Verification of MPI Programs Using Spin

Steven Barrus, Ganesh Gopalakrishnan,
Robert M. Kirby, Robert Palmer

UUCS-04-008

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

Abstract

Verification of distributed systems is a complex yet important process. Concurrent systems are vulnerable to problems such as deadlock, starvation, and race conditions. Parallel programs written using the MPI (Message Passing Interface) Standard are no exception. Spin can be used to formally verify a parallel program if it is given an accurate model written is Spin's process meta language (Promela). In this paper, we describe a generalized framework for verification of MPI-based parallel programs using the Spin model checker. Only select MPI calls are covered, but this framework could potentially be extended to include all of the MPI Standard. Our reduced MPI implementation (written in Promela) is designed to follow the MPI Standard as well as allow for the flexibility provided in certain aspects (like buffering). We also present a few examples to illustrate the use of our MPI implementation in Promela.

Verification of MPI Programs Using Spin

Steven Barrus, Ganesh Gopalakrishnan,
Robert M. Kirby, Robert Palmer

18th May 2004

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

Abstract

Verification of distributed systems is a complex yet important process. Concurrent systems are vulnerable to problems such as deadlock, starvation, and race conditions. Parallel programs written using the MPI (Message Passing Interface) Standard are no exception. Spin can be used to formally verify a parallel program if it is given an accurate model written is Spin's process meta language (Promela). In this paper, we describe a generalized framework for verification of MPI-based parallel programs using the Spin model checker. Only select MPI calls are covered, but this framework could potentially be extended to include all of the MPI Standard. Our reduced MPI implementation (written in Promela) is designed to follow the MPI Standard as well as allow for the flexibility provided in certain aspects (like buffering). We also present a few examples to illustrate the use of our MPI implementation in Promela.

1 Introduction

The MPI Standard[3] has been popular for years and has provided developers with a powerful way to create portable parallel programs. MPI is used on supercomputers and clusters around the world. Given an efficient way to parallelize a program, designers can get complex programs to run is a fraction of the time. However, MPI programs are susceptible to synchronization bugs just like other such systems. There is currently no standard mechanism for finding these bugs (like deadlock, starvation, etc.), but other projects, such as Spin, could potentially address this very issue. There are a number of tools available for verification of parallel programs and Spin is arguably the most powerful and the most popular[4]. Spin is ideal for modeling distributed systems, such as message passing protocols, which makes it well suited for modeling MPI-based programs[1].

1.1 Message passing

Message passing is a common way to share data between processes in a concurrent system. This method has been around for a long time and it has seen many different interfaces. The MPI Standard is an attempt to standardize the way the message passing is done and has been widely successful. The standard outlines and efficient and robust interface for implementing parallel programs that rely on message passing. It also provides MPI-based programs with portability. An MPI program can easily be made to work with any MPI library and any system that has that library. For ten years, the MPI Standard has addressed many of concerns about message passing. However, there is one issue with message passing that has not been resolved.

Sending and receiving provide a conceptional basis for all more extensive message passing commands expressed within the MPI Standard. A message is passed when one process does a send and another does a receive. But what happens when all processes are sending and none are receiving or vice-versa? Deadlock and other synchronization problems can creep into any system that employs message passing. MPI-based programs are susceptible to these bugs just like other such systems. The MPI Standard may be powerful, but that does not mean that it takes care of these bugs for you. Luckily, there are tools available that help eliminate synchronization bugs using formal verification.

1.2 Spin

Spin is a software model checker. Models are created in Spin's own process meta-language, Promela and then Spin can be used to validate correctness of those models. It is used for formal verification of concurrent systems. If it finds that a requirement is not satisfied, it can produce an execution path that leads to the violation[4].

Since Spin is a useful tool for verifying concurrent systems, putting MPI and Spin together makes sense and in many ways they are a good match. Unlike many languages, Promela has built-in channels for communication between processes. These channels allow for data to be passed between processes similarly to the way MPI passes messages. A model of MPI calls can be built around these channels. In this paper we will walk through the modeling of 12 MPI calls and show examples of how this model can be used in MPI programs.

