PREDICTIVE ANALYSIS OF MESSAGE PASSING APPLICATIONS

by

Subodh Sharma

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

School of Computing

The University of Utah

March 2012

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Subodh Sharma

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

	Chair: Ganesh Gopalakrishnan
	Matt Might
	Chris Myers
	Eric Mercer
	Greg Bronevetsky

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of Subodh Sharma in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Ganesh Gopalakrishnan
Chair, Supervisory Committee

Approved for the Major Department

Alan Davis
Chair/Dean

Approved for the Graduate Council

Charles A Wight
Dean of The Graduate School

ABSTRACT

Message passing (MP) has gained a widespread adoption over the years, so much so, that even heterogeneous embedded multicore systems are running programs that are developed using message passing libraries. Such a phenomenon is a shift in computing practices, since, traditionally MP programs have been developed specifically for high performance computing. With growing importance and the complexity of MP programs in today's times, it becomes absolutely imperative to have formal tools and sound methodologies that can help reason about the correctness of the program.

It has been demonstrated by many researchers in the area of concurrent program verification that a suitable strategy to verify programs which heavily rely on non-determinism, is dynamic verification. Dynamic verification integrates the best features of testing and model-checking. In the area of MP program verification, however, there have been only a handful of dynamic verifiers. These dynamic verifiers, despite their strengths, suffer from the explosion in execution scenarios. All existing dynamic verifiers, to our knowledge, exhaustively explore the non-deterministic choices in an MP program. It is apparent that an MP program with many non-deterministic constructs will quickly inundate such tools.

This dissertation focuses on the problem of containing the exponential space of execution scenarios (or interleavings) while providing a soundness and completeness guarantee over safety properties of MP programs (specifically deadlocks). We present a *predictive verification* methodology and an associated framework, called MAAPED(**M**essaging **A**pplication **A**nalysis with **P**redictive **E**rror **D**iscovery), that operates in *polynomial time* over MP programs to detect deadlocks among other safety property violations. In brief, we collect a single execution trace of an MP program and *without re-running other execution schedules*, reliably construct the artifacts necessary to predict any mis-happening in an un-explored execution schedule with the afore-mentioned formal guarantee.

The main contributions of the thesis are the following:

- The Functionally Irrelevant Barrier Algorithm to increase program productivity and ease in verification complexity.

- A sound pragmatic strategy to reduce the interleaving space of existing dynamic verifiers which is complete only for a certain class of MPI programs.
- A *generalized Matches-Before* ordering for MP programs.
- A *predictive polynomial time verification framework* as an alternate solution in the dynamic MP verification landscape.
- A soundness and completeness proof for the predictive framework's deadlock detection strategy for many formally characterized classes of MP programs.

In the process of developing solutions that are mentioned above, we also collected important experiences relating to the development of *dynamic verification schedulers*. We present those experiences as a minor contribution of this thesis.

CONTENTS

ABSTRACT . . . . . **ii**

LIST OF FIGURES . . . . . **vii**

LIST OF TABLES . . . . . **ix**

LIST OF ALGORITHMS . . . . . **x**

LIST OF SYMBOLS . . . . . **xi**

ACKNOWLEDGEMENTS . . . . . **xii**

CHAPTERS

1. INTRODUCTION **1**

1.1 Dynamic Verification of Message Passing Programs 2
1.2 Thesis statement 3
1.3 Contributions of this dissertation 3
1.3.1 Dynamic Program Analysis for Performance 3
1.3.2 Safe Reduction of Persistent Sets for MPI programs 4
1.3.3 Predictive Verification Framework for MPI programs 4
1.3.4 A Dynamic Verification Scheduler for MCAPI Programs 5
1.4 Organization of the Dissertation 5

2. BACKGROUND **7**

2.1 Message Passing Interface (MPI) 7
2.1.1 Notation for MPI Calls 11
2.1.2 Non-determinism in MPI 11
2.1.3 Common Bugs in MPI 12
2.2 Details of ISP 14
2.2.1 MPI Correctness Guarantee and the Matches-Before Ordering 14
2.2.2 ISP's Profiler, Scheduler and the POE Algorithm 16
2.2.2.1 *Profiler* 16
2.2.2.2 *Scheduler* 16
2.2.2.3 *POE* 17
2.2.2.4 *Dynamic re-writing* 19
2.2.3 Notations for IntraMB Ordering 20

3. FUNCTIONALLY IRRELEVANT BARRIERS IN MPI APPLICATIONS 22

3.1 Introduction 22
3.2 Overview of FIB, and the InterMB Relation 24
3.3 InterMB relation 27
3.4 Matches-Before Relation 28

3.5 The Functionally Irrelevant Barrier (FIB) Detection Algorithm 28
3.6 Correctness Proof 30
3.7 Implementation and Experimental Results 31
3.8 Summary 32
3.8.1 Discussion 32

4. PERSISTENT-SET REDUCTION HEURISTIC FOR MPI PROGRAMS 33

4.1 Introduction 33
4.2 Preliminaries 37
4.2.1 Nature of Transitions in a Persistent-set 38
4.3 Formal Definition of Independent Transitions 39
4.4 MSPOE Algorithm 41
4.5 Experimental Results 44
4.5.1 MSPOE for Identifying FIBs 47
4.6 Discussion 47
4.7 Conclusions 49

5. GENERALIZED MATCHES-BEFORE RELATION 50

5.1 Introduction 50
5.2 Preliminaries 52
5.3 Potential Match (M^o) Relation 55
5.4 Wait-for (W) Relation 60
5.5 Potential Match (M^o) Relation Refinement 62
5.6 Proof of Correctness 64
5.7 Conclusion 70

6. A PREDICTIVE POLYNOMIAL TIME DEADLOCK DETECTION ALGORITHM FOR MESSAGE PASSING APPLICATIONS 71

6.1 Introduction 71
6.2 Deadlock Detection Rules 73
6.3 Correctness Proof 76
6.4 Complexity Analysis 77
6.5 Messaging Application Analysis with Predictive Error Discovery (MAPPED) 78
6.6 Results 78
6.7 Discusson 80
6.8 Conclusions 80

7. MCC: A DYNAMIC VERIFICATION SCHEDULER FOR MCAPI APPLICATIONS 81

7.1 Introduction 81
7.2 Overview of MCAPI 82
7.3 Verification of MCAPI User Applications 83
7.4 MCAPI Checker (MCC) Overview 86
7.4.1 MCC Scheduler Explanation Through an Example 87
7.4.2 MCC Scheduler Algorithm 90
7.5 Results and Concluding Remarks 93

8. RELATED WORK **94**
8.1 Correctness and Verification Tools in MPI 94
8.2 Tools for Checking MCAPI Applications 96
8.3 Related Work in Barrier Analysis 97

9. CONCLUSIONS AND FUTURE DIRECTIONS **98**
9.1 Future Research Directions 99
9.1.1 Proof for the Conjecture 99
9.1.2 A Static Analysis Framework for Synergistic Static-Dynamic Analysis . 99
9.1.3 Task Permutation vs Match Permutation 99
9.1.4 Verification for Performance 100
9.1.5 Hybrid Program Verification 101

REFERENCES **102**

LIST OF FIGURES

1.1 MPI example to illustrate the deadlock (*Heisenbug*) 2
2.1 An MPI program with Master-Slave communication pattern 8
2.2 Deadlock due to send receive mismatch 13
2.3 Head-to-head deadlock 13
2.4 Deadlock due to nondeterministic receive 13
2.5 Deadlock due to collective call order mismatch 13
2.6 Crooked Barrier: Issue order vs. Match Order 14
2.7 MB ordering for S and R 16
2.8 MB ordering between R* and R 16
2.9 Conditional MB ordering 16
2.10 Overview of ISP tool 17
2.11 Example explaining POE 19
3.1 InterMB relation w.r.t the match-sets 28
3.2 FIB framework 29
3.3 Example 4(a) in Section 3.2 with InterMB and IntraMB edges 29
4.1 Deadlock free example 35
4.2 State graph for Figure 4.1 35
4.3 Deadlocking example 36
4.4 Possibilities after first R(*) match 36
4.5 Dependence among DTG transitions 38
4.6 Commuting example 40
4.7 Transition independence 40
4.8 MSPOE with redundant exploration 42
4.9 Communication in 2D-Diff 45
4.10 Deadlock because cyclic dependency between $S_{1,2}$ and $S_{2,1}$ 48
4.11 Deadlock because barriers do not discharge 48
5.1 Example illustrating inconclusiveness of InterMB ordering 51
5.2 Example illustrating ordering enforced by deterministic operations 51

5.3 Deterministic Recv pinning a send 56
5.4 Upward and downward M^o edges 57
5.5 F-rule and L-rule 59
5.6 Complete M^o graph of example in Figure 5.4(a) 60
5.7 Figures demonstrating the three parts in S-rule 61
5.8 Condition for introducing Wait-for from a wildcard recv 62
5.9 Wait-for edges introduced due to S and R rules with complete M^o graph 63
5.10 Refinement due to a Wait-for edge 63
5.11 Final M^o of the example from Figure 5.9 65
5.12 Corealizability of M^o edges 69
6.1 Example with buffer dependent deadlock 72
6.2 Deadlock due to Wait-for on Send 72
6.3 Orphaned deterministic Receive scenario 74
6.4 Example illustrating a deadlock despite No Wait-for dependencies 74
6.5 MAAPED Workflow 78
6.6 DTG-deadlock program trace 80
7.1 An instrumented MCAPI example C program 84
7.2 MCAPI Receive Nondeterminism 85
7.3 MCC workflow 87
7.4 Re-ordering Example 88
7.5 Interactions of the scheduler with the example from Figure 7.4 89
9.1 Match Permutation vs Task Permutation 100

