Chapter 12
ISP Tool Update: Scalable MPI Verification

Anh Vo, Sarvani Vakkalanka, and Ganesh Gopalakrishnan

Abstract We provide a status update of ISP, our dynamic formal verifier for MPI programs. ISP determines and explores all relevant schedules of an MPI program. The highlights of this paper are (i) a recap of ISP's features, and (ii) an overview of work in progress, including a graphical explorer for message passing (GEM) and a distributed MPI analyzer (DMA), an adaptation of ISP for the distributed setting.

12.1 Introduction

The Message Passing Interface (MPI) [6] library remains one of the most widely used APIs for implementing distributed message passing programs. Its projected usage in critical, future applications such as Petascale computing [5] makes it imperative that MPI programs be free of programming logic bugs. This is a very challenging task considering the size and complexity of optimized MPI programs. In particular, performance optimizations often introduce many types of non-determinism in the code. For example, the `MPI_Recv(MPI_ANY_SOURCE, MPI_ANY_TAG)` call that can potentially match a message from any sender in the same communication group (we will later refer to this as a *wildcard receive*) is often used for re-initiating more work on the first sender that finishes the previous item of work. A more general version of this call is the `MPI_Waitsome` call that waits for a subset of the previously issued communication requests to finish. These non-deterministic constructs result in MPI program bugs that manifest intermittently – the bane of debugging.

Traditional MPI debugging tools such as Marmot [8] insert delays during repeated testing under the same input to pertube the MPI runtime scheduling. Experience indicates that this technique is often unreliable [1]. In order to detect all scheduling-related bugs, the framework under which MPI programs are debugged

Anh Vo, Sarvani Vakkalanka, Ganesh Gopalakrishnan
School of Computing, University of Utah, Salt Lake City, UT 84112, USA

M.S. Müller et al. (eds.), *Tools for High Performance Computing 2009*,
DOI 10.1007/978-3-642-11261-4_12,

needs to have the ability to *determine* and *enforce* all *relevant* schedules. These three concepts are explained in § 12.1.1, including why only the relevant schedules must be enforced. These concepts underlie ISP [11, 12, 13, 14], a unique dynamic verifier for MPI programs. In the rest of this paper, we provide an updated status of ISP matching the tool presentation in the workshop.

12.1.1 Determining and Enforcing Relevant Schedules

Determining Schedules: In an MPI program, the MPI calls are always issued in program order, and the computational (C or Fortran) code is also executed in program order. However, MPI calls may match out of program order [12]. Furthermore, there are non-deterministic send/wild-card receive matches that cannot be discovered unless these out of order matches are considered.

```
P0:Isend(to P1, &h0) ; Barrier;                    Wait(h0);
P1:Irecv(*, &h1)     ; Barrier;                    Wait(h1);
P2:                  Barrier; Isend(to P1, &h2);   Wait(h2);
```

Fig. 12.1: Illustration of Barrier Semantics and the POE Algorithm

Consider the example in Figure 12.1 where an `MPI_Isend` is issued by `P0` and another `MPI_Isend` is issued by `P2`, both directed at `P1` which issues a wildcard `Irecv`. There are `Barrier` calls intersperced in this code.

According to the MPI library semantics, no MPI process can *issue* an instruction past its barrier unless all other processes have *issued* their barrier calls; however these calls need not be matched when a `Barrier` is crossed. The following execution is possible: (i) `Isend(to P1, h0)` is issued but not matched yet, (ii) `Irecv(*, h1)` is issued but not matched, (iii) the processes issue *and match* their barriers, (iv) `Isend(to P1, h2)` is issued but not matched, and (v) now both sends and the receive are alive, and non-determinism arises.

However, notice that in any schedule where P0's `Isend` and P1's `Irecv` are forced to match (*i.e.*, issue order matchings), there is no non-determinism in this program! ISP is the only dynamic formal verification tool into which the out of order matching semantics of MPI is built in.