2 Spin Models of MPI Calls

Every implementation of the MPI Standard has a rich set of details that make it unique. Our Promela MPI model is no exceptions. The MPI Standard is not entirely strict guidelines. It allows for some implementation freedom that can make programs behave differently on under different MPI implementations. This can make bugs difficult to detect and also makes it difficult to model. In our model, we try to account for the areas of flexibility in the standard and in some cases, this means sacrificing simplicity. This section introduces the technical

details of our Promela MPI implementation. It first covers the framework for the overall system and then lays out the details of the individual MPI calls.

Most of the MPI calls require a mechanism for communication with other processes. MPI uses what are called communicators to accomplish this task. Promela provides channels which are well suited to model MPI communicators. The data that gets sent over a communicator is critical to calculations preformed by an MPI program. A generalized model of an MPI program is not so demanding because it is not actually doing any kind of scientific calculation. These models are more focused on the communication patterns rather than the flow of the computationally intensive portions of the application. Modeling the actual calculation with Spin would be difficult since Promela does not provide floating-point numbers (or floating-point arithmetic).

We model communicators with an array of channels that has the length equal to the number of processes in that group. The array is indexed by a process's rank and therefore each process has its own channel. If a message is to be sent it is put in the channel correspond to the destination process. If a message is to be received, the receiving processes checks the channel corresponding to its rank. The size of these channels is set to a constant before the model is simulated or verified. Channels of zero size (i.e. rendezvous channels) are currently unsupported because the channel poll operator does not work on such channels. Also, our model currently only provided the the global communicator, MPI_COMM_WORLD.

```
chan MPI_COMM_WORLD[PROC_SIZE] = [CHAN_SIZE] of { byte, byte, byte };
```

2.1 MPI_Send

MPI_Send is one of the most straight-forward MPI calls to model in Promela. It can simply pass a message on a channel (`chan`) using Promela's send operator. The data gets sent on a channel of the communicator that corresponds to the rank of the destination process. The send operation will block if the channel has no buffer space available (i.e. the channel is full). This directly correspond to the MPI Standard which states that the routine may block until the message is received. The data is sent along with the tag and the rank of the sending process. The information sent, aside from the data, can be used by the receiving process to determine whether or not to accept that message or wait for another one.

```
inline MPI_Send(buf, dest, tag, comm)
{
    comm[dest]!buf,tag,PROC_RANK
}
```

2.2 MPI_Recv

The MPI_Recv routine is a little more complicated. It is modeled with Promela's chan receive operator. However, it requires additional checks to see if a message in the channel (indexed by rank) matches the source and tag provided. If a match is found, the call returns and if not, the call will block until a message does match. Since MPI supports source and tag wild-cards (MPI_ANY_SOURCE and MPI_ANY_TAG), fields of the message may be ignored and discarded. This also means that a constant process rank and a tag identifier must be reserved for the wild-cards. This limits the maximum number of processes, as well as tags, to 254.

```
inline MPI_Recv(buf, source, tag, comm)
{
    if
    :: (tag == MPI_ANY_TAG) ->
        if
        :: (source == MPI_ANY_SOURCE) ->
            comm[PROC_RANK]??buf,_,_
        :: else ->
            comm[PROC_RANK]??buf,_,eval(source)
        fi
    :: else ->
        if
        :: (source == MPI_ANY_SOURCE) ->
            comm[PROC_RANK]??buf,eval(tag),_
        :: else ->
            comm[PROC_RANK]??buf,eval(tag),eval(source)
        fi
    fi
}
```

2.3 MPI_Sendrecv

The MPI_Sendrecv call does a send and receive. It can be modeled by directly calling the MPI_Send and MPI_Recv. The order of the the sending and receiving is not specified in the standard, so we have to use Promela's nondeterminism to get an accurate model[1]. This way both possible paths will be verified.

```
inline MPI_Sendrecv(sendbuf, dest, sendtag, recvbuf, source, recvtag, comm)
{
    if
    :: MPI_Send(sendbuf, dest, sendtag, comm) ->
        MPI_Recv(recvbuf, source, recvtag, comm);
    :: MPI_Recv(recvbuf, source, recvtag, comm) ->
```

```
            MPI_Send(sendbuf, dest, sendtag, comm);
        fi
    }
```