LIST OF TABLES

4.1 Interleaving results for deadlock detection 44
4.2 FIB results with MSPOE 47
5.1 Computation of C, E, K and D details 58
6.1 Results for deadlock detection via predictive verification 79

LIST OF ALGORITHMS

1 COMPUTE-FIB 30
2 PATHS 30
3 MSPOE Algorithm 42
4 GenerateInterleaving from state s 43
5 Choose P_s 43
6 ComputeKPAncs 54
7 M-W Construction 65
8 Discharge Algorithm 75
9 FindEnabledSends Algorithm 75
10 MCC scheduler pseudocode 90
11 Find a suitable match-set 91

LIST OF SYMBOLS

$S_{i,l}$ Nonblocking Send 11
$R_{i,l}$ Nonblocking Recv 11
$W_{i,l}$ Wait call 11
$B_{i,l}$ Barrier call 11
$\prec_{lp}$ IntraMB ordering 20
$Op^{\ll}$ Ancestor operation of Op 20
$Op^{<}$ Immediate Ancestor of Op 20
$Op^{\gg}$ Descendant operation of Op 20
$Op^{>}$ Immediate Descendant of Op 20
$\prec$ Total ordering among match-sets 21
$\prec_{ip}$ InterMB ordering 27
$\prec_{mb}$ Matches-Before ordering 28
$\hat{Op}$ Operation sequence 28
$\equiv_t$ Type equality 52
$\equiv_{t,d}$ Type-Target equality 53
$Op^{\ll_k}$ K many ancestors 53
$Op^{\gg_k}$ K many descendants 53
$Op^{\ll_{k,p}}$ K many ancestors satisfying p 53
$Op^{\gg_{k,p}}$ K many descendants satisfying p 53
$F^j(Op_i)$ First from P_j 59
$L^j(Op_i)$ Last from P_j 59
$\prec_w$ Wait-for ordering 62

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Professor Ganesh Gopalakrishnan, whole heartedly for the constant support and advice that he has generously provided me over the duration of my study. His enthusiasm in research is a great source of inspiration for all his students including myself. Ganesh is the best advisor one can hope to be supervised by.

I would also like to thank all my committee members for their constant support and excellent suggestions without which completing this dissertation would not be possible. I also have had the opportunity to work closely with few of my committee members (Eric and Greg) and they are one of the smartest and most helping people I have come across. Thanks Eric and Greg. Working with you was an enriching experience. I am especially grateful to Prof. Robert M. Kirby for initiating me in to the Gauss verification group and guiding me in my early years of the program.

I also express my gratitude to all the present and past Gauss group members, specially Sarvani Vakkalanka, Anh Vo, Yu Yang, Sriram Aanantakrishnan, Michael Delisi, Geof Saway, and Wei-fan Chiang for the discussions that I have had with them in the past relating to this dissertation.

Last but not the least, finishing this dissertation would not have been possible without the love and support of my parents and my siblings. They will always have my gratitude. I would also like to express my gratefulness to my brother-in-law who has always been a source of inspiration. I am also thankful to a lot of my friends who made my stay in Salt Lake City comfortable and cherishable. I especially thank Manu Awasthi and Amlan Ghosh for making my graduate school experience memorable.

CHAPTER 1

INTRODUCTION

Parallel computing has become ubiquitous. Each year, we witness the arrival of more powerful supercomputers and parallel platforms that outperform their predecessors. The need to simulate larger problems with increased performance requirements is not the only reason for propelling parallel computing into such ubiquity. Even low powered embedded hand-held devices are also increasingly adopting parallel computing. The expectation to see lesser response times and higher throughput on the devices is leading to such widespread adoption of parallelism at all component (hardware/software) of computing. At a software layer, parallel computing can be realized by writing programs that are run on multiple processes/threads wherein the participating processes/threads communicate either by share memory (multithreading) or via explicit messages. In the domain of MP (Message Passing), the most successful and widely adopted standard for library implementation is MPI (Message Passing Interface [44]). The *extreme scale computing* roadmap [53, 21] clearly indicates that both, shared-memory and a standard such as MPI, are essential and must coexist in order to achieve the goal of exascale computing, thus, reaffirming what many believe, that MPI is not dead yet. The work in this dissertation focuses on programs written using MPI and the Mutlicore Communications APIs (MCAPI [42])

It is a widely accepted fact that writing *correct* parallel programs is difficult. Even if we concentrate on the correctness of reactive aspects of the program (such as absence of deadlocks, races, etc.), reasoning about program correctness still remains an arduous task. The primary reason for the difficulty in constructing correct parallel programs is the unexpected ways in which participating processes of the program interact leading to exponentially vast number of execution scenarios that an application developer must visualize. Such an expectation (to be able to visualize all possible program interpretations) is unreal. These unanticipated interactions are due to the non-deterministic constructs employed by application developers while developing the program. Such interactions are a big source of worry, since, it is possible that conventional ad-hoc testing only explores a

P_0	P_1	P_2
`Send(to` P_1`,`d_1`)`	`Recv(from:*,x)`	`Send(to` P_1`,` d_2`)`
	`Recv(from:*,y)`	
	`if(x==`d_2`) ERROR`	

Figure 1.1. MPI example to illustrate the deadlock (*Heisenbug*)

segment of the schedule space which may never expose the bug. However, porting the code to a different machine architecture or running the program under a different environment may manifest the bug. Such bugs are also called as *Heisenbugs* [33]. Figure 1.1 illustrates a simple MPI example where a deadlock is present as a *Heisenbug*. Note that for simplicity we have only synchronous sends and receive operations in the example. Observe that `Send` from P_0 is racing with `Send` from P_2 for the first receive from P_1. Further assume that the data payload d_2 is very large in size as opposed to d_1. Under traditional testing, one may never discover the bug because `Send` from P_0 would always reach its destination before `Send` from P_2. However, under certain unusual conditions where the network latency is high on the $P_0 - P_1$ line, we may witness `Send` from P_2 racing ahead, thus, exposing the bug.

Unfortunately, existing ad-hoc testing/debugging methodologies [38, 66, 82, 47, 13] fall vitally short to examine programs where bugs are deep-seated. Pursuing formal verification strategy is the only plausible solution to validate such parallel programs. There are many ways to formally validate parallel programs, viz. static analysis, model checking, and dynamic verification. Static analysis can validate all possible program interpretations independent of the input, however, there is a possibility that any imprecision in the analysis may produce false alarms. Attaining high precision in a scalable manner is still a area of active research in this domain. While model checking methodology offers the coverage guarantee without producing false-alarms, the effort to model large real code-bases in a modeling language is often a laborious and a error prone task. Dynamic verification, is a choice that offers some of the better benefits. Dynamic verification integrates the best features of testing (ability to directly run the programs) and model-checking (coverage guarantees). This dissertation focuses on creating efficient dynamic verification algorithms for MP programs.

1.1 Dynamic Verification of Message Passing Programs

There has been a considerable body of work on developing state-of-the-art debugging and visualization methodologies/tools for MP (specifically MPI) programs [38, 13, 74, 66].

However, for the reasons elicited in the previous section, such tools fall short in validating parallel program with non-determinism. While schedule perturbation methods such as [80] enhance the likelihood that alternate execution paths are taken, very often such techniques lack the fine control necessary to actually affect the send/receive matches in an MPI program. Tools like MPI-SPIN [60] are the first to provide model-checking based solutions in the MPI program verification landscape. MPI-SPIN is built by extending the SPIN [35] language and tool. There have also been tools that perform symbolic analysis of MPI programs [63] written for scientific applications, however, such tools suffer from a common problem of the blowup in the constraint formula. Moreover, they are geared to show functional equivalence of scientific software which is a solution to a different problem altogether.

In the area of formal dynamic verification of MPI programs, ISP [69, 78, 72, 67] and DAMPI [76, 75] are the known tools that perform exhaustive exploration of the non-deterministic schedule space of the program. For the purpose of this dissertation, we choose ISP as the baseline, however, our algorithms are very well applicable to DAMPI. ISP is a centralized verification scheduler and DAMPI is a distributed verification scheduler, both of which generate the *relevant schedule space* of MPI programs and exhaustively explore such a space by *repeatedly executing the program with a fixed input under the control of the verification scheduler* that orchestrates different interleavings in each separate run. Irrespective of whether the verification scheduler is centralized or distributed, we believe that there is a substantially large class of MPI programs for which exhaustive verification is not necessary. Through, this dissertation we demonstrate that to be the fact.

1.2 Thesis statement

Building a predictive dynamic verification framework that can circumvent the exponential schedule space search problem and yet provide the coverage guarantee over certain safety properties, is feasible and novel.

1.3 Contributions of this dissertation

1.3.1 Dynamic Program Analysis for Performance

The result of my initial efforts in understanding the ISP scheduler led to the construction of a dynamic algorithm that detects the presence *functionally irrelevant barriers* [57] (FIB) in an MPI program. A barrier whose removal does not alter the communication structure of

the program is defined to be functionally irrelevant. Note that MPI barriers unlike shared-memory barriers have weaker semantics. MPI barriers enforces an ordering constraint on operations appearing after the barrier as opposed to shared-memory barriers which enforce an ordering constraint on memory operations before and after the barrier.

Often application developers employ barriers for good measure; they are unsure whether a barrier is indeed necessary. Sometimes barriers are also inserted to avoid network or I/O contention. There removal not only increases the parallelism in the application but also eases the verification complexity. Any dynamic verification scheduler (centralized or otherwise) would be able to run the application faster under its orchestration.

The FIB algorithm is implemented on top of the ISP scheduler. Since it is tightly coupled to the ISP scheduler, FIB algorithm could successfully scale up to MPI programs running on ~ 30 processes. However, for FIB to scale to larger problem sizes, successful strategies must be devised to contain the exploding schedule space. This served as the motivation for the next piece of my dissertation work.