Enforcing Schedules: Once we determine that non-determinism is possible in Figure 12.1, we need to have a scheduling strategy to discover this non-determinism and then to enforce it. ISP discovers non-determinism through a single idea (basically): *delay any non-blocking non-deterministic receive if allowed by the MPI semantics* [12]!

In this example, the ISP scheduler collects the `Irecv` command but does not issue it into the MPI runtime (effectively delaying it). This forces the execution to proceed in such a manner that the non-deterministic matches are revealed.

Once revealed, ISP invokes its program replay mechanism. That is, ISP is designed to remember the non-deterministic choices discovered and re-execute an MPI program to cover this space of non-deterministic matches. During each *replay*, it *dynamically rewrites* the wildcard `Irecv(*, &h1)`: first, it is turned into a `Irecv(from P0, &h1)` and during the next replay, it is turned into `Irecv(from P2, &h1)`. This way, the ISP scheduler can "fire and forget" the `Irecv(from P0, &h1)` and `Isend(to P1, &h0)`, knowing that within the MPI runtime, these instructions have no choice, but to match! Had the ISP scheduler simply issued `Irecv(*, &h1)` (without determinizing it first), this receive could have matched *a send from yet another process, say* `P3` – not the intended one.

Why only the *Relevant* Schedules? If *all possible schedules* of an MPI program are considered, a considerable degree of wasted testing happens. For example, nothing is gained by permuting the order in which (i) the send/receive MPI calls are *issued*, (ii) the order in which the `Barrier's` are *issued*. For instance, a delay introducing tool may simply explore all $N!$ ways of issuing a `Barrier` call whereas ISP simply issues the barriers in one canonical way (it issues them together).

Thus the key difference between an *ad hoc* testing tool and a formal dynamic verifier such as ISP is that whereas the former tools get caught in an exponentially growing amount of irrelevant schedules, ISP avoids *many* of these exponentials. In practice, this makes all the difference between complete verification and ad hoc testing with omissions. For example, in a recent series of dynamic verification runs using ISP, ParMETIS [7] (a widely used hypergraph partitioner of over 14,000 lines of MPI C code) was verified [2] for 32 processes on an ordinary laptop computer in under 40 minutes, generating a message log file of 512 MB! This verification could simply not be imagined if we permuted all 32-way barriers within this code, posted all determinstic sends and receives in every order, etc.

12.2 ISP Overview

At a high level, ISP works by intercepting the MPI calls made by the target program and making decisions on when to send these MPI calls to the MPI library. This is accomplished by the two main components of ISP: the Interposition Layer and the Scheduler. Figure 12.2 describes the general framework of ISP, showing the interaction between the ISP library (linked with the executable) and the ISP scheduler.

The Interposition Layer: The interception of MPI calls is accomplished by compiling the ISP interposition layer together with the target program source code. The interposition layer makes use of the profiling mechanism PMPI. It provides its own version of `MPI_`f for each corresponding MPI function f. Within each of these

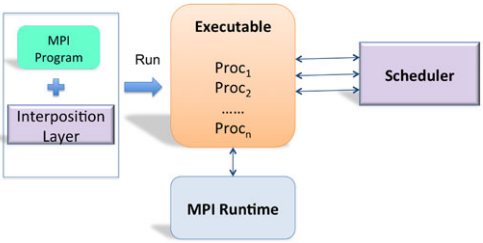

Fig. 12.2: Overview of ISP

`MPI_`f, the profiler communicates with the scheduler using TCP sockets[1] to send information about the MPI call that the process wishes to make. It will then wait for the scheduler to make a decision on whether to send the MPI call to the MPI library or to postpone it until later. When permission to fire f is granted from the scheduler, the corresponding `MPI_`f will be issued to the MPI run-time. Since all MPI libraries come with functions such as *PMPI_*f for every MPI function f, this approach provides a portable and light-weight instrumentation mechanism for MPI programs being verified.

