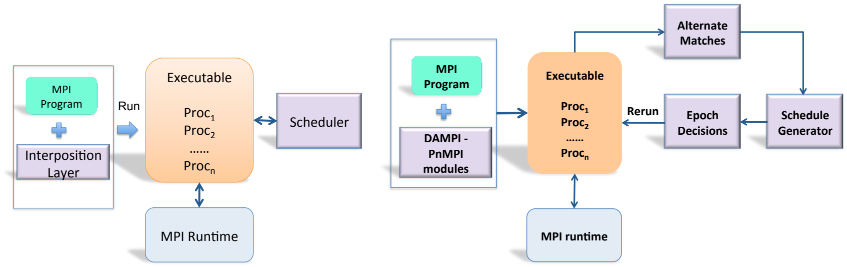

Fig. 5. ISP (left) and DAMPI (right)

Cover the input-space: The bug "*bug*" in process P_1 will be hit only if P_0 and P_2 supply the right values in their $Isend$ operations. This may ultimately be a function of the inputs to the whole program. Currently ISP does not have the ability to perform input selection; this is an important item of future work.

Provide a well engineered development environment: We have released a tool called Graphical Explorer of MPI programs (GEM, [17]). GEM is now an official part of the Eclipse Parallel Tools Platform (PTP, [28]) release 4.0. One can directly download GEM from the official Eclipse PTP site, and couple it with ISP which is released from our site. The availability of GEM makes it much easier to understand (and teach) MPI. One can use a rigorous approach in this teaching, as GEM is equipped with facilities for viewing the MPI program behavior through the happens-before relation. GEM can also display the MPI schedule from the programmers' perspective, but can also display the internally reordered schedule that ISP actually uses. The release of GEM through PTP is expected to encourage integration with other MPI tool efforts (e.g., conventional debuggers).

ISP and DAMPI. We have built two tools to carry out dynamic verification. The first is ISP (mentioned already) which exerts centralized scheduling control through a profiling layer. The ISP approach guarantees nondeterminism coverage [47] because of its dynamic MPI call reordering and rewriting already explained. ISP's dynamic verification algorithm is as follows:

- It picks a process and runs it under the control of the verification scheduler.
- ISP sends the intercepted MPI calls to the MPI runtime whenever the calls are deterministic. Nondeterministic MPI calls are delayed until after all processes are at a *fence*.
- A process reaches a *fence* when all its further operations are happens-before ordered after its current operation. At this time, ISP switches processes.
- When all processes are at a fence, ISP reaches a decision point. It can now exactly determine all the matches for an MPI nondeterministic operation. It replaces the nondeterministic calls by their determinized counterparts as explained before.

ISP's dynamic partial order reduction algorithm can be expressed as a prioritized transition system consisting of process transitions and MPI runtime transitions. The theory of ample sets [7] formalizes this algorithm.

Distributed Analyzer of MPI programs (DAMPI). ISP uses a centralized scheduler. It ends up duplicating much of what an MPI runtime would do, but precisely with a view to obtain scheduling control that is important in order to guarantee coverage. ISP's usage of the MPI library—however advanced it might be—is to merely "finish up" message matches. All these result in many limitations: (i) it is very difficult to parallelize ISP's scheduler; (ii) we slow down the processing of an MPI application by intercepting at every juncture; (iii) a highly efficient MPI library may be underutilized; (iv) ISP's scheduler is very complex and difficult to maintain. DAMPI incorporates a few key innovations. First, it tracks happens-before in a distributed setting using logical clocks. While vector clocks will ensure full coverage, Lamport clocks [20] are a much cheaper alternative adequate for realistic MPI programs. Second, it only tracks the nondeterministic component of the happens-before relation. All other calls can be "fired and forgotten." Last but not least, DAMPI allows processes to run at full speed, with piggyback messages helping convey logical clocks. At the end of each execution run, DAMPI calculates which alternative matches were possible on that run for nondeterministic receives. It then generates these alternate run schedules, and enforces them through MPI call rewriting as before. This process is repeated until the space of nondeterminism is exhausted or bugs are located.

5 Concluding Remarks

MPI continues to be of central importance to programmers in high performance computing, and will continue to hold this position for the forseeable future. In addition, MPI's influence can be seen in recently proposed message passing notations such as the Multicore Communications API [25] and the RCCE library [24]. In [32], we show that many of the lessons learned from the design of ISP can be applied to some of these APIs.

But more research and new ideas are needed in order for formal methods to become a truly practical tool for HPC developers [12]. We have already seen the challenges certain nondeterministic MPI constructs pose to standard state enumeration techniques. Similar issues arise using dynamic model checking: in the example of Fig. 2, ISP generates 18 interleavings for 4×4 matrices using four processes. This shoots up to 54 interleavings for a 5×4 times 4×5 multiplication. Certainly many, if not all, of these executions could be considered "equivalent" under some suitable notion of equivalence. The goal is to find a notion of equivalence which obtains significant reductions in commonly-occuring coding patterns, while still preserving properties of interest. There is a large body of work on symmetry reduction in model checking, but it is not yet clear how this can be applied to programs such as those of Fig. 2.