2.4 MPI_Bcast

The MPI_Bcast routine broadcasts a message from a specified root process to all other processes. Our model for the broadcast routine is simple. It has the root process iterate over all other processes, sending the data to each. All of the other process merely call MPI_Recv which will block waiting for a message with the special broadcast tag.

```
inline MPI_Bcast(buf, root, comm)
{
    if
    :: (root == PROC_RANK) ->
        byte _mpi_i;
        _mpi_i = 0;
        do
        :: (_mpi_i < PROC_SIZE) ->
            if
            :: (_mpi_i != PROC_RANK) ->
                MPI_Send(buf, _mpi_i, _MPI_BCAST_TAG, comm);
            :: else
            fi;
            _mpi_i++
        :: else -> break
        od
    :: else ->
        MPI_Recv(buf, root, _MPI_BCAST_TAG, comm);
    fi;
    MPI_Barrier();
}
```

2.5 MPI_Gather

MPI_Gather gathers data from all processes. It can be modeled essentially by doing the reverse of our broadcast model. The root process iterates over all other process and waits to receive data from each. The other processes simply have to do an MPI_Send to the root.

```
inline MPI_Gather(sendbuf, recvbuf, root, comm)
{
    if
    :: (root == PROC_RANK) ->
        byte _mpi_i;
```

```
        _mpi_i = 0;
        do
        :: (_mpi_i < PROC_SIZE) ->
            if
            :: (_mpi_i != PROC_RANK) ->
                MPI_Recv(recvbuf, _mpi_i, _MPI_BCAST_TAG, comm)
            :: else
            fi;
            _mpi_i++
        :: else -> break
        od
    :: else ->
        MPI_Send(sendbuf, root, _MPI_BCAST_TAG, comm);
    fi;
    MPI_Barrier();
}
```

2.6 MPI_Reduce

MPI_Reduce combines the values from all processes to a single value. The model for this call is similar to MPI_Gather. The difference is that the root process has to preform a reduction operation to the received data.

```
inline MPI_Reduce(sendbuf, recvbuf, op, root, comm)
{
    if
    :: (root == PROC_RANK) ->
        byte _mpi_temp, _mpi_i;
        _mpi_i = 0;
        recvbuf = sendbuf;
        do
        :: (_mpi_i < PROC_SIZE) ->
            if
            :: (_mpi_i != PROC_RANK) ->
                MPI_Recv(_mpi_temp, _mpi_i, _MPI_BCAST_TAG, comm);
                if
                :: (op == MPI_MAX) ->
                    recvbuf = ((recvbuf > _mpi_temp) -> recvbuf : _mpi_temp);
                :: (op == MPI_MIN) ->
                    recvbuf = ((recvbuf < _mpi_temp) -> recvbuf : _mpi_temp);
                :: (op == MPI_SUM) ->
                    recvbuf = (recvbuf + _mpi_temp);
                :: (op == MPI_PROD) ->
                    recvbuf = (recvbuf * _mpi_temp);
                :: (op == MPI_LAND) ->
                    recvbuf = (recvbuf && _mpi_temp);
```

```
                :: (op == MPI_BAND) ->
                    recvbuf = (recvbuf & _mpi_temp);
                :: (op == MPI_LOR) ->
                    recvbuf = (recvbuf || _mpi_temp);
                :: (op == MPI_BOR) ->
                    recvbuf = (recvbuf | _mpi_temp);
                :: (op == MPI_BXOR) ->
                    recvbuf = (recvbuf ^ _mpi_temp);
                :: else
                fi
            :: else
            fi;
            _mpi_i++
        :: else -> break
        od
    :: else ->
        MPI_Send(sendbuf, root, _MPI_BCAST_TAG, comm);
    fi;
    MPI_Barrier();
}
```

2.7 MPI_Scatter

MPI_Scatter sends data from one process to all other processes. It can be modeled as a MPI_Gather to the root process and then a MPI_Broadcast to all processes.

```
inline MPI_Scatter(sendbuf, recvbuf, root, comm)
{
    MPI_Gather(sendbuf, recvbuf, root, comm);
    MPI_Bcast(recvbuf, root, comm);
}
```

2.8 MPI_Allgather

The MPI_Allgather call collects data from all processes and distributes the collected data back to each process. The model for the call can just make a call to MPI_Gather to make the collection and then call MPI_Bcast to distribute the collected data.

```
inline MPI_Allgather(sendbuf, recvbuf, comm)
{
    MPI_Gather(sendbuf, recvbuf, _MPI_COMM_ROOT, comm);
    MPI_Bcast(recvbuf, _MPI_COMM_ROOT, comm);
}
```