The ISP Scheduler: The scheduler is where our main scheduling algorithm, namely POE (Partial Order avoiding Elusive interleavings) is carried out. The scheduler meets the following objectives: G1: discovers the maximal set of sends that can match a wildcard receive (viewed across all MPI-standard compliant MPI libraries); G2: accurately models the semantics of the global operations (such as barriers) of MPI. In MPI, not all MPI operations issued by a process complete in that order, and a proper modeling of this out-of-order completion semantics is essential in order to meet goals G1 and G2. For example, two `MPI_Isend` commands issued in succession by an MPI process P1 to the same target process (say P2) are forced to match in order, whereas if these `MPI_Isends` are targeted to two *different* MPI processes, then they *may match contrary to the issue order*. As another example, any operation following an `MPI_Barrier` must complete only after the barrier has completed, while an operation issued *before* the barrier may linger across the barrier, as already illustrated using Figure 12.1.

Main Steps of the POE Algorithm: The POE algorithm works as follows. There are two classes of statements to be executed: (i) those statements of the embedding programming language (C, C++, and Fortran) that do not invoke MPI commands and (ii) the MPI function calls. The embedding statements in an MPI process are local in the sense that they have no interactions with those of another process. Hence, under POE, they are executed in program order. When an MPI call f is encountered, the scheduler records it in its state; however, it does not (necessarily) issue this call into the MPI run-time. (Note: When we say that the scheduler issues/executes MPI call f, we mean that the scheduler grants permission to the process to issue the corresponding PMPI_f call to the MPI run-time). This process continues until the

[1] When running within a local machine, ISP uses unix sockets to reduce communication overhead.

scheduler arrives at a *fence*, where a fence is defined as an MPI operation that cannot complete after any other MPI operation following it. The list of such fences include `MPI_Wait`, `MPI_Barrier`, *etc.*, and are formally defined in [11]. When all MPI processes are at their individual fences, the full extent of all senders that can match a wildcard receive becomes known, and dynamic rewriting can be performed with respect to these senders. The collection of sends and matching receives can then be issued. In the case where a receive does not have a matching send (or vice versa) and there is no other MPI calls that can proceed, a deadlock is declared by the scheduler.