Other interesting avenues of research include applications of static analysis and Abstract Interpretation to MPI or other message-passing systems. These approaches could potentially reason without bounds on parameters or process counts. Yet very little research has been done in this area. (Some exceptions are [8,53,5].) Parametrized model checking approaches might also be applicable.

There are many avenues for further research on symbolic execution support for MPI. For example, typical scientific programs perform many complex array operations. A significant portion of the verification time involves many element-by-element symbolic array operations. If instead these could be recognized as part of a single high level operation (such as copying one segment of an array to another), the analysis could scale much further, and perhaps even deal with arrays of arbitrary size.

References

1. Armstrong, J.: Programming in Erlang: Software for a Concurrent World. Pragmatic Bookshelf (Jul 2007)
2. Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., Yelick, K.: A view of the parallel computing landscape. Comm. ACM 52(10), 56–67 (2009)
3. Atzeni, S.: ISP takes Steve's midterm exam, `http://www.cs.utah.edu/~simone/Steve_Midterm_Exam/`
4. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer (2007)
5. Bronevetsky, G.: Communication-sensitive static dataflow for parallel message passing applications. In: Proceedings of The Seventh International Symposium on Code Generation and Optimization. pp. 1–12. IEEE Computer Society (2009)
6. Cappello, F., Hérault, T., Dongarra, J. (eds.): Recent Advances in Parallel Virtual Machine and Message Passing Interface, 14th European PVM/MPI User's Group Meeting, LNCS, vol. 4757. Springer (2007)
7. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (2000)
8. Cousot, P., Cousot, R.: Semantic analysis of communicating sequential processes. In: Proceedings of the 7th Colloquium on Automata, Languages and Programming. pp. 119–133. Springer-Verlag, London, UK (1980)
9. DeLisi, M.: Test results comparing ISP, Marmot, and mpirun, `http://www.cs.utah.edu/fv/ISP_Tests`
10. Dijkstra, E.W.: Cooperating sequential processes. In: Genuys, F. (ed.) Programming Languages: NATO Advanced Study Inst., pp. 43–112. Academic Press (1968)
11. Godefroid, P.: Model checking for programming languages using VeriSoft. In: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages. pp. 174–186. POPL '97, ACM, New York, NY, USA (1997)
12. Gopalakrishnan, G.L., Kirby, R.M.: Top ten ways to make formal methods for HPC practical. In: 2010 FSE/SDP Workshop on the Future of Software Engineering Research. ACM (Nov 2010), to appear
13. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: portable parallel programming with the Message-Passing Interface. MIT Press (1999)
14. Havelund, K., Pressburger, T.: Model checking Java programs using Java PathFinder. Intl. J. on Software Tools for Technology Transfer 2(4) (Apr 2000)

15. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall Intl., Englewood Cliffs (1985)
16. Holzmann, G.J.: The Spin Model Checker. Addison-Wesley, Boston (2004)
17. Humphrey, A., Derrick, C., Gopalakrishnan, G., Tibbitts, B.R.: GEM: Graphical explorer for MPI programs. In: Parallel Software Tools and Tool Infrastructures (ICPP workshop) (2010), http://www.cs.utah.edu/fv/GEM
18. Jones, G., Goldsmith, M.: Programming in occam2. Prentice Hall Intl. Series in Computer Science (1988), http://www.comlab.ox.ac.uk/geraint.jones/publications/book/Pio2/
19. King, J.C.: Symbolic execution and program testing. Comm. ACM 19(7), 385–394 (1976)
20. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7), 558–565 (1978)
21. Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.): Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI User's Group Meeting, Proceedings, LNCS, vol. 5205. Springer (2008)
22. Li, G., Palmer, R., DeLisi, M., Gopalakrishnan, G., Kirby, R.M.: Formal specification of MPI 2.0: Case study in specifying a practical concurrent programming API. Science of Computer Programming (2010), http://dx.doi.org/10.1016/j.scico.2010.03.007
23. Matlin, O.S., Lusk, E., McCune, W.: SPINning parallel systems software. In: Bosnacki, D., Leue, S. (eds.) Model Checking of Software: 9th International SPIN Workshop. LNCS, vol. 2318, pp. 213–220. Springer-Verlag (2002)
24. Mattson, T., Wijngaart, R.V.: The 48-core SCC processor: the programmers view. In: SC10 [30], to appear
25. Multicore association, http://www.multicore-association.org
26. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, version 2.2, September 4, 2009 (2009), http://www.mpi-forum.org/docs/
27. Milner, R.: Communication and Concurrency. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1989)
28. The Eclipse Parallel Tools Platform, http://www.eclipse.org/ptp
29. Research, M.: CHESS: Find and reproduce Heisenbugs in concurrent programs, http://research.microsoft.com/en-us/projects/chess, accessed 11/7/10
30. SC10: The International Conference for High Performance Computing, Networking, Storage and Analysis, New Orleans, LA. ACM (2010), to appear
31. Schulz, M., de Supinski, B.R.: P^{N}MPI tools: a whole lot greater than the sum of their parts. In: Proceedings of the 2007 ACM/IEEE conference on Supercomputing. pp. 30:1–30:10. SC '07, ACM, New York, NY, USA (2007)
32. Sharma, S., Gopalakrishnan, G., Mercer, E., Holt, J.: MCC - A runtime verification tool for MCAPI user applications. In: 9th International Conference Formal Methods in Computer Aided Design (FMCAD). pp. 41–44. IEEE (Nov 2009)
33. Sharma, S., Vakkalanka, S., Gopalakrishnan, G., Kirby, R.M., Thakur, R., Gropp, W.: A formal approach to detect functionally irrelevant barriers in MPI programs. In: Lastovetsky et al. [21], pp. 265–273
34. Siegel, S.F.: The Toolkit for Accurate Scientific Software web page. http://vsl.cis.udel.edu/tass (2010)
35. Siegel, S.F.: Efficient verification of halting properties for MPI programs with wildcard receives. In: Cousot, R. (ed.) Verification, Model Checking, and Abstract Interpretation: 6th Intl. Conference, VMCAI 2005. LNCS, vol. 3385, pp. 413–429 (2005)