1.3.2 Safe Reduction of Persistent Sets for MPI programs

After evaluating number of MPI benchmarks, we believe that barriers in MPI programs are almost never textually unaligned and their issuance is not dependent on the input. Hence w.r.t a certain safety property (absence of deadlocks in our case) we can safely reduce the persistent-set for non-deterministic receive call thereby effectively pruning the schedule space. This piece of work served as a motivation for my subsequent work that forms the basis of the title of this dissertation work. We realized that the current strategy of reducing persistent sets works on a very restricted class of programs and we would ideally want to device a strategy by relaxing the afore-mentioned constraint.

1.3.3 Predictive Verification Framework for MPI programs

- **Generalized Matches-Before relation** An MPI call can exist in one of the multiple states of existence after its issuance. Either the call is simply *enabled* but not matched yet, or the call has found its match but not completed yet, or the call has successfully completed. Knowing precisely when the call has completed would need probes in to the runtime which often the communication libraries provide in the form of *Wait* and *Test* functions. However, after formally studying the call semantics, it was demonstrated that *call issue order* or *call completion order* are far from true ordering

among operations. We build upon the established *Matches-Before* relation in [67, 75] and extend it to a more generalized form.

- **Polynomial deadlock detection algorithm**

  We provide the first novel predictive polynomial time deadlock detection algorithm for MPI programs that do not have input dependent communication flow in the program. A large class of MPI programs fall under this category. We further demonstrate that the artifacts constructed in this predictive framework can also be utilized for a cheaper predictive FIB analysis. Finally, we present the soundness and completeness proof (refer [64] for the definition of soundness[1]) of our deadlock algorithm which depends on the completeness of the potential match relation and the generalized matches-before relation that we construct early on. We conjecture that the generalized Matches-Before ordering and the potential match relation construction is complete and provide a proof sketch for it.

1.3.4 A Dynamic Verification Scheduler for MCAPI Programs

We also developed a dynamic formal verifier for MCAPI application which reinforced our understanding of *Matches-Before* relation and the exposed us to various forms of non-determinism in different flavors of MP libraries. We present the experiences in building a dynamic verifier for MCAPI applications as another contribution of this dissertation with the focus on answering the following questions:

- What consideration one must make in order to build a non-intrusive dynamic verification scheduler?
- What solutions can be attempted in order to have a deterministic replay capability under the presence of non-determinism?

1.4 Organization of the Dissertation

This dissertation is organized as follows: Chapter 2 introduces an overview of MPI and presents some relevant facts about ISP tool on which some of the subsequent work is based on. Chapter 3 presents the FIB algorithm and Chapter 4 presents the strategy to perform safe persistent-set reduction in ISP. Chapter 5 defines potential match-graph

[1]The definitions of soundness and completeness used by the researchers in the Abstract Interpretation landscape are different from what we use in the bug-hunting literature.

and presents a generalized Matches-Before relation. Chapter 6 presents a polynomial deadlock detection strategy (based on the artifacts discussed in the previous chapter) along with the *soundness and completeness proof*. Chapter 7 presents some of the findings we collected while developing the MCC (Multicore Checker) for MCAPI applications. We finally conclude and discuss the future directions in the Chapter 9.

CHAPTER 2

BACKGROUND

The work described in Chapter's 3, 4 is built upon the ISP tool. In this chapter, we provide a brief introduction to MPI along with a succinct description of the ISP tool.

2.1 Message Passing Interface (MPI)

MPI is a library interface specification designed to primarily help application developers write scalable and portable HPC (high performance computing) software. Almost all supercomputers and clusters of today run software written using MPI. It would not be incorrect to say that MPI is a *lingua-franca* of HPC software.

MPI library comes with C/C++ and Fortan bindings. MPI provides *synchronous* and *asynchronous communication* primitives and further classifies communication type as either *point-to-point* or *collective communication*. For a detailed report on MPI, readers are encouraged to refer to [44]. For illustrative purposes, we would limit all future discussions in this dissertation to the following MPI operations: `Send`, `Recv`, `Barrier` and `Wait`. All MPI calls must be gated within `MPI_Init` and `MPI_Finalize` calls as illustrated in Figure 2.1 Failure to comply will result in a compilation error. We will assume that all examples provided in this dissertation have followed the correct rules of writing the MPI program. Thus, to make the presentation easier, we will skip showing `MPI_Init`, `MPI_Finalize` call. Furthermore, we will only show relevant arguments to the calls, when necessary. Figure 2.1 illustrates a simple example with master-worker configuration. Such a communication pattern is widely witnessed in MPI applications. Once a MPI call after it has been issued, it can exist in only one of the following states:

- *Enabled*: The call has been issued by the process but is yet to be matched by the MPI runtime

- *Matched*: The call has been issued and matched with a compatible operation by the runtime, however, the calls have not *completed*.

```
#include <stdio.h>
#include "mpi.h"

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

    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    if (rank == 0)
       master();
    else
       slave();

    MPI_Finalize( );
    return 0;
}

int master()
{
    int        i,j, size;
    char       buf[256];
    MPI_Status status;

    for (i=1; i<size; i++) {
      MPI_Recv( buf, 256, MPI_CHAR, i, 0, master_comm, &status );
    }
    return 0;
}

int worker()
{
    char buf[256];
    int  rank;

    MPI_Comm_rank( comm, &rank );
    MPI_Send( buf, strlen(buf) + 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    return 0;
}
```

Figure 2.1. An MPI program with Master-Slave communication pattern

- *Completed*: The call is said to be completed when all associated memory effects have transpired. For instance, a send call is completed when the data payload is copied from the sender's address space to the receiver's address space.

We now describe the syntax and semantics of the MPI calls mentioned earlier.

- *Send*: MPI API provides various versions of the send call such as: plain send *MPI_Send*, buffered send *MPI_BSend*, synchronous send *MPI_Ssend*, ready send *MPI_Rsend*, and non-blocking send *MPI_Isend*. *MPI_Send* can act as a buffered send (*MPI_BSend*) call when there is a availability of runtime buffering, otherwise, it acts as a typical blocking/synchronous send (*MPI_Ssend*). The syntax of MPI_Send is the following:

```
MPI_Send(void *buff, int count, MPI_Datatype dt, int dest, int tag,
        MPI_Comm comm);
```

  Here *buff* is the pointer to the data payload to be sent, *count* is the number of elements in *buff* of data-type *dt* and *dest* signifies the destination process ID for *buff*. Additionally, *tag* is an identifier associated with the message and *comm* is a world of processes that are grouped to interact with each other. *MPI_Comm_World* is the default communicator wherein all the processes supplied by the user are grouped. Tags and communicators facilitate finer grained communication. *MPI_Isend*, on the other hand, is non-blocking and will immediately return. Its syntax is the following:

```
MPI_Isend(void *buff, int count, MPI_Datatype dt, int dest, int tag,
        MPI_Comm comm, MPI_Request* handle);
```

  the additional argument to *Isend* call is the request handle using which MPI runtime can uniquely identify this non-blocking request. Such a handle can be used by developers to ascertain the status of the non-blocking call, for instance, whether the call has completed or still pending. According to the MPI standard [44] (pg 52 lines 41-42), accessing *buff* before the successful completion of the call is illegal. In order to ascertain the completion of a non-blocking request, we rely on the *Wait* call.

- *Recv*: MPI API provides two types of receive calls, viz. blocking receive *MPI_Recv* and non-blocking receive *MPI_Irecv*. Blocking receive call successfully returns after the sent data has been copied in the receiver's intended address space. Non-blocking

receive call like non-blocking send, immediately returns and the completion of the call can happen at any later point of the time. The syntax of blocking and non-blocking receive calls are the following:

```
MPI_Recv(void *buff, int count, MPI_Datatype, dt, int src,
         int tag, MPI_Comm comm, MPI_Status *status);

MPI_Irecv(void *buff, int count, MPI_Datatype, dt, int src,
         int tag, MPI_Comm comm, MPI_Request *handle);
```

Each argument, except *src* and *status*, holds similar meaning as described earlier for the send call. The argument *src* denotes the process ID of the sender and when set to `MPI_ANY_SOURCE`, implies that the receiver is free to receive from any matching sender that is enabled. Such receive calls are termed as *wildcard receive* calls. Also note that the tags in receive calls can be set as `MPI_ANY_TAG` wich can be another source of receive non-determinism. The argument *status* is an object that stores the current state of the call and other information such as error return code (if any) and process ID of the matched sender.

- *Wait*: Wait is a blocking call that detects the completion of non-blocking call whose request handle is passed as an argument to the wait call. It returns successfully only after the non-blocking request has completed. The syntax for the wait call is the following:

  ```
  MPI_Wait(MPI_Request * handle, MPI_Status *status);
  ```

- *Barrier*: MPI API provides many constructs that require the participation of all the processes in a communicator and for this reason such calls are called as collective communication calls. Barrier is a collective *synchronization* construct. The MPI standard requires that if one process has issued a barrier within a certain communicator then all processes within that communicator must issue barrier calls too. No single process within a communicator can progress until all processes have successfully issued their barrier calls. The syntax of the barrier is the following:

  ```
  MPI_Barrier(MPI_Comm comm);
  ```

Although, there are other collective calls such as *Bcast*, *Reduce*, etc., for the purpose of this dissertation having an understanding of barriers alone will suffice.

2.1.1 Notation for MPI Calls

We will consistently use the following notation throughout this dissertation with respect to the MPI calls:

- A non-blocking send call from process i to process j with d to be data sent will be denoted by $S_{i,-}(j,d)$. The extra field next to source process ID i, is to signify the issue index of the MPI call from the process i. The symbol $-$ denotes a *don't care* value. From here on, for brevity, we will suppress the fields that are not relevant in the context.