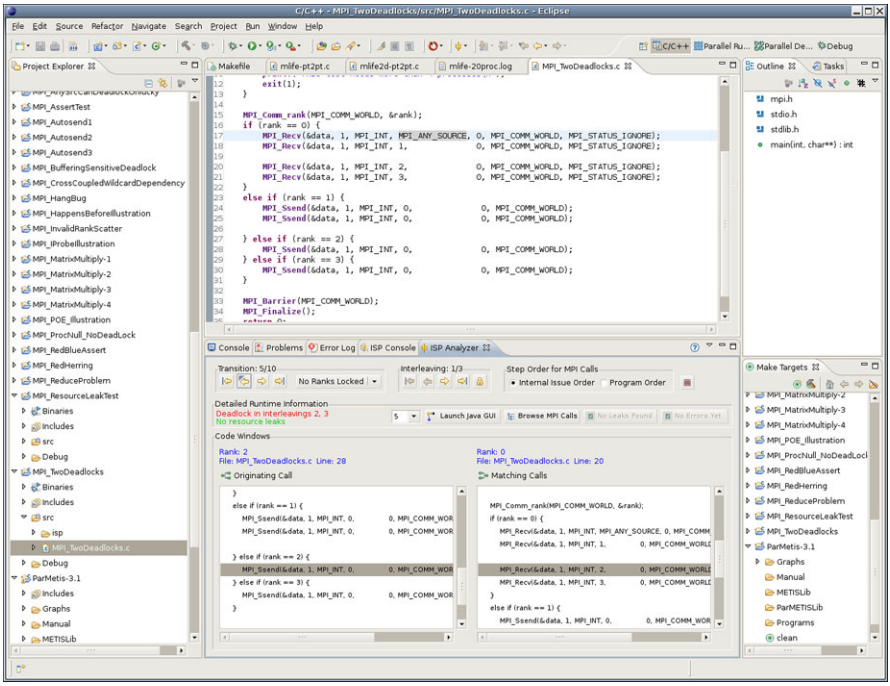

Fig. 12.3: ISP plugin in Eclipse

Completes-Before Ordering: The Completes-Before (CB) ordering accurately captures when two MPI operations *x* and *y* issued from the same process in program order are guaranteed to complete in that order. For example, if an MPI process `P1` issues an `MPI_Isend` that ships a large message to `P2` and then issues `MPI_Isend` that ships a small message to `P3`, it is possible for the second `MPI_Isend` to complete first. A summary of the completes-before order of MPI is as follows: (i) **Send Order**: Two `Isend`s sending data to the same destination complete in issue order. (ii) **Receive Order**: Two `Irecv`s receiving data from the same source complete in issue order. (iii) **Wildcard Receive Order**: If a wildcard `Irecv` is followed by another `Irecv` (wildcard or not), the issue order is respected by the completion order.

(iv) **Wait Order**: A `Wait` and another MPI operation following it complete in issue order. For a formal description of the CB relation, please see [12].

12.3 GEM

GEM (Graphical Explorer for Message passing) is an Eclipse plugin which serves as a graphical front-end for ISP. Given a collection of files to analyze using ISP, GEM helps compile and links the files using the ISP library, and then invoke ISP's scheduler on the executable creating the log file containing the post-verifcation results. GEM then parses the log file and organizes its contents. It then attempts to associate MPI calls with one another (e.g., sends need to be associated with their corresponding receives). GEM also allows users to view the execution results according to the program order or according to ISP's internal execution order (modeled by the Completes-Before-Ordering).

In addition to the usual Eclipse textual console view, GEM also provides an analyzer view that serves three functions: (i) summarize verification results, (ii) link to the completes-before viewer, (iii) allow the users to step through matching MPI calls. Figure 12.3 shows a small MPI project opened in Eclipse under GEM while Figure 12.4 depicts the analyzer view obtained by running a 10-process version of ParMETIS through GEM.

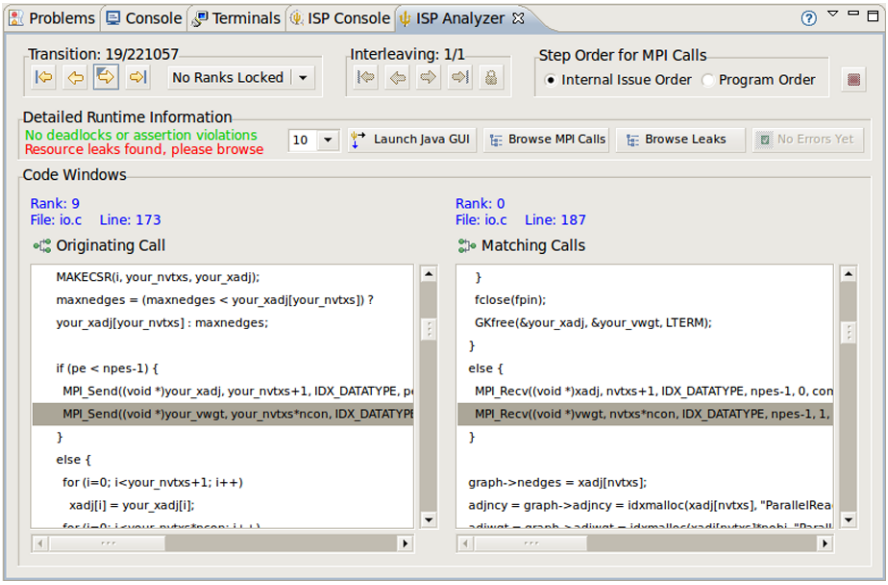

Fig. 12.4: ISP plugin in Eclipse

For a more detail description of GEM, please see [2].