2.9 MPI_Allreduce

The MPI_Allreduce call combines data from all processes and distributes the results back to all of the processes. The model for this call makes a call to MPI_Reduce to collect and combine the data and then it makes a call to MPI_Bcast to distribute the results.

```
inline MPI_Allreduce(sendbuf, recvbuf, op, comm)
{
    MPI_Reduce(sendbuf, recvbuf, op, _MPI_COMM_ROOT, comm);
    MPI_Bcast(recvbuf, _MPI_COMM_ROOT, comm);
}
```

2.10 MPI_Barrier

MPI_Barrier is a procedure that blocks until all other processes in the communicator have reached an MPI_Barrier call. The model for the MPI_Barrier routine is a little less straight-forward than some of the others. It could potentially be modeled in a number of different ways, but none are trivial. This call has to be able to be called multiple times is succession. In our model we elect one process to be the master of the lock. When it enters, it waits for the all of the other processes to enter as well. The other process enter the barrier if the barrier lock is not set and then waits for that lock to be activated before leaving. Once each process enters, the master then sets the lock allowing all of the other processes to leave the barrier. The master must then wait for each process to actually leave before reseting the lock making it safe for a subsequent call.

```
byte _mpi_barrier = 0;
byte _mpi_barrier_lock = 0;
inline MPI_Barrier()
{
    (!_mpi_barrier_lock);
    if
    :: (PROC_RANK == _MPI_COMM_ROOT) ->
        (_mpi_barrier == (PROC_SIZE - 1));
        _mpi_barrier_lock = 1;
        (_mpi_barrier == 0);
        _mpi_barrier_lock = 0;
    :: else ->
        _mpi_barrier++;
        (_mpi_barrier_lock);
        _mpi_barrier--;
    fi;
}
```

2.11 MPI_Probe

MPI_Probe is a blocking call that tests for a message. The message is required to match the specified source and tag (which can be wild-cards). MPI_Probe makes use of Promela's ability to poll channels. It can checks for messages that match certain conditions. In this case, it check the source and the tag (both optionally) for a match and ignores the message data.

```
inline MPI_Probe(source, tag, comm)
{
    if
    :: (tag == MPI_ANY_TAG) ->
        if
        :: (source == MPI_ANY_SOURCE) ->
            comm[PROC_RANK]??[_,_,_];
        :: else ->
            comm[PROC_RANK]??[_,_,eval(source)];
        fi
    :: else ->
        if
        :: (source == MPI_ANY_SOURCE) ->
            comm[PROC_RANK]??[_,eval(tag),_];
        :: else ->
            comm[PROC_RANK]??[_,eval(tag),eval(source)];
        fi
    fi
}
```

2.12 MPI_Iprobe

MPI_Iprobe is a non-blocking call for message testing. The model for MPI_Iprobe uses MPI_Probe directly. Since the call to MPI_Probe is placed in a conditional part of the if statement, it will not block. The flag is then set to 1 if the a message matches and 0 otherwise.

```
inline MPI_Iprobe(source, tag, comm, flag)
{
    if
    :: MPI_Probe(source, tag, comm) ->
        flag = 1;
    :: else ->
        flag = 0;
    fi
}
```

The Promela models of these various MPI routines show many of the qualities of MPI and Spin. They are both powerful tools and seem to fit well together.

Promela provides a fairly straight-forward way for modeling MPI routines. It is almost as if they are a natural match. Another import lesson learned here is that Promela could potentially be used to model the entire MPI Standard. Since we were able to model many of the basic calls that make up the foundation of MPI, it does not take much imagination to see that this path can take us deeper into to the standard and may even provide a way completely through.

3 Examples

Having models of the MPI calls makes it easier to model entire MPI programs. This section illustrates the usefulness of an MPI implementation in Promela using three examples.

3.1 Greetings

We will start out with a simple example[5]. Below is a MPI program that does one of two things. If its rank is not 0 then it sends a message to the process with rank 0 that consists of one integer (MPI_INT) that is the rank of the sending process. If its rank is 0 then it calls MPI_Recv p-1 times, where p is the total number of processes. After process 0 receives a message, it prints out a greeting from the sending process.