36. Siegel, S.F.: Model checking nonblocking MPI programs. In: Cook, B., Podelski, A. (eds.) Verification, Model Checking, and Abstract Interpretation: 8th International Conference, VMCAI 2007. LNCS, vol. 4349, pp. 44–58 (2007)
37. Siegel, S.F.: Verifying parallel programs with MPI-Spin. In: Cappello et al. [6], pp. 13–14
38. Siegel, S.F.: MPI-Spin web page. `http://vsl.cis.udel.edu/mpi-spin` (2008)
39. Siegel, S.F., Avrunin, G.S.: Verification of MPI-based software for scientific computation. In: Graf, S., Mounier, L. (eds.) Model Checking Software: 11th International SPIN Workshop. vol. 2989, pp. 286–303. Springer-Verlag (2004)
40. Siegel, S.F., Avrunin, G.S.: Modeling wildcard-free MPI programs for verification. In: Proceedings of the 2005 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05). pp. 95–106. ACM Press (2005)
41. Siegel, S.F., Avrunin, G.S.: Verification of halting properties for MPI programs using nonblocking operations. In: Cappello et al. [6], pp. 326–334
42. Siegel, S.F., Mironova, A., Avrunin, G.S., Clarke, L.A.: Combining symbolic execution with model checking to verify parallel numerical programs. ACM Transactions on Software Engineering and Methodology 17(2), Article 10, 1–34 (2008)
43. Siegel, S.F., Rossi, L.F.: Analyzing BlobFlow: A case study using model checking to verify parallel scientific software. In: Lastovetsky et al. [21]
44. Siegel, S.F., Zirkel, T.K.: Collective assertions. In: Jhala, R., Schmidt, D. (eds.) Verification, Model Checking, and Abstract Interpretation: 12th International Conference, VMCAI 2011. LNCS (2007), to appear
45. Siegel, S.F., Zirkel, T.K.: Automatic formal verification of MPI-based parallel programs. In: Proceedings of the 2011 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '11). ACM Press (2011), to appear
46. Sutter, H.: The free lunch is over: A fundamental turn toward concurrency in software. Dr. Dobb's Journal 30(3) (Mar 2005), `http://www.drdobbs.com/architecture-and-design/184405990`
47. Vakkalanka, S.: Efficient Dynamic Verification Algorithms for MPI Applications. Ph.D. thesis, University of Utah (2010), `http://www.cs.utah.edu/formal_verification/pdf/sarvani_dissertation.pdf`
48. Vakkalanka, S., Gopalakrishnan, G., Kirby, R.M.: Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. In: Computer Aided Verification (CAV 2008). pp. 66–79 (2008)
49. Vakkalanka, S., Vo, A., Gopalakrishnan, G., Kirby, R.: Precise dynamic analysis for slack elasticity: Adding buffer without adding bugs. In: Recent Advances in the Message Passing Interface, 17th European MPI Users' Group Meeting, EuroMPI 2010. LNCS, vol. 6305 (Sep 2010)
50. Vo, A., Aananthakrishnan, S., Gopalakrishnan, G., de Supinski, B.R., Schulz, M., Bronevetsky, G.: A scalable and distributed dynamic formal verifier for MPI programs. In: SC10 [30], `http://www.cs.utah.edu/fv/DAMPI/sc10.pdf`, to appear
51. Vo, A., Vakkalanka, S., DeLisi, M., Gopalakrishnan, G., Kirby, R.M., Thakur, R.: Formal verification of practical MPI programs. In: PPoPP. pp. 261–269 (2009)
52. Vuduc, R., Schulz, M., Quinlan, D., de Supinski, B., Sæbjørnsen, A.: Improving distributed memory applications testing by message perturbation. In: PADTAD '06: Proceeding of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging. pp. 27–36. ACM (2006)
53. Zhang, Y., Duesterwald, E.: Barrier matching for programs with textually unaligned barriers. In: Proceedings of the 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming. pp. 194–204. PPoPP '07, ACM (2007)