- Similarly a non-blocking receive call receiving the data in variable x is denoted by $R_{i,-}(j,x)$.

- A non-deterministic receive from process i is denoted by $R_{i,-}(*)$. Note that we surpassed the data field in the representation. This is to illustrate the future uses of these notations where data or certain other fields hold no importance.

- A wait call associated with a handle $h_{i,l}$ is $W_{i,-}(h_{i,l})$. The handle $h_{i,l}$ denotes that a non-blocking request was made from process i at index l.

- A barrier call will be henceforth be denotes s $B_{i,-}$.

An `MPI_Send` is equivalent to $S;W$ (a non-blocking send immediately followed by a wait). Similarly, `MPI_Recv` is equivalent to $R;W$ (a non-blocking receive followed by a wait).

2.1.2 Non-determinism in MPI

MPI API provides non-deterministic constructs primarily to squeeze out most parallelism from the program whenever possible. Here is a list of constructs that introduce non-determinism in the MPI programs:

- *MPI_ANY_SOURCE* and *MPI_ANY_TAG* can be set as arguments to receive or probe calls making them non-deterministic. Receive or probes that use source and tag non-determinism will match or return true whenever there is a sender present (within the communicator) that is a compatible match with the receive/probe regardless of the process ID of the source or tag of the message.

- *MPI_Waitany* and *MPI_Waitsome* are another source of non-determinism. *MPI_Waitany* will return true when ever any one of the request handles that the wait call is waiting upon, complete successfully. *MPI_Waitsome* will return only after the set number of requests that the waitsome call is waiting upon, have completed.

2.1.3 Common Bugs in MPI

Errors in MPI programs can be caused by a variety of reasons. We present some of the bug classes in MPI programs that are most common found.

- *Deadlocks*: The main reason for the presence of a deadlock in the program is because a certain send/receive operation has *orphaned* (not found a match). The reasons for this mis-match can be:

  1. The MPI program is not *well-formed*, i.e., number of send and receive calls is the not equal. Figure 2.2 illustrates such an example.
  2. An MPI program with wrong buffering assumptions where two processes issue sends to each other. Note that in the absence of sufficient runtime buffering, the sends would act as blocking calls, leading to a *head-to-head* deadlock. Figure 2.3 illustrates such a deadlock.
  3. Presence of a non-deterministic receive which causes a deterministic receive appearing later to be orphaned. Figure 2.4 illustrates this scenario.
  4. Mismatched collective call orderings leading to a deadlock. Figure 2.5 illustrates such a deadlock.

- *Resource leaks*: Resource leaks can be fairly common in MPI applications. Application developers can create a new type or a buffer and forget to free the type or buffer thereby leading to a resource leak. From the benchmarks that we studied, we have observed that in many practical situations the resource leaks are *interleaving oblivious* errors.

- *Erroneous buffer reuse*: Accessing the buffer that has been passed as an argument to a non-blocking call, before the successfully completion of the call is illegal.

The example figures 2.2-2.5 are borrowed from [75].

```
if (rank != 0)
   MPI_Send(sendbuf, count, MPI_INT, 0, 0, MPI_COMM_WORLD);
else
   for (i = 0; i < proc_count; i++)
      MPI_Recv(recvbuf+i, count, MPI_INT, i, 0, MPI_COMM_WORLD, status+i);
```

Figure 2.2. Deadlock due to send receive mismatch

```
if (rank == 0) {
   MPI_Isend(buf, count, MPI_INT, 1, 0, MPI_COMM_WORLD, &h);
   MPI_Wait(&h, &status);
   MPI_Irecv(buf, count, MPI_INT, 1, 0, MPI_COMM_WORLD, &h);
   MPI_Wait(&h, &status);
 }
else if (rank == 1) {
   MPI_Isend(buf, count, MPI_INT, 0, 0, MPI_COMM_WORLD, &h);
   MPI_Wait(&h, &status);
   MPI_Irecv(buf, count, MPI_INT, 0, 0, MPI_COMM_WORLD, &h);
   MPI_Wait(&h, &status);
}
```

Figure 2.3. Head-to-head deadlock

```
if (rank == 0) {
   MPI_Send(buf, count, MPI_INT, 1, 0, MPI_COMM_WORLD);
}
else if (rank == 1) {
   MPI_Recv(buf, count, MPI_INT, MPI_ANY_SOURCE, 0,
            MPI_COMM_WORLD, &status);
   MPI_Recv(buf, count, MPI_INT, 2, 0, MPI_COMM_WORLD);
}
else if (rank == 2) {
   MPI_Send(buf, count, MPI_INT, 1, 0, MPI_COMM_WORLD);
}
```

Figure 2.4. Deadlock due to nondeterministic receive

```
if (rank == 0)
   MPI_Bcast(buffer, count, MPI_INT, 0, MPI_COMM_WORLD);
   MPI_Barrier(MPI_COMM_WORLD);
else if (rank == 1)
   MPI_Barrier(MPI_COMM_WORLD);
   MPI_Bcast(buffer, count, MPI_INT, 0, MPI_COMM_WORLD);
```

Figure 2.5. Deadlock due to collective call order mismatch

P_0	P_1	P_2
$S_{0,1}(1, d_0)$	$B_{1,1}$	$B_{2,1}$
$B_{0,2}$	$R_{1,2}(*, x)$	$S_{2,2}(1, d_2)$
	if$(x == d_2)$ *error*	

Figure 2.6. Crooked Barrier: Issue order vs. Match Order

2.2 Details of ISP

ISP (In-situ Partial Order analysis) [69, 78, 68, 79, 73, 71] is a dynamic verification scheduler for MPI programs. The basic working strategy of ISP is similar to Verisoft [29]. We provide brief details of ISP in this dissertation. Complete details of the ISP scheduler can be found in [69].

ISP employs a MPI-specific dynamic partial order reduction strategy (DPOR) called as POE [69] (**P**artial **O**rder under **E**lusive interleavings). POE differs with DPOR [25] in a significant manner. First of all, DPOR was constructed for multithreaded programs and as pointed by [67], DPOR implicitly assumes that instructions are executed under a total issue order. This cannot be applied to MPI, since issue order has little in common with the *match order*. Consider, for instance, the example shown in Figure 2.6. If we proceeded by verifying the MPI program according to the rules of classical DPOR with a global issue order as the only criterion then we would miss exploring the match of $S_{2,2}$ with $R_{1,2}$; therefore, the error will not be discovered. This is because $S_{0,1}$ would always precede $S_{2,2}$ in a global issue order. However, note that $S_{0,1}$ can be *concurrently alive* with $S_{2,2}$ (since, with sufficient runtime buffering available, $S_{0,1}$ successfully crosses the barrier $B_{0,2}$) and either of the racing sends can match with $R_{1,2}$.

ISP successfully verifies MPI programs for all the bug classes that were presented in Section 2.1.3. We present here some of the important details of ISP.

2.2.1 MPI Correctness Guarantee and the Matches-Before Ordering

ISP utilizes MPI runtime's correctness guarantee in order to build its Matches-Before (MB) ordering for MPI programs. According the MPI standard, the runtime must ensure that when two sends or two receive operations are issued in succession from the same process targeting/sourcing from the same destination process then the second operation must match *after* the first operation has matched. Simply put, the MPI standard enforces FIFO match ordering among subsequent send/recv calls that are of the same kind.

We refer such an MB ordering by *IntraMB* ordering, since, all the operations involved are issued from a single process. Following is detailed presentation of the MB ordering enforced by the MPI runtime that ISP dynamically builds:

- For any two send calls targeting, $S_{i,l}(j)$ and $S_{i,l'}(j)$, such that they target the same destination process j and $l < l'$ then the earlier send $S_{i,l'}(j)$ is always *matched* with a receive call before the later send call $S_{i,l'(j)}$. In other words, *sends that target the same destination must match in the issue order*. Note, however, a similar guarantee can not established w.r.t the *completion* status of such sends. It is perfectly feasible for $S_{i,l}$ and $S_{i,l'}$ to complete out-of-order. Figure 2.7 illustrates these ideas pictorially. The curved lines with arrow depict the MB ordering among operations. Note that $S_{0,1}$ matches before $S_{0,2}$, however, due to runtime buffering constraints it is possible that $S_{0,2}$ (which has a smaller data to send) will complete before $S_{0,1}$. Finally, stating the obvious, two sends $S_{i,l}(j)$ and $S_{i,l'}(k)$ that target different destinations (i.e.,$j \neq k$) can match out-of-order.

- For any two receive calls, $R_{i,l}(*)$ and $R_{i,l'}(j)$, such that $l < l'$ and $R_{i,l}$ is a wildcard receive then $R_{i,l}$ will always match before the later $R_{i,l'}$. Figure 2.8 illustrates this ordering. Note that $R_{i,l'}$ can either be a wildcard or a deterministic receive.

- For any two receive calls, $R_{i,l}(j)$ and $R_{i,l'}(*)$, such that $l < l'$ and $R_{i,l'}$ is a wildcard receive then there exists a MB ordering between $R_{i,l}$ and $R_{i,l'}$ only on the *condition* that there is a send from process j that enabled when $R_{i,l}(j)$ is issued. If the condition is not met then, $R_{i,l'}$, even though issued later than $R_{i,l}$, can match before $R_{i,l}$. Such an ordering is denoted as *Conditional MB* ordering. Figure 2.9 captures this scenario. The dotted directed edge from $R_{1,1}$ to $R_{1,2}$ would become non-existent if $S_{0,1}$ was enabled post-matching of $S_{2,1}$.

- For any two MPI calls, $Op_{i,l}$ and $Op_{i,l'}$, such that $l < l'$ and $Op_{i,l}$ is a synchronous call then $Op_{i,l}$ and $Op_{i,l'}$ are MB ordered.