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{
    int rank, p, i;
    int message;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    if (rank != 0) {
        message = rank;
        MPI_Send(&message, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    } else {
        for (i = 1; i < p; i++) {
            MPI_Recv(&message, 1, MPI_INT, MPI_ANY_SOURCE,
                MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            printf("Greeted by %d\            n", message);
        }
    }
    MPI_Finalize();
    return 0;
}
```

This example only uses two MPI calls (MPI_Send and MPI_Recv). This makes it very easy to model in Promela and it is make even easier to model with our collection of pre-modeled MPI routines.

```
#include "mpi.prom"
MPI_Proctype main()
{
    byte rank, p, i;
    byte message;
    MPI_Comm_size(p);
    MPI_Comm_rank(rank);
    if
    :: (rank != 0) ->
        message = rank;
        MPI_Send(message, 0, 0, MPI_COMM_WORLD)
    :: else ->
        i = 1;
        do
        :: (i < p) ->
            MPI_Recv(message, MPI_ANY_SOURCE, MPI_ANY_TAG,
                MPI_COMM_WORLD);
            printf("Greeted by %d\n", message);
            i++
        :: else -> break
        od
    fi
}
```

As you can see, the Promela code looks very similar to the MPI code that is being modeled. One of the goals of our Promela-based MPI implementation is to look and fell much like MPI in C or Fortran. However, you may have noticed that the models of MPI_Send and MPI_Recv are missing two of the arguments, the count (or size) and the data type. This is because the models do not need this information be be correct and Spin do not support all of the data types that MPI requires (like floats).

3.2 Trapazoidal Rule

Here is an example of a model of a more practical MPI program[5]. This program shows a simple illustration of the trapezoidal rule used for integral approximation. Each of the processes involved calculates an equal sized portion of the integral and then sends its results to the root process where the final integral is summed up.

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
```

```
{
    int my_rank, p;
    float a = 0.0, b = 1.0;
    int n = 1024;
    float h, local_a, local_b;
    int local_n;
    float integral, total;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    h = (b-a)/(float) n;
    local_n = n/p;
    local_a = a + (float) my_rank * (float) local_n * h;
    local_b = local_a + (float) local_n * h;
    integral = Trapezoidal(local_a, local_b, local_n, h);
    MPI_Reduce(&integral, &total, 1, MPI_FLOAT, MPI_SUM,
        0, MPI_COMM_WORLD);
    if (my_rank == 0)
        printf("Integral = %f\n", total);
    MPI_Finalize();
    return 0;
}
```

This example is limited to one MPI function, MPI_Recude. However, it is more complicated around the MPI calls and does floating-point computations that have to be abstracted away in our Promela model.

```
#include "mpi.prom"
MPI_Proctype main()
{
    byte my_rank;
    byte integral, total;
    MPI_Comm_rank(my_rank);
    /* Computed local integral */
    integral = 1;
    MPI_Reduce(integral, total, MPI_SUM, 0, MPI_COMM_WORLD);
    if
    :: (my_rank == 0) ->
        printf("Integral = %d\n", total);
    :: else
    fi
}
```

3.3 Jacobi's Method

The final example that we present is more complicated than the others[5]. It involves Jacobi's method for solving linear systems which is an iterative method

that can be parallelized. The details of the method itself are outside the scope of this document. We will focus on the modeling of the MPI code.

```
/* Return 1 if iteration converged, 0 otherwise */
int Jacobi(MATRIX_T A_local, float x_local[], float b_local[], int n,
     float tol, int max_iter, int p, int my_rank)
{
    int i_local, i_global, j;
    int n_bar, iter_num;
    float x_temp1[MAX_DIM];
    float x_temp2[MAX_DIM];
    float *x_old, *x_new;
    n_bar = n/p;
    MPI_Allgather(b_local, n_bar, MPI_FLOAT, x_temp1, n_bar,
        MPI_FLOAT, MPI_COMM_WORLD);
    x_new = x_temp1;
    x_old = x_temp2;
    iter_num = 0;
    do {
        iter_num++;
        Swap(x_old, x_new);
        for (i_local = 0; i_local < n_bar; i_local++) {
            i_global = i_local + my_rank*n_bar;
            x_local[i_local] = b_local[i_local];
            for (j = 0; j < i_global; j++)
                x_local[i_local] -= A_local[i_local][j]*x_old[j];
            for (j = i_global+1; j < n; j++)
                x_local[i_local] -= A_local[i_local][j]*x_old[j];
            x_local[i_local] = x_local[i_local]/A_local[i_local][i_global];
        }
        MPI_Allgather(x_local, n_bar, MPI_FLOAT, x_new, n_bar,
            MPI_FLOAT, MPI_COMM_WORLD);
    } while ((iter_num < max_iter) && (Distance(x_new, x_old, n) > tol));
    MPI_Gather(x_local, n_bar, MPI_FLOAT, x_local, n_bar,
        MPI_FLOAT, 0, MPI_COMM_WORLD);
    if (Distance(x_new, x_old, n) <= tol)
        return 1;
    else
        return 0;
}
```