12.4 DMA - A Distributed MPI Analyzer

Up until now, ISP has been used to verify several large programs such as ParMETIS, MADRE [10], MPI-Blast [3], and some of the SPECMPI2007 [4] benchmarks. However, most of the experiments have been confined a small number of processes. Fortunately, most MPI bugs will manifest themselves within these scale-down experiments. Unfortunately, *some* MPI bugs can only be reproduced when the program is run with a large number of processes. With Petascale computing coming to reality and Exascale computing looming in the horizon, the need for a scalable tool that can formally verify very large MPI programs is clear. § 12.4.1 will discuss several technical limitations of ISP and we will propose our design of the new framework in § 12.4.2.

12.4.1 Limitations of ISP

ISP was designed as a debugger that works best in a single machine setting. The assumption is that most MPI bugs still manifest themselves even with a small number of processes. However, as mentioned before, there are bugs that will manifest only at the extreme end of computing and it is unrealistic to expect the developers to roll back to a desktop version of the program to debug. Since [13] was published, many improvements were made to ISP to enhance the performance of the tool, both in terms of scalibity and speed. However, there are key technical limitations of ISP which prevent further significant scaling. We list the two most important issues here:
Centralized Scheduler: ISP employs a centralized scheduler which communicates with the MPI processes through TCP sockets. Later versions of ISP can take advantage of Unix sockets when operating within a machine to eliminate the socket communication overhead. For small to medium MPI programs, the verification can complete within a reasonable amount of time in a desktop/workstation. However, for large programs, the MPI processes will often need to be executed in a cluster environment (launching a couple hundred threads on a local machine can slow down the verification tremendously, even on today's powerful machines). While ISP can operate in distributed mode, which means that the processes compiled with the interposition layer can be run distributedly in a cluster, the scheduler remains centralized and becomes the bottleneck of the whole system. This is mainly due to amount of sequential processing that has to be done by the scheduler, and also due to the memory limitation of a single machine. Despite being a stateless verifier (meaning that it does not store all visited states), ISP still needs to store the information along the current execution trace so that it can remember correctly which interleavings to

explore. When the number of MPI processes and MPI calls become large, even this space grows inordinately.

Synchronous Communication: The ISP scheduler and the MPI processes operates in a lock-step fashion, which means the MPI processes communicate in a handshake fashion with the scheduler about each MPI call (except those that have purely local semantics such as type creation, communicator grouping, etc.). Since ISP has to hold back communication to build the completes-before edges properly, this handshake mechanism is necessary. However, when the number of processes is large enough, the scheduler has to cycle through each process to do handshaking, which means several MPI processes will unnecessarily block to wait for the response from the ISP scheduler.

12.4.2 DMA

In order to address the limitations of ISP, we have designed a new dynamic verification framework called DMA. Since the implementation is still in its early phase, this paper will only provide a brief overview of the framework.

12.4.2.1 PnMPI

DMA is designed to operate as a P^{N}MPI [9] module. P^{N}MPI extends the PMPI profiling interface to support multiple PMPI-based tools (ISP is an example of a PMPI-based tool). The advantages of using P^{N}MPI are as follows:

- P^{N}MPI allows the implementation of the DMA tool to be split up into multiple layers, with each layer addressing orthogonal issues (interleaving generation layer, resource leak tracking layer, deadlock detection layer, etc.).
- P^{N}MPI also eliminates the need to recompile the MPI target code every time changes are made to DMA
- Each layer or all of DMA layers can be turned off through a simple configuration file, which enables the developers quickly to choose which aspect of the program he wants to debug (or not debug at all)
- P^{N}MPI has very low overhead. For more details on P^{N}MPI overhead, please see [9]

12.4.2.2 Going Distributed

With the lessons learned from developing ISP, DMA is designed as a distributed tool. In DMA, each process will maintain its own trace and figure out the *potential* matching that can occur for each nondeterministic MPI event that happens within itself. To accomplish this, each process needs to construct its *view of the world*

through *piggybacking* (i.e., sending extra information in each MPI message), which itself is implemented as a P^NMPI module. Intuitively, the view helps the process reason about the completes-before relationship that was described earlier. In the case of ISP, the scheduler maintains the trace record of all the processes and thus can construct the global view by itself. DMA maintains the trace record in each process and eliminates the need for centralized processing.