- Each non-blocking request is MB ordered with its associated wait call. Figure 2.9 illustrates one such MB edge between $S_{0,1}$ and $W_{0,2}$.

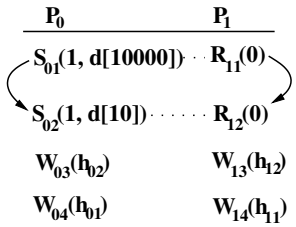


Figure 2.7. MB ordering for S and R

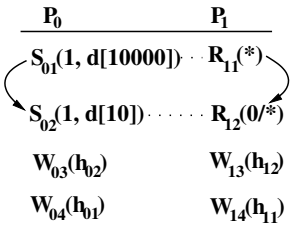


Figure 2.8. MB ordering between R* and R

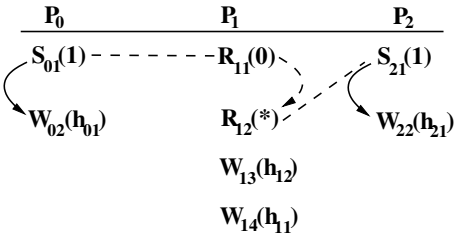


Figure 2.9. Conditional MB ordering

2.2.2 ISP's Profiler, Scheduler and the POE Algorithm

Figure 2.10 illustrates the basic blocks in the ISP tool, namely, the *profiler* and the *scheduler*.

2.2.2.1 ***Profiler*** : ISP intercept the MPI calls from the program with the help of the profiler. The profiler is essentially a collection of wrapper calls for MPI API functions utilizing the PMPI interface. Each wrapper function communicates to the scheduler and only after getting a signal to proceed from the scheduler, it issues the actual MPI call to the runtime. The profiler is compiled with the source code of the program.

2.2.2.2 ***Scheduler*** : ISP's scheduler is responsible for building the MB ordering and executing the POE algorithm. ISP scheduler is a stateless dynamic verification engine. Initially, the scheduler intercepts all the `MPI_Init` calls from each process. Each process subsequently enters a blocked state. Once the scheduler has received the initialization call from all the processes set by the user, it broadcasts the *go-ahead* signal (signal to proceed with the execution of the program) to all the blocked processes. Scheduler, subsequently operates by intercepting calls from each process. If the call issued is a non-blocking send/recv then the scheduler immediately signals a go-ahead to the process. However, if the call is a blocking call, then scheduler searches for a process that is not in a blocked

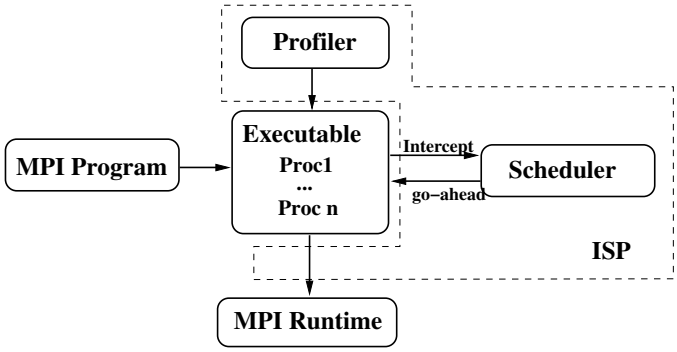


Figure 2.10. Overview of ISP tool

state and will switch to that process and start intercepting and collecting calls from that process. When scheduler arrives at a state where no process is *runnable* then we say that the scheduler has hit a *decision-point*. It invokes its verification algorithm (POE) and identifies a set of processes that can be signaled to proceed. If such set of processes is computed to be empty and there exists at least one process that has not finished executing the program then the scheduler has discovered a deadlock. Otherwise, if the set of processes is not empty, then the scheduler forms the *match-sets* (sets of matching operations). If at a decision-point, more than one match-set is formed, the scheduler explores those choices exhaustively by re-executing the program (replay up to the choice point and then pursue the alternatives).

2.2.2.3 ***POE*** : POE algorithm works in the following way: when an MPI call is encountered during program execution, the scheduler intercepts the call and records it in its state. If the call is non-blocking, the scheduler immediately signals the process that encountered the call to simply proceed with its execution. However if the call is blocking (*fence* instruction) then the scheduler searches for an another runnable process. When all processes have hit their respective fence instructions, the scheduler arrives at a decision-point. At the decision-point, the scheduler forms the match-sets. The rules for computing the match-sets are as follows:

- If at the decision-point, the scheduler has recorded barrier instruction from each process, then a set of all the barrier instructions forms a single big-step match-set move.

- If at the decision-point, the scheduler has recorded wait operations of a requests that has already been matched then the scheduler signals such waits to be issued to the

runtime.

- If at the decision-point, the scheduler has recorded a synchronous deterministic receive from a process and a compatible matching send from an another process then the scheduler forms again a match-set move comprising of the receive and the send operation. Note that multiple such match-sets comprising of synchronous recv and send calls can be exist at a decision-point. Since such match-sets are *independent* of each other (i.e. they can commute), all the match-sets can be simultaneously issued to the runtime.

- If at the decision-point, the scheduler records a wildcard receive and none of the afore-mentioned match-sets can be constructed then the scheduler constructs a set of match-sets with each match-set comprising of the same wildcard receive with one distinct matching send. An important point to note here is that only one of such match-sets can be explored in a single interleaving. The program has to be re-run, taking the same choices in the previous run until the same decision-point is witnessed at which point the un-examined choices are explored.

A natural question that comes to mind is, how does the scheduler choose a match-set when at a decision-point there are multiple type of match-sets constructed (for instance, barrier match-set, deterministic receive and send match-set, or sets of match-sets consisting of wildcard receive)? ISP scheduler assigns a priority to the match-sets. At each decision-point, the scheduler chooses a match-set with highest available priority. Following are the priority levels assigned to match-sets:

- A big-step barrier match-set is assigned the highest priority.

- A big step deterministic recv-send match-set is assigned the next highest priority.

- Wildcard receive match-sets are assigned the lowest priority.

The reason for such a prioritization is the following: ISP scheduler delays the matching of a wildcard receive as much as possible in order to discover all the matching senders. Each wildcard recv-send based match-set at a decision-point is explored in a separate interleaving by the ISP scheduler. The scheduler replays the program repeatedly until all such choices are exhausted.
The collection of match-sets that have the same priority assigned at a decision-point are

P_0	P_1	P_2
$S_{0,1}(1)$	$B_{1,1}$	$B_{2,1}$
$B_{0,2}$	$R_{1,2}(*)$	$S_{2,2}(1)$
$W_{0,3}(h_{0,1})$	$R_{1,3}(*)$	$W_{2,3}(h_{2,2})$
	$W_{1,4}(h_{1,3})$	

Figure 2.11. Example explaining POE

termed as **Persistent-sets**. This term finds it beginnings in a rich set of literature associated with the partial order reduction theory [27, 26, 32, 31] for concurrent programs. Persistent-set are singleton sets at decision-points where barrier match-sets or the synchronous receive and send match-sets are available to move. It is only when wildcard receive based match-sets are the only choice, we witness non-singleton Persistent-sets.

2.2.2.4 ***Dynamic re-writing*** : If at a decision-point, the lowest priority (wildcard receive based) match-sets are only available then in order to avoid matching sends *race* at runtime, the scheduler performs a *dynamic re-writing* of the wildcard receive operation. The wildcard receive is re-written in to a deterministic receive sourcing from the process ID of the sender that was also the part of the match-set.

Example-run: Consider the example shown in Figure 2.11.

- At the first decision-point, the ISP scheduler has recorded $S_{0,1}$ and $B_{0,2}$ from P_0, $B_{1,1}$ from P_1 and $B_{2,1}$ from P_2. The only possible match-set at this decision-point is $\langle B_{0,2}, B_{1,1}, B_{2,1}\rangle$. This match-set is issued in to the MPI run time.

- At the second decision-point, the ISP scheduler has following instructions enabled: $S_{0,1}$ and $W_{0,3}$ from P_0, $R_{1,2}$ from P_1 and $S_{2,2}$ from P_2. The match-sets computed are the following: $\langle S_{0,1}, R_{1,2}\rangle$ and $\langle S_{2,2}, R_{1,2}\rangle$. ISP scheduler picks the match-set $\langle S_{0,1}, R_{1,2}\rangle$ and rewrites the $R_{1,2}(*)$ to $R_{1,2}(0)$ and issues them to the runtime (note that when we say scheduler issues a match-set to runtime, we actually mean that the scheduler signals the profiled calls of the associated processes to proceed).

- Subsequently $W_{0,3}$ is issued in to the runtime.

- At the next decision-point, $\langle S_{2,2}, R_{1,3}\rangle$ is chosen and issued to the runtime.

- Once the execution completes, the ISP scheduler re-runs the program and explores the choice $\langle S_{2,2}, R_{1,2}\rangle$ at the second decision-point.

2.2.3 Notations for IntraMB Ordering

We now present the notations and definitions surrounding IntraMB ordering that we will use in forth-coming chapters. IntraMB ordering is a *local process* ordering. It establishes an ordering among two operations issued from the same process. Let $\prec_{lp}$ be the notation that captures the IntraMB ordering among two operations. Then the IntraMB ordering, with the assumption that $l < l'$, can be represented by the following:

- $S_{i,l}(j) \prec_{lp} S_{i,l'}(j)$.
- $R_{i,l}(j/*) \prec_{lp} R_{i,l'}(j)$.
- $R_{i,l}(j) \prec_{lp} R_{i,l'}(*)$ and there exists a $S_{j,-}(i)$ that was enabled with $R_{i,l}(j)$.
- $S_{i,l}(j) \prec_{lp} W_{i,l'}(h_{i,l})$ and $R_{i,l}(j/*) \prec_{lp} W_{i,l'}(h_{i,l})$
- $B_{i,l} \prec_{lp} Op_{i,l'}$ and $W_{i,l} \prec_{lp} Op_{i,l'}$. The barrier and wait constitute the *fence* operations.