Creating a model for this code is much more complicated because the result of a foalting-point calculation (`Distance`) controls the flow of the program. In order to abstract this calculation, we have to again rely on Promela's non-determinism.

```
#include "mpi.prom"
bit dist;
```

```
inline Distance(rank)
{
    if
    :: (rank == 0) ->
        if
        :: (true) -> dist = 0;
        :: (true) -> dist = 1;
        fi;
    :: else -> skip;
    fi;
    MPI_Barrier();
}
MPI_Proctype main()
{
    byte p, rank;
    byte iter_num;
    byte b_local = 1;
    byte x_local, x_new;
    byte max_iter = 2;
    MPI_Comm_size(p);
    MPI_Comm_rank(rank);
    MPI_Allgather(b_local, x_new, MPI_COMM_WORLD);
    iter_num = 0;
    do
    :: iter_num++;
        x_local = 1;
        MPI_Allgather(x_local, x_new, MPI_COMM_WORLD);
        Distance(rank);
        if
        :: ((iter_num >= max_iter) || (dist)) -> break;
        :: else -> skip;
        fi;
    od;
    MPI_Gather(x_local, x_local, 0, MPI_COMM_WORLD);
    if
    :: rank == 0 ->
        if
        :: (dist) ->
            printf("Converged\n");
        :: else ->
            printf("Did not converged\n");
        fi
    :: else ->
    fi
}
```

This example shows that the modeling process is not always straight-forward. This may be a problem if we ever wanted to automate the conversion from C to Promela.

4 Conclusion

Every concurrent system has certain requirements that it must meet. One requirement common to almost all such systems is the absence of synchronization problems. MPI-based programs are no exception. Programmers do not want and in many cases, can not afford to have these bugs in their systems. Formal verification tools, like Spin, provide a means of detecting flaws of this nature. Since the MPI Standard does not outline a means for verification, Spin can fill in with its strength in validation.

The MPI community could benefit greatly from formal verification. The framework outlined in the paper could help MPI programmers find bugs before they submit their jobs to be run on a supercomputer after a week of waiting. Finding a bug in their job after it crashes would waste a considerable amount of time. Also, after that bug has been eliminated, how can they be certain that different bug will not cause another failure? Using Spin for verification can reduce the overall effort in creating a reliable MPI program by finding bugs before they can cause a problem.

What the future may bring is always uncertain. However, looking at the existing MPI routines that we have modeled in Promela, one could foresee models of all of the MPI calls. This may or may not be possible, but a complete model the MPI Standard is certainly desirable. Also, automation is missing from what we have shown. Currently, each MPI program has to be modeled by hand which can be complicated and lead to incorrect models. In the future, perhaps a method can be devised that will model and verify existing MPI programs with minimal work by the user. This is a tantalizing prospect that, if coupled with a complete model of the MPI Standard, could address the matter of verification MPI programs and help ensure that they are free from concurrency bugs.

References

[1] Siegel, S. F., Avrunin, G. S.: Verification of MPI-Based Software for Scientific Computation. Department of Computer Science, University of Computer Science, University of Massachusetts, 2004.

[2] Message Passing Interface Standard 2.0. http://www.mpi-forum.org/docs/, 1997.

[3] Message Passing Interface Standard 1.1. http://www.mpi-forum.org/docs/, 1995.

[4] Holzmann, G. J.: The Spin Model Checker. Addison-Wesley, Boston, 2004.

[5] Pacheco, P. S.: Parallel Programming with MPI. Morgan Kaufmann Publishers, San Francisco, 1997.