After the program finishes, the information written out by each process can then be processed offline by a *Scheduler Generator* which then generates the necessary interleavings upon which the MPI processes will follow upon restarting. During all these interleavings exploration, other DMA layers can also check the processes for resource leaks, deadlocks, and local assertion violations.

12.4.2.3 Practical Considerations

DMA currently has two different implementations which serve different set of programs. The first implementation ignores the potential dependency between concurrent non-deterministic events, in which certain choices made by one event can affect the number of choices for the other event. The second implementation takes into account all the possible dependencies between concurrent events. Obviously, the second implementation has a higher overhead than the first one. However, based on our experiments with ISP, many programs do not exhibit dependency between concurrent nondeterministic events.

In our experience, many large MPI programs tend to follow several programming patterns, such as master-slave, work-stealing, nearest-neighbor, etc. Several of these patterns employ heavy usage of wildcard receives, which results in an exponentially large number of relevant interleavings to be explored. Yet, only a handful of these interleavings represent meaningful code paths that have to be verified. The rest of them can be declared safe once a representative of those interleavings is checked to be safe. The analysis for this is complex and is part of future work for DMA

12.5 Conclusions

In this paper, we provided a status update of our dynamic verifier ISP matching the tool presentation made at the 3rd Parallel Tools Workshop. We described the graphical integration (fully explained in [2]) and a distributed version of ISP under design.

Our future plans include a true *in situ* verification of large MPI programs made possible by the distributed dynamic verification algorithm of the DMA tool. Our plans also include an integration of the graphical capabilities of the tool presented at [2] with DMA.

Acknowledgements The authors have a long list of Gauss group members as well as researchers of the Lawrence Livermore National Laboratories (LLNL) as well as IBM to thank: of the Gauss group members, we like to especially point out the work of Alan Humphrey and Chris Derrick of Utah on the GEM tool, in collaboration with Beth Tibbitts and Greg Watson (IBM). We also thank Bronis de Supinski, Martin Schulz, and Greg Bronevetsky of LLNL for joint work on DMA.

References

1. http://www.cs.utah.edu/formal_verification/ISP_Tests/.
2. http://www.cs.utah.edu/formal_verification/ISP-Eclipse/.
3. http://www.mpiblast.org/.
4. http://www.spec.org/mpi.
5. A. Geist. Sustained Petascale: The next MPI challenge. Invited Talk at EuroPVM/MPI 2007.
6. W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.
7. G. Karypis. METIS and ParMETIS. http://glaros.dtc.umn.edu/gkhome/views/metis.
8. B. Krammer, K. Bidmon, M. S. Müller, and M. M. Resch. Marmot: An MPI analysis and checking tool. In *Parallel Computing 2003*, Sept. 2003.
9. M. Schulz and B. R. de Supinski. P^NMPI tools: A whole lot greater than the sum of their parts. In *SC*, page 30, 2007.
10. S. F. Siegel and A. R. Siegel. MADRE: The Memory-Aware Data Redistribution Engine. EuroPVM/MPI 2008
11. S. Vakkalanka, G. Gopalakrishnan, and R. M. Kirby. Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings, *CAV* 2008.
12. S. Vakkalanka, A. Vo, G. Gopalakrishnan, and R. M. Kirby. Reduced execution semantics of MPI: From theory to practice. In *FM*, pages 724–740, 2009.
13. A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, and R. Thakur. Formal verification of practical MPI programs. In *PPoPP*, pages 261–269, 2009.
14. A. Vo, S. S. Vakkalanka, J. Williams, G. Gopalakrishnan, R. M. Kirby, and R. Thakur. Sound and efficient dynamic verification of MPI programs with probe non-determinism. In *EuroPVM/MPI*, pages 271–281, 2009.