Note that IntraMB is a transitively closed relation. We further define the following terms:

Definition 1 (Ancestor) *An operation $Op_{i,l}$ is an* Ancestor *of operation $Op_{i,l'}$ when $Op_{i,l} \prec_{lp} Op_{i,l'}$.*

Let $Op^{\ll}$ denote the set of ancestors to Op. Further, $Op^{<}$ denote the set of *immediate ancestors* to Op. Can we have a situation where we witness mutliple immediate ancestors of an operation? If not, then the set definition of immediate ancestors is not required. However, in reality we can come across situations where a single operation can have multiple ancestors. Imagine two non-blocking send calls targeting the same destination are followed by two wait calls for the first and the second send respectively. Notice that the second wait call will have two ancestors: the immediately preceding wait call and the second send call on which it waits. Let $Op^{\ll *}$ denote the set of ancestor operations of Op that includes Op.

Definition 2 (Descendant) *An operation $Op_{i,l}$ is a* Descendant *of operation $Op_{i,l'}$ when $Op_{i,l'} \prec_{lp} Op_{i,l}$.*

Let $Op^{\gg}$ denote the set of descendants to Op. Further, $Op^{>}$ denotes the set of *immediate descendants* to Op. There are situations where a single operation can have a set of immediate descendants. For instance, consider a program where in a certain process a wait call is followed by two non-blocking send calls targeting different destinations. In such a scenario

notice that both the sends are immediate descendants of the wait call. Let $Op^{\gg *}$ denote the set of descendant operations of Op that includes Op.

We further define an operator $\prec$ that establishes a total order on match-sets in an interleaving explored by ISP. Thus, $m \prec m'$ tells us that in the interleaving of the program match-set m was issued by the scheduler earlier (in time) than the match-set m'.

CHAPTER 3

FUNCTIONALLY IRRELEVANT BARRIERS IN MPI APPLICATIONS

This chapter presents the details of Functionally Irrelevant Barriers (FIB) [57]. Identifying FIBs increases the performance without compromising the correctness of the programs. Note that for illustration purposes, we will assume that the tag based non-determinism is absent and the communicator is `MPI_COMM_WORLD`. However, the algorithm operates even if we relax such constraints. Specifically this chapter covers the following contributions:

- A notion that captures Matches-Before ordering among operations from distinct processes. We denote it by InterMB relation.
- Algorithm for identification of Barrier match-sets that are not required.

3.1 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. 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 the page [23]).

The importance of detecting irrelevant barriers comes from a number of perspectives. Many MPI users are known to employ collective barriers for "good measure;" they are unsure whether it is necessary. The authors of [6] 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 [51], 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 3.2. 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, the *MB* relation introduced in the Chapter 2. This relation has two aspects: intra matches-before (IntraMB), and inter matches-before (InterMB). In a nutshell, the FIB algorithm detects a change in the set of communication possibilities by computing the InterMB 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 3.2 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([49, 70, 68]), 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 [23]), 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 small fraction of all possible interleavings, by computing only the *relevant interleavings* of an MPI program using a formal verification method called *partial order reduction* [12, 25].

Related Work: FIB is a significant extension of our POE algorithm implemented in the ISP verification tool. The mathematical relation IntraMB is employed in POE (formally defined in [69], summarized in the Section 2.2.2.3). The relation InterMB builds on IntraMB, and is brand new to the FIB algorithm.

In [62], 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 3.2 provide added insights into our criterion.

The work in [46] uses vector clocks [40], and provides a method for identifying the racing messages in a single trace of an MPI program execution across "frontiers" or *consistent cuts* [40]. 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 3.2.

Roadmap: Section 3.2 provides the intuition behind our FIB algorithm through several examples. The FIB algorithm itself is detailed in Section 3.5. Section 3.7 provides experimental results, and Section 3.8 provides concluding remarks.

3.2 Overview of FIB, and the InterMB Relation

In this section, we present a number of examples, introducing the concepts of IntraMB and InterMB 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:

$$P_0 : R_{0,1}(0); W_{0,2}(h_{0,1}); B_{0,3}; S_{0,4}(0);$$

In this program, the collective barrier is a singleton set containing $B_{0,3}$. 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 $W_{0,2} \prec_{lp} B_{0,3} \prec_{lp} S_{0,4}$. An IntraMB edge implies the MPI guarantee of not issuing any instruction after $W_{0,2}$ until $R_{0,1}$ has been completed. The IntraMB is explained in sufficient detail in Section 2.2.1. In our example, there is $S_{0,4}$ after $W_{0,2}$, and unfortunately `Wait` cannot finish unless `Isend` finishes—a circular dependency causing the deadlock.

In MPI, there is also an IntraMB edge from a `Barrier` to any following instruction (since Barrier operation is a blocking/synchronous operation). 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)[1]:

P_0	P_1
$R_{0,1}(*)$	$S_{1,1}(0)$
$B_{0,2}$	$B_{1,2}$
$S_{0,3}(0)$	$W_{1,3}(h_{1,1})$
$W_{0,4}(h_{0,1})$	

In this example, it is possible for $S_{0,3}$ to match the receive $R_{0,1}$, whether the collective barrier is there or not! This is because even though $B_{0,2} \prec_{lp} S_{0,3}(0)$, there is no IntraMB ordering between $R_{0,1}(*)$ and $B_{0,2}$, and similarly there is no IntraMB ordering from $S_{1,1}(0)$ and $B_{1,2}$. Thus, $R_{0,1}(*)$, $S_{1,1}(0)$, and $S_{0,3}$ can all be alive post-barriers and any one of the two sends can race ahead to match the receive. Therefore, for this program, FIB will flag

[1]Note all examples upto ex 5 are deadlock free hence assume count of sends and recvs match in the program. For full code please refer [23]

the collective barrier as **irrelevant**.

Example 3: Consider the program:

P_0	P_1
$R_{0,1}(*)$	$S_{1,1}(0)$
$B_{0,2}$	$B_{1,2}$
$W_{0,3}(h_{0,1})$	$S_{1,3}(0)$

Here, the collective barrier is indeed **irrelevant**, and will be flagged as such by the FIB algorithm, following this line of reasoning: (i) $R_{0,1}(*)$ and $S_{1,1}(0)$ can be issued; (ii) the `Barriers`, $B_{0,2}, B_{1,2}$, in the respective processes can be crossed, as $R_{0,1} \not\prec_{lp} B_{0,2}$ and $S_{1,1}(0) \not\prec_{lp} B_{1,2}$; (iii) before $R_{0,1}(*)$ matches, $S_{1,3}(0)$ can also be issued; (iv) however, $S_{1,1}(0) \prec_{lp} S_{1,3}(0)$ Therefore, $R_{0,1}(*)$ can match $S_{1,1}(0)$ only.

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

P_0	P_1	P_2
$R_{0,1}(*)$	$S_{1,1}(0)$	
$B_{0,2}$	$B_{1,2}$	$B_{2,1}$
$W_{0,3}(h_{0,1})$		$S_{2,2}(0)$

The `send` calls are in different processes. Therefore, there is no IntraMB ordering between them. Also, $R_{0,1} \not\prec_{lp} B_{0,2}$ and $S_{1,1}(0) \not\prec_{lp} B_{1,2}$. Thus, $R_{0,1}$ and $S_{1,1}(0)$ can live past their respective barriers. Therefore, the collective barrier is **irrelevant**. Now consider an alternative example (call it **Example 4(a)**) in which $W_{0,3}$ the is moved to be *before* its `Barrier` $B_{0,2}$.

P_0	P_1	P_2
$R_{0,1}(*)$	$S_{1,1}(0)$	
$W_{0,2}(h_{0,1})$		
$B_{0,3}$	$B_{1,2}$	$B_{2,1}$
		$S_{2,2}(0)$

Now, the collective barrier becomes **relevant**. This is because $W_{0,2} \prec_{lp} B_{0,3}$ Hence, $B_{0,3}$ cannot be crossed until $R_{0,1}(*)$ finishes. Therefore $S_{2,2}(0)$ cannot issue. Therefore, $R_{0,1}(*)$ has to match $S_{1,1}(0)$.

The reasoning employed in Example 4(a) highlights the need for the notion of *InterMB* edges. Basically, $S_{2,2}(0)$ "wishes to match" $R_{0,1}(*)$. The only thing that prevents this is

that the collective barrier orders $R_{0,1}(*)$ to be before it, and $S_{2,2}(0)$ to be after it. This is the ordering defined by InterMB (detailed in Section 3.3). Furthermore, there is no alternative ordering path starting from $R_{0,1}(*)$ to $S_{2,2}(0)$ 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:

P_0	P_1	P_2
	$S_{1,1}(2)$	$R_{2,1}(1)$
$B_{0,1}$	$B_{1,2}$	$B_{2,2}$
$S_{0,2}(2)$		$R_{2,3}(*)$

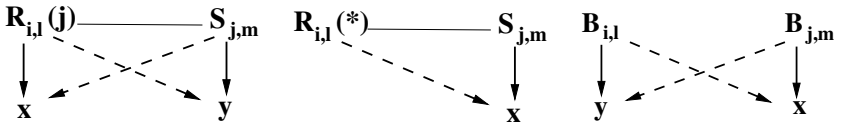
Here, the barrier is **irrelevant**. Note that $S_{1,1} \not\prec_{lp} B_{1,2}$ and $R_{2,1} \not\prec_{lp} B_{2,2}$. Thus, $S_{1,1}$ and $R_{2,1}$ can exist past the barriers. However, if there is a specific-source nonblocking receive followed by a wildcard receive in an MPI program, the wildcard receive can *trump* the specific receive (i.e. may match before it), *if there is no matching sender to the specific-source receive*! (This the conditional MB ordering explained in Section 2.2.1). In Example **5**, however, there *is* a matching $S_{1,1}$, and so trumping does not happen. Since there is no trumping, implies, $R_{2,1} \prec_{lp} R_{2,3}$. Thus, $S_{0,2}$ cannot match $R_{2,1}$ and $S_{1,1}$ cannot match $R_{2,3}$, thereby causing the barrier to be **irrelevant**.

3.3 InterMB relation

InterMB relation is built on top of IntraMB relation and the match-sets that were explored in the interleaving under focus. This makes the InterMB relation as *interleaving-specific*. Let $\prec_{ip}$ be the operator that denotes InterMB ordering between two operations. Following are the rules for InterMB computation:

- if the match-set is $\langle S_{j,m}(i), R_{i,l}(j)\rangle$ then $\forall x, y : x = R^{>}_{i,l}, y = S^{>}_{j,m}$, we have, $S_{j,m} \prec_{ip} x$ and $R_{i,l}(j) \prec_{ip} y$.
- if the match-set is $\langle S_{j,m}(i), R_{i,l}(*)\rangle$ then $\forall x : x = S^{>}_{j,m}$, we have, $R_{i,l}(j) \prec_{ip} x$.
- if the match-set is $B = \langle B_1, ..., B_n\rangle$ then $\forall x, i, k : x = B^{>}_{i}, k \neq i$ then $B_k \prec_{ip} x$.

Figure 3.1 illustrates the relation pictorially. The solid directed arrows are the IntraMB edges. The solid undirected edge is the match-set and dotted arrow are the InterMB edges. In Figure 3.1(b), note that we do not add an edge from the $S_{j,m}$ to $R^{>}_{i,l}$. This is because,



(a) Deterministic Recv (b) Non-deterministic Recv (c) Blocking Operations

Figure 3.1. InterMB relation w.r.t the match-sets

$R_{i,l(*)}$ could have matched some other send causing the $S_{j,m}$ to match with a later receive. The InterMB edges are added after the POE orchestrated interleaving has finished.

3.4 Matches-Before Relation

The MB relation is a union of InterMB and IntraMB relations. Let $\prec_{mb}$ be the operation that establishes an MB ordering among two operations. Thus, $Op_i \prec_{mb} Op_j$ implies either $Op_i \prec_{lp} Op_j$ or $Op_i \prec_{ip} Op_j$.

Definition 3 (MB-Path) *An* `MB-Path` *from operation* Op_i *to operation* Op_j *in an observed trace* τ *is defined to be an ordered sequence of operations* $\hat{Op} = \langle Op_1, ..., Op_n \rangle$ *(excluding* Op_i *and* Op_j*) such that the following conditions are met:*

- $Op_i \prec_{mb} Op_1$ *and* $Op_n \prec_{mb} Op_j$
- $\forall k : Op_k, Op_{k+1} \in \hat{Op}$ *then* $Op_k \prec_{mb} Op_{k+1}$

MB-Path between Op_i and Op_j is a path containing operations wich are either intraMB or interMB ordered with Op_j.

3.5 The Functionally Irrelevant Barrier (FIB) Detection Algorithm

We now provide a detailed presentation of the FIB algorithm and then describe the FIB algorithm. The FIB tool framework is illustrated in Figure 3.2. We have already presented the details of the *InterMB Constructor* block in the previous section. The details of the *FIB detector* are expressed in the Algorithm 1. The function *Paths(a,b)* compute a set of *MB-Paths* from operation *a* to operation *b*. The algorithm for *Paths* is illustrated in the Algorithm 2. For each send observed in the trace, the FIB algorithm looks up in the trace to check whether the send can match an earlier wildcard receive. The FIB algorithm then computes all paths from such a send to the prior wildcard receive. Now if there exists a path

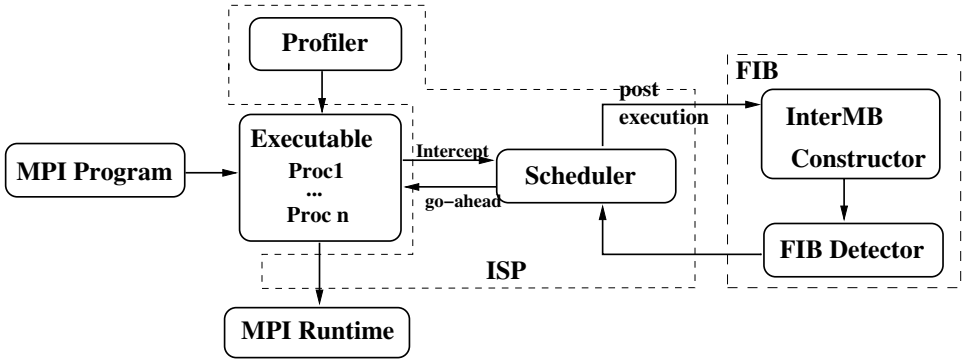


Figure 3.2. FIB framework

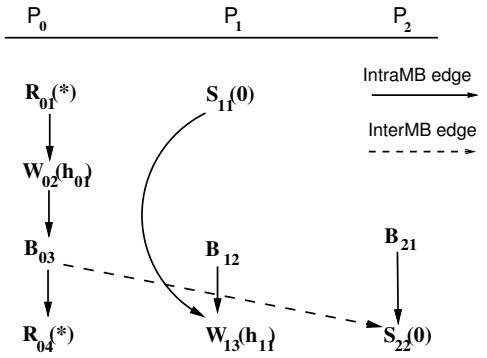


Figure 3.3. Example 4(a) in Section 3.2 with InterMB and IntraMB edges

from the send to the wildcard receive that does not involve a focal barrier match-set then that barrier match-set is FIB. Alternatively if in each interleaving all the paths have the presence of the focal barrier match-set then the barrier match-set is a functionally relevant barrier (FRB).

Illustration: In Example 4, $R_{0,1} \not\prec S_{2,2}$. Now in the alternate example called **Example 4(a)** discussed earlier, the above procedure will end up creating a path $R_{0,1} \prec_{mb} W_{0,2} \prec_{mb} B_{0,3} \prec_{mb} S_{2,2}$. There is no alternate ordering path – so the collective barrier containing $B_{0,3}$is relevant. Figure 3.3 summarizes the above explanation. The IntraMB edges depicted in Figure 3.3 for process P_0 are easy to reason. After the collective barriers are discharged in to the runtime, FIB constructs InterMB edges from one barrier to another barrier's immediate successor. After adding InterMB edges, the only path that reaches to $S_{2,2}$ from $R_{0,1}$ involves a barrier. Thus the barrier and all the barrier operations from other processes that formed the match set are flagged to be relevant.

Algorithm 1 COMPUTE-FIB

```
1:  Input:
2:        Set IBL, RBL              ▷ Irrelevant, Relevant Barrier List; Intially empty
3:        It                                                  ▷ Interleaving Tree
4:  Output:
5:        Set IBL, RBL
6:  for all interleaving in It {
7:     for all ms in interleaving {
8:        if ms = ⟨S_{i,l}(j), R_{j,m}(−)⟩ {
9:           if ∃ ms' = ⟨S_{k,−}(j), R_{j,m'}(∗)⟩ : m' < m {
10:             P ← Paths (R_{j,m'}, S_{i,l})
11:             if ∀p ∈ P, ∃ B_{i,l'} ∈ p : B_{i,l'} ≺_ip S_{i,l} {
12:                RBL ← RBL ∪ {B}              ▷ Let B be the match-set:B_{i,l'} ∈ B
13:                if B ∈ IBL {
14:                   IBL ← IBL \ {B}
15:                }
16:             } else
17:                IBL ← IBL ∪ {B}
18:             }
19:          }
20:       }
21:    }
22: }
23: if ∃B : B ∉ IBL ∧ B ∉ RBL {
24:    IBL ← IBL ∪ {B}
25: }
```

Algorithm 2 PATHS

```
1: Input: a, b
2: Output: P                          ▷ P is a set of MB-Paths from a to b
3:  Paths(a, b) {                     ▷ Computes all MB-Paths from a to b
4:    P = {}
5:    S = {}                                                ▷ Stack for DFS
6:    for all op adjacent to a {
7:       S ← DFS(op,b)
8:       P ← P ∪ S
9:    }
10: }
```

3.6 Correctness Proof

Soundness: Assume that a caught FIB $B_{i,l'}$ in a program by MSPOE is in fact not an FIB. That means it is an FRB (Functionally Relevant Barrier). Let $B_{i,l'}$ be a part of the match-set m. If it is an FRB then by definition, removing barrier operations in m, of which

a barrier $B_{i,l'}$ is a part of, will enable a later appearing send $S_{i,l}$ ($l' < l$) to match $R_{j,m}$. Let $S_{i,l}$ be a part of match-set m' and $R_{j,m}$ be a part of match-set m''. Notice that if $B_{i,l'}$ is an FRB for $S_{i,l}$ then $B_{i,l'} \prec_{ip} S_{i,l}$. Also, in any interleaving $m'' \prec m \prec m$ which implies that $R_{j,m} \prec_{mb} B_{k,n}$ where $B_{k,n} \in m$. Since $B_{i,l'} \prec_{lp} S_{i,l}$ then $B_{k,n} \prec_{mb} S_{i,l}$. Hence, regardless of which interleaving is explored, every path from $R_{j,m}$ to $S_{i,l}$ **must** include a barrier from m. The barrier set m would then accordingly be added to the FRB list (lines 10-14 of Algorithm 1) and removed from the IBL list. This proves, that a discovered FIB is indeed an FIB.

Completeness: Assume the algorithm fails to discover an FRB in an MPI program. Being an FRB implies that a certain receive $R_{j,m}$ is MB ordered w.r.t a certain send $S_{i,l}$ that targets process j via a barrier $B_{i,l'}$. The algorithm, since is based precisely on the above definition, can miss detecting an FRB only when a certain interleaving is not explored by the ISP. Note, however, that ISP being a exhuastive verifier, explores all the relevant interleavings. Thus, Algorithm 1 is complete.

3.7 Implementation and Experimental Results

We 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. The instrumentation tool is written using CIL [45] 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 [74]. As for runtimes, the ISP algorithm introduces a slowdown because of its scheduler-mediated executions ([70] provides ideas for improving the execution time). The *added* overhead that FIB introduces over and above ISP is negligible. Our web page [23] 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 [74]. Of the 36 tests that had barriers, all were flagged as **irrelevant**, with negligible runtimes.

3.8 Summary

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.*

3.8.1 Discussion

Note that the FIB algorithm cannot declare the barriers relevant or otherwise until ISP has explored all the interleavings of a program. Observe that FIB algorithm was run with the input process count unrealistically small. When the examples were made to run on larger processes, the size of the schedule space that ISP has to examine grows exponentially. Even though the execution time of FIB algorithm is negligible as opposed to the time taken by the ISP to orchestrate a schedule, such a measure has little meaning when the schedule space that must be examined by ISP, is inanely large. This observation, led us to examine ways to prune the schedule space of MPI programs over-and-above the pruning performed by *POE*. The work presented in the next chapter is an effort in such a direction. Some important questions that one must bear in mind before exploring any schedule space pruning strategies are:

- Does the pruning strategy masks a certain safety property violation?
- Does the strategy offer any formal guarantee w.r.t. detection of certain safety property violation or (in our case) detection of relevant/irrelevant barriers?

CHAPTER 4

PERSISTENT-SET REDUCTION HEURISTIC FOR MPI PROGRAMS

In this chapter, we present the details of a heuristic to effectively reduce the **Persistent-sets** (described in Section 2.2.2.3) at a decision-point. The heuristic is highly effective for common applications in the MPI landscape. In other words, the assumptions that lay the foundations of this work are common programming practices in the MPI application space. The heuristic work presented in this chapter is sound, however, it is not complete w.r.t. deadlock detection as we explain in later sections of this chapter. For the purpose of FIB detection, the heuristic is complete as long as the barriers are *textually aligned*. The specific contributions of this chapter are:

- Discuss prime motivation for the heuristic which is, MPI codes deadlock under the presence of deterministic receive calls.
- Present the notion of independent operations in a MPI program and finally discuss the details of the heuristic algorithm.

4.1 Introduction

A significant risk facing MPI codes being used in practice is that when they employ non-deterministic communication constructs (such as MPI wildcard receives), there may be a vast number of unexamined behaviors. Recently created *formal dynamic verifiers* such as ISP [69, 78] and DAMPI [77] take an approach that integrates the best features of testing tools (ability to run on user applications) and model checking (coverage guarantees). They run the MPI program under the control of a verification scheduler, and thanks to their MPI semantics-aware algorithms, *guarantee to detect* all potential communication matches for wildcard receives. They also guarantee to *enforce* these matches. The net effect is that they can scale up to 1000s of MPI processes (such as in DAMPI) and handle realistic MPI program runs on cluster machines, and regardless of the actual speed-paths in the cluster,

ensure full coverage of non-determinism.

Problem Statement: Unfortunately, dynamic formal verifiers such as ISP and DAMPI are *indiscriminate* in covering non-determinism. This can lead to an exponential blow-up in the number of execution schedules that a verification scheduler has to explore. For instance, consider an MPI program with $n + 1$ processes where each of the n processes sends a message to the $(n + 1)^{th}$ process. The $(n + 1)^{th}$ process posts n wildcard receive calls (say in a loop). One can easily observe that even in such a simple setting, there will be $n!$ execution schedules. This is clearly unacceptable: all dynamic verifiers must, ideally, be equipped with approaches to detect when such exhaustive explorations are unnecessary, and then avoid them.

Eliminating unnecessary nondeterministic matches in a program with multiple identical processes is an instance of *parameterized reasoning* which is formally undecidable [12] and very difficult to approximate in practice. We don't attempt to solve the entire problem—but do provide a specialized dynamic analysis method that significantly reduces the number of interleavings while detecting deadlocks due to the *orphaning of deterministic receive* operations —something that MPI programmers do worry about. Our method is implemented by augmenting the ISP tool and its dynamic verification algorithm POE, and is called **MSPOE** (the name comes from "macroscopic POE") [54]. We first let POE compute the potential send matches for MPI non-deterministic receives, as it currently does. The execution history following the non-deterministic receive is then examined by MSPOE. It chooses to include only some of these sends (called *relevant sends*) for later exploration with respect to this non-deterministic receive. These sends are the ones considered relevant to cause orphaned receive deadlocks.

Observation: We say that an MPI program does not "decode data" if it does not employ data dependent control flows, and does not alter its control flows based on which specific send/receive matches occurred. For an MPI program that does not decode data and has a orphaned deterministic receive causing a deadlock, it must either have an unequal number of sends and receives in some execution path, or must satisfy these conditions: (i) it employs a process employing a wildcard receive and a specific receive; (ii) a previous wildcard receive consumes a send that was meant for the later occurring specific receive, thus orphaning the specific receive. MSPOE exploits this observation and computes relevant sends based on the occurrence of specific receives.

P_0	P_1	P_2
$S_{0,1}(2)$;	$S_{1,1}(2)$;	$for(i = 1$ to $4)$
		$R_{2,i}(*)$;
$S_{0,2}(2)$;	$S_{1,2}(2)$;	end for;

Figure 4.1. Deadlock free example

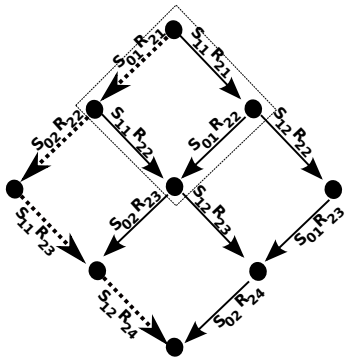


Figure 4.2. State graph for Figure 4.1

One may initially think that our problem is one of symmetry detection, which has been extensively researched [9, 36, 19, 11]. Symmetry detection is based on constructing a smaller *quotient structure* of the system by exploiting the *automorphism* in the system's state space. These are computationally hard problem [9] which are impractical during dynamic verification of MPI programs. The work in [15] computes symmetries in communicating programs based on channel graphs and not directly applicable for our purposes.

Contributions:

- We present a macroscopic partial order elusive interleaving reduction (MSPOE) algorithm that exploits communication symmetry.
- We demonstrate the savings made by MSPOE for the purpose of deadlock detection and FIB detection.

Motivating examples:

Observe that the example shown in Figure 4.1. The ISP scheduler will explore six interleavings for this example. The six interleavings are illustrated in Figure 4.2. Note that solid circles are the states and the directed edges are the match-sets signaled to the runtime at that state. The dotted arrow edges is the first interleaving that ISP explores. However, observe that the example code has only wildcard receive calls. Thus, as long as all sends *commute*, such examples cannot have deadlocks and there is no necessity to examine other schedules.

MSPOE will analyze the program in Figure 4.1 in the following way:

- MSPOE will explore the first interleaving as shown by dotted arrows in Figure 4.2.

P_0	P_1	P_2	P_3	P_4
$S_{0,1}(4);$	$S_{1,1}(4);$	$S_{2,1}(4);$	$S_{3,1}(4);$	$R_{4,1}(*);$
				$R_{4,2}(3);$
				$R_{4,3}(*);$
				$R_{4,4}(*);$

Figure 4.3. Deadlocking example

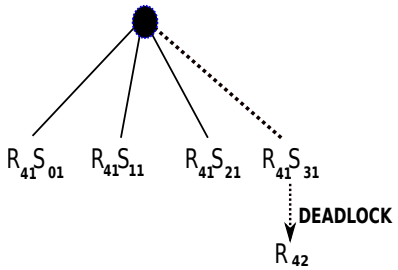


Figure 4.4. Possibilities after first R(*) match

- MSPOE discovers that it did not encounter any specific receive calls. Thus, MSPOE will reduce the persistent-set of each non-deterministic receive to a singleton set (containing the entry that was explored in the current run of the program). Note that in the Figure 4.2, the states that are bounded in the dotted box will witness their persistent-set reduced. For the rest of the states, the persistent-set is a singleton-set to begin with.

- At the end of the interleaving exploration, the ISP scheduler removes the entry (chosen in the current interleaving) from the persistent-set at each state. Since, each persistent-set has already been reduced to singleton set by MSPOE, the ISP scheduler subsequently will erase these single entries. Hence, the ISP scheduler's check whether another run of the program is required based on the presence of a state with un-examined persistent-set entry, will return false, thereby ending the verification process.

In the example of Figure 4.3, there is a deadlock introduced by the use of the deterministic receive call. Figure 4.4 shows that if $R_{4,1}$ were to match $S_{3,1}$ (right-most transition from the initial node), the subsequent deterministic call ($R_{4,2}$) will be orphaned, thus creating a refusal deadlock. ISP would explore all the matches starting from leftmost choice shown in Figure 4.4 and then moving right with every new run, generating four interleavings before finding the deadlock. MSPOE will, on the other hand, choose $S_{3,1}$ as the next relevant send to explore after any initial run. This guarantees that the deadlock will be detected in two interleavings, at most.

In a nutshell, MSPOE allows one