A Formal Approach to Detect Functionally Irrelevant Barriers in MPI Programs*

Subodh Sharma[1], Sarvani Vakkalanka[1], Ganesh Gopalakrishnan[1], Robert M. Kirby[1], Rajeev Thakur[2], and William Gropp[3]

[1] School of Computing, Univ. of Utah, Salt Lake City, UT 84112, USA
[2] Math. and Comp. Sci. Div., Argonne Nat. Lab., Argonne, IL 60439, USA
[3] Dept. of Computer Sci., Univ. of Illinois, Urbana, Illinois, 61801, USA

Abstract. We examine the unsolved problem of automatically and efficiently detecting functionally irrelevant barriers in MPI programs. A functionally irrelevant barrier is a set of `MPI_Barrier` calls, one per MPI process, such that their removal does not alter the overall MPI communication structure of the program. Static analysis methods are incapable of solving this problem, as MPI programs can compute many quantities at runtime, including send targets, receive sources, tags, and communicators, and also can have data-dependent control flows. We offer an algorithm called Fib to solve this problem based on dynamic (runtime) analysis. Fib applies to MPI programs that employ 24 widely used two-sided MPI operations. We show that it is sufficient to detect barrier calls whose removal causes a *wildcard receive statement* placed before or after a barrier to now begin matching a send statement with which it did not match before. Fib determines whether a barrier becomes relevant in *any* interleaving of the MPI processes of a given MPI program. Since the number of interleavings can grow exponentially with the number of processes, Fib employs a sound method to drastically reduce this number, by computing only the *relevant interleavings*. We show that many MPI programs do not have data dependent control flows, thus making the results of Fib applicable to all the input data the program can accept.

1 Introduction

The barrier construct (`MPI_Barrier`) is an important function in the MPI library. It is a *collective* call, meaning that all processes in the communicator must call the barrier. We define such a collective call defined by a set of barrier calls (one from each process) to be a *collective barrier*. A collective barrier is *functionally irrelevant* ("*irrelevant*" for short) if its removal does not alter the overall MPI communication structure of the program in terms of correctness and matching of operations. To the best of our knowledge, this problem has not been solved before. In this paper, we present an algorithm called Fib to solve this problem based on dynamic (runtime) analysis for MPI programs employing 24 widely used two-sided MPI operations (detailed on our web page [1]).

The importance of detecting irrelevant barriers comes from a number of perspectives. Many MPI users are known to employ collective barriers for "good

* Supported in part by NSF CNS-00509379, and Microsoft HPC Institutes Program.

measure;" they are unsure whether it is necessary. The authors of [2] narrate the example of an MPI program where a barrier was considered irrelevant, and removed. A year later, they were proven wrong, as a race condition was introduced by its removal. In [3], it is shown that barriers can consume a significant fraction of the total application time. Of course, users wanting to control performance by avoiding network or I/O contention may *insert* collective barriers. In this case, they are employing *functionally irrelevant* barriers for controlling the *non-functional* aspects of their program. The Fib algorithm can help these users by checking that these barriers are indeed functionally irrelevant.

Detecting irrelevant barriers by inspection is not straightforward, as we show through a number of small examples in Section 2. While each example seems to warrant a different justification, a nice feature of the Fib algorithm is that it reduces all these justifications to a single mathematical relation called the *completes-before relation*. This relation has two aspects: intra completes-before (IntraCB), and inter completes-before (InterCB). In a nutshell, the Fib algorithm detects a change in the set of communication possibilities by computing the InterCB relation in the presence of a barrier, and checks whether the barrier plays a role in ordering a send and a wildcard receive.

The examples given in Section 2 do not reflect the following additional difficulties. In realistic MPI programs, a user may forget to use a collective barrier (*i.e.* forget to place a barrier within a process), thus introducing a deadlock. Also, realistic programs may compute many quantities at run time, including send targets, receive sources, tags, and communicators. They also have data-dependent control flows which can determine the actual sends and receives issued. The Fib algorithm works in the presence of all these realities:

• Since Fib is implemented as an extension to the dynamic formal verification methodology employed in our tool ISP([9, 13, 4]), it is capable of detecting deadlocks, and then aborting its analysis. Here are some example deadlock scenarios that ISP can detect: (i) deadlocks due to a collective barrier being incorrectly placed, (ii) those introduced when the user forgets to issue the (supposed) collective call from within some of the processes, (iii) the user employing the wrong communicator for one of the barrier calls, or (iv) MPI messages not matching.

• Since Fib employs dynamic (runtime) analysis, all computed quantities would be fully resolved, and become known. For the same reason, data-dependent control flows are also not an issue for Fib, *in so far as path coverage goes*. It is clear that in general, the behavior of an MPI program can change in response to the input data being analyzed (addressing this issue is considered future research). However, a preliminary static analyzer that we have implemented confirms that for many examples (e.g., all our examples in [1]), control flow does not depend on data; for such programs, the analysis results of Fib are good for *all* input data.

Fib flags a barrier as functionally irrelevant *if and only if* it is functionally irrelevant across *all possible executions* (process interleavings) of the program *for the given input data*. Clearly, we cannot hope to examine all the interleavings of any realistic MPI program naïvely, because this number grows exponentially

with the number of processes. Fortunately, the ISP tool actually generates only a *miniscule fraction* of all possible interleavings, by computing only the *relevant interleavings* of an MPI program using a formal verification method called *partial order reduction* [11, 12]. Without such a reduction algorithm, an algorithm similar to Fib would be difficult to build.

Related Work: Fib is a significant extension of our POE algorithm implemented in the ISP verification tool. The mathematical relation IntraCB is employed in POE (formally defined in [5], but summarized in this paper). The relation InterCB builds on IntraCB, and is brand new to the Fib algorithm, and this paper.

In [6], the authors provide a formal approach for arguing about the relevance of barriers in MPI programs that do not employ wild-card receives. They prove that for *wild-card receive free* MPI programs that are deadlock free, all barriers are irrelevant. This justifies our criterion for relevant barrier detection, which is: *In a deadlock-free program, the removal of a barrier causes a* wildcard receive statement *placed before or after a barrier to now begin matching a send statement with which it did not match before.* The examples in Section 2 provide added insights into our criterion.

The work in [8] uses vector clocks [7], and provides a method for identifying the racing messages in a single trace of an MPI program execution across "frontiers" or *consistent cuts* [7]. While these ideas are somewhat related, the classical vector clock formulation does not directly apply to MPI because of its out-of-order completion semantics and barrier semantics, pointed out in Section 2.

Roadmap: Section 2 provides the intuition behind our Fib algorithm through several examples. The Fib algorithm itself is detailed in Section 3, where we also include sufficient background on the POE algorithm and our ISP tool. Section 4 provides experimental results, and Section 5 provides concluding remarks.

2 Overview of Fib, and the Completes-before Relation

In this section, we present a number of examples, introducing the concepts of IntraCB and InterCB in context. These relations can be assumed to be always maintained in a transitively closed manner. Please note that we omit the prefix `MPI_` in most cases, and also suppress irrelevant arguments of MPI calls. Also for immediate-mode operations, we show a corresponding `Wait` only in some cases.

Example 1: As our simplest example, consider the following single process (rank) MPI pseudo-code program:

```
P0: Irecv(from P0, x, &h); Wait(&h); Barrier; Isend(to P0, 22);
```

In this program, the collective barrier is a singleton set containing `Barrier` from `P0`. Curiously, `P0` is trying to send to itself, which is allowed in MPI. In this case, Fib will report a deadlock whether there is a barrier or not. This is because of an IntraCB edge from `Wait` to any following instruction. An IntraCB edge implies the MPI guarantee of not issuing any instruction after a `Wait` until `Wait` has been issued. In our example, there is an `Isend` after `Wait`, and unfortunately `Wait` cannot finish unless `Isend` finishes—a circular dependency causing the deadlock.

In MPI, there is also an IntraCB edge from a `Barrier` to any following instruction. This means that instructions following the barrier cannot be issued

until the collective barrier can be crossed. Now, suppose we *alter this example* by moving `Wait` to be *after* the `Isend`. In this altered example, `Barrier` can be crossed after issuing `Irecv`, and this leads to `Isend` being issued. Thus, for this altered example, the barrier is **irrelevant**.

Example 2: Here `*` indicates `ANY_SOURCE` (a wildcard receive):

```
P0: Irecv(*, x, &h);  Barrier; Isend(to P0, 22); Wait(&h);
P1: Isend(to P0, 33); Barrier;
```

In this example, it is possible for `x` to attain the value `22`, whether the collective barrier is there or not! This is because even though there *is* an IntraCB edge from `Barrier` to `Isend` in `P0`, there is no IntraCB edge from `Irecv` to `Barrier` in `P0`, and similarly there is no IntraCB edge from `Isend` to `Barrier` in `P1`. Therefore, for this program, Fib will flag the collective barrier as **irrelevant**.

Example 3: Consider the program:

```
P0: Irecv(*, x, &handle); Barrier; Wait(&handle);
P1: Isend(to P0, 33);     Barrier; Isend(to P0, 22);
```

Here, the collective barrier is indeed **irrelevant**, and will be flagged as such by the Fib algorithm, following this line of reasoning: (i) the first `Isend` of `P1` and the `Irecv` of `P0` can be issued; (ii) the `Barrier` in the respective processes can be crossed, as there is no IntraCB edge to these `Barrier`s; (iii) before `Irecv` occurs, `Isend(to P0, 22);` can also be issued; (iv) however, MPI's message-matching rules require process-to-process FIFO message ordering; in other words, there is an IntraCB edge from the first `Isend` to the second `Isend` in `P1`. Therefore, `x` can attain the value of `33` only.

Example 4: In contrast with Example 3, in this program, we move the second send to process `P2`:

```
P0: Irecv(*, x, &handle); Barrier; Wait(&handle);
P1: Isend(to P0, 33);     Barrier; ...rest of P1...
P2: ...some code...       Barrier; Isend(to P0, 22);
```

The `Isend`s are in different processes. Therefore, there is no IntraCB ordering between them. However, the `Irecv` of `P0` as well as `Isend` of `P1` also do not have an IntraCB to their barriers. Therefore, the collective barrier is **irrelevant**.

Now consider an alternative example (call it **Example 4(a)**) in which the `Wait` in `P0` is moved to be *before* its `Barrier`. Now, the collective barrier becomes **relevant**. This is because there would be an IntraCB edge from `Wait` to `Barrier`. Hence, `Barrier` cannot be crossed until the `Irecv` finishes. Therefore the `Isend` from `P2` cannot issue. Therefore, `Irecv` has to finish based on the `Isend` from `P1`.

InterCB: The reasoning employed in this example highlights the need for the notion of *InterCB* edges. Basically, the `Isend` of `P2` "wishes to match" the `Irecv` of `P0`. The only thing that prevents this is that the collective barrier orders `Irecv` to be before it, and `Isend` to be after it. This is the ordering defined by InterCB (detailed in Section 3). Furthermore, there is no alternative ordering path starting from this `Irecv` to `P2`'s `Isend` that does not involve a barrier. Hence the barrier is **relevant**.

Example 5: In all previous examples, the wildcard receive statement appeared before a barrier. In this example, it appears afterwards:

```
P0:                 Barrier; Send(to P2);
P1: Send(to P2);    Barrier;
P2: Irecv(from P1); Barrier; Recv(*, ..);
```

Here, the barrier is **irrelevant** because `P2`'s `Irecv(from P1)` is ordered before `Recv(*)`. The reasoning now relies on another fact about MPI. If there is a specific-source nonblocking receive followed by a wildcard receive in an MPI program, the wildcard receive can *trump* the specific receive (i.e. may match before it), *if there is no matching sender to the specific-source receive*! In Example **5**, however, there *is* a matching `Send(to P2)` in `P1`, and so trumping does not happen. Since there is no trumping, the IntraCB ordering is maintained between `Irecv(from P1)` and `Recv(*,..)`.

3 The Fib Algorithm

We now provide an overview of the POE algorithm used in our ISP tool (Section 3.1), and then describe the Fib algorithm (Section 3.2).

3.1 POE Overview

The crucial idea embodied in POE is the notion of exploring only *relevant* interleavings—a technique known in model checking as *partial order reduction* [11]. Without this idea, any exploration method for MPI will go out of hand. For instance, consider **Example 6** below (used to illustrate the POE algorithm, and not used to illustrate Fib) that begins with two sends in P0 and P2, and a wildcard receive in P1. The total number of interleavings of all these MPI calls is 210.[4] However, to trigger `error1`, we need to consider the interleavings in which the send of `P2` matches the wildcard receive. Testing oriented tools may easily miss these interleavings. Thanks to partial order reduction, POE will: (i) pick an arbitrary order for executing `P0`'s first send and `P1`'s first receive, (ii) pick an arbitrary order to execute `P2`'s first send and `P1`'s second receive, and then (iii) consider *both* the `Send` matches with the wildcard receive (shown by `*`). In other words, POE will examine only two interleavings.

Example 6:

```
P0: MPI_Send(to P1...); MPI_Send(to P1, data = 22);
P1: MPI_Recv(from P0...); MPI_Recv(from P2...);
    MPI_Recv(*, x); IF (x==22) THEN error1 ELSE MPI_Recv(*, x);
P2: MPI_Send(to P1...); MPI_Send(to P1, data = 33);
```

POE has built-in knowledge of the commuting properties of MPI functions. As another example, consider an MPI program in which `MPI_Barrier` is invoked by N processes. POE would, in general, explore only one of the $N!$ ways in which to have invoked the barrier calls. In our implementation of the 24 MPI functions, the cases where alternate interleavings are to be explored include wildcard receives, `WAIT_ANY`, and `TEST_ANY`.

[4] (7!)/((2!).(3!).(2!))

Overview of POE's use of PMPI: POE uses the well-known PMPI mechanism. It introduces an extra process called the *verification scheduler*. POE provides its own version of "MPI_f" for each MPI function f. Within MPI_f, POE arranges for handshakes with the scheduler that realizes the POE algorithm. When the scheduler finally gives permission to fire f, POE invokes PMPI_f. The MPI runtime only sees the PMPI_f calls.
Match Sets: Consider Example **6** above. If one repeatedly runs this example under MPICH2 (for example), it is not guaranteed that `error1` will be caught. In other words, the matching of `Send` from `P0` with the wildcard receive in `P1` may prove *elusive*, as the MPI runtime may never schedule this match!

POE solves the problem of elusive matches *without requiring changes to the MPI library* and *without adding padding delays*—these are expensive, and/or brittle solutions. Instead, it *dynamically rewrites* wildcard receives into specific receives, one for each actual sender that, it computes to be a *certain* match. In the context of **Example 6**, if we can rewrite `Recv(*)` to `Recv(from P0)` and `Recv(from P1)`, in turn, and: (i) pair the first one with `Send(to P1, data=22)`, and (ii) pair the second one with `Send(to P1, data=33)`, in turn, and (iii) issue only these send/receive pairs into the MPI system, we can force these matches to occur. Such groupings of sends and receives are called *match sets* in POE's parlance. In POE, in addition to match sets obtained by grouping dynamically rewritten wildcard receives with their matching sends, (i) point to point sends and their receives also form match sets, and (ii) a collective barrier also form match sets.

3.2 Fib Algorithm

The Fib algorithm can be expressed through the following steps:

- Run the POE Algorithm
  - For each interleaving explored by POE
  - let result_list.append(CheckRelevant());
  - End For
- if result_list == empty then print all the barriers as **irrelevant**
- else print result_list as **relevant**

The CheckRelevant algorithm can be expressed through the following steps:

- Consider a Match Set with events x and y (in general, it will have N events; we take two events for illustration)
- If there is any IntraCB edge from x to any operation *op*, then introduce an InterCB edge from y to *op*. Repeat this till no InterCB edges can be added starting from y.
- Do the above step for all the operations in the match set.
- Now, consider the set $\{R, S\}$ where R is derived from a wildcard receive (through rewriting) and S is a send that targets the process of the receive.
- If there exists a path using InterCB and IntraCB edges from R to S going through a barrier B, and *there is no alternate ordering path* going from R to S, flag the barrier as **relevant**.

  - Add the found relevant barrier to the local_result_list.
  - Add the rest of the barriers that are in the match set of the barrier found in the above step also to local_result_list.
- Do the above step with R and S swapped.
- Repeat the process until all such $\{R, S\}$ sets are evaluated.
- Return local_result_list.

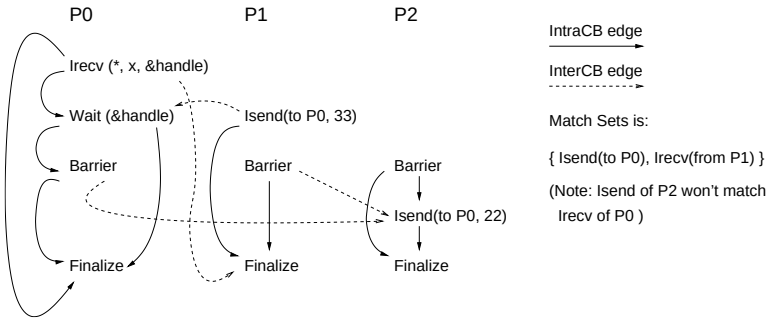

Fig. 1. Example 4(a) in Section 2 with InterCB and IntraCB edges

Illustration: In Example 4, there is no InterCB ordering from `Irecv` to the `Isend` of `P2`. Now in the alternate example called **Example 4(a)** discussed earlier, the above procedure will end up creating an IntraCB path from `Irecv` to `Wait` to `Barrier` in `P0`. Also, all of `Barrier` also form a match set. Furthermore, `Isend` of `P2` is ordered to be after the `Barrier`. There is no alternate ordering path – so the collective barrier is relevant. Figure 1 summarizes the above explanation. The IntraCB edges depicted in Figure 1 for process P0 are easy to reason. In process P1, `Isend(to P0)` has no IntraCB edge to the following `Barrier` since Isend being a non blocking call has no obligation to finish before the barrier. However, since Fib knows that `Irecv(*)` in P0 matches this Isend, we add InterCB edges from Isend to operations that are bound to complete *after* Irecv. This explains the InterCB edge between `Isend( to P0)` to `Wait(&handle)`. The same reasoning explains the InterCB edge from `Irecv(*)` to `Finalize` of P1. After adding InterCB edges, the only path that reaches to `Isend(to P0)` of P2 from `Irecv(*)` of P0 involves a barrier. Thus the barrier and all the barrier operations from other processes that formed the match set are flagged to be relevant.

4 Implementation and Experimental Results

We automatically instrument the MPI user code where all `MPI_Barrier(comm)` calls are replaced by `MPI_Barrier_new(comm,__LINE__, __FILE__)`. The two new arguments are system macros that keep the information of line number the

function call and the file name that contains it. Our instrumentation tool is written using CIL [15] which offers a framework to create a custom source-to-source program-instrumentation pass. We have run our Fib tool on several MPI programs including: (i) the Monte-Carlo computation of *Pi*, (ii) 2D diffusion, and (iii) all 69 tests that came along with UMPIRE tool [10]. As for runtimes, the ISP algorithm introduces a slowdown because of its scheduler-mediated executions (in [13], we provide ideas for improving the execution time). The *added* overhead that Fib introduces over and above ISP is negligible. Our web page [1] provides detailed results; here is a summary:

- **Monte-Carlo**: The code of Monte-Carlo, did not have any barrier calls. To acid-test our implementation, we deliberately inserted an irrelevant collective barrier, which our implementation flagged as such. The run times of the Fib algorithm are as follows: (i) with 4 processes, it explored 6 interleavings in 0.2 seconds, and with 5 processes, it explored 24 interleavings in 1.52 seconds.
- **2D Diffusion** This code had 2 irrelevant barriers which were caught by the tool. In fact, this example does not employ wildcard receives, and so all its barriers are irrelevant, and Fib finishes with one interleaving. The runtime of Fib on this example was less than a second. This reinforces that without wildcards we need only one interleaving.
- **Umpire test suite**: We ran our tool successfully on all the 69 tests that came along with Umpire tool [10]. Of the 36 tests that had barriers, all were flagged as **irrelevant**, with negligible runtimes.

5 Concluding Remarks

Removing unnecessary barriers is important, because they needlessly add to the program-execution time. This is particularly true for applications running on petascale machines with thousands of processors. We presented an algorithm, Fib, that is built as an extension to our verification tool ISP for MPI programs. Fib works by detecting, for each barrier, whether its removal causes a *wildcard receive statement* placed before or after a barrier to now begin matching a send statement with which it did not match before. We report success in detecting irrelevant barriers in a number of examples. Since all these examples have control that does not depend on data, the analysis is good for all input data. Our future plans include extending this analysis to cover interesting classes of data dependent control, as well as aiming to cover all of MPI 2.0.

Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

References

1. http://www.cs.utah.edu/formal_verification/europvm-mpi08/FIB
2. G. S. Avrunin, S. F. Siegel, and A. R. Siegel. Finite-state Verification for High Performance Computing. Proc. Second Intl. Wkshp. on Soft. Eng. for High Perf. Computing Syst. Apps. 2005, pages 68–72.
3. Rolf Rabenseifner. Automatic MPI Counter Profiling. In proceedings of the 42nd Cray User Group Conference, CUG SUMMIT 2000, May 22-26, Noorwijk, The Netherlands.
4. , Sarvani Vakkalanka, Michael DeLisi, Ganesh Gopalakrishnan, Robert M. Kirby, Rajeev Thakur, and William Gropp. Implementing Efficient Dynamic Formal Verification Methods for MPI Programs. (In this proceeding - EuroPVM/MPI 2008.)
5. Sarvani Vakkalanka, Ganesh Gopalakrishnan, and Robert M. Kirby. Dynamic verification of mpi programs with reductions in presence of split operations and relaxed orderings. In *Computer Aided Verification*, 2008. Accepted. ,
6. Stephen F. Siegel and George S. Avrunin. Modeling Wildcard-free MPI Programs for Verification. PPoPP, 2005. pages 95–106.
7. Friedemann Mattern. Virtual Time and Global States of Distributed Systems. Parallel and Distributed Algorithms: Proc. Intl. Wkshp. Par. and Dist. Algo, 1989.
8. R. H. B. Netzer and B. P. Miller. Optimal Tracing and Replay for Debugging Message Passing Parallel Programs. Supercomputing, pages 502–511, 1992.
9. S. Pervez et.al. Practical model checking method for verifying correctness of MPI programs. *EuroPVM/MPI*, pages 344–353, 2007.
10. Jeffrey S. Vetter and Bronis R. de Supinski. Dynamic Software Testing of MPI Applications with Umpire. *Proc. of SC2000*, pages 70–79, 2000.
11. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.
12. Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. *POPL*, pages 110–121. ACM, 2005.
13. Sarvani Vakkalanka, Subodh V. Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby. ISP: A tool for model checking MPI programs. PPoPP 2008. 285-286.
14. W. D. Gropp and E. Lusk. Using MPI-2: A Problem-based Approach. EuroPVM/MPI 2007 Tutorial.
15. G. C. Necula et.al. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. *CC*, pages 213–228. Springer-Verlag, 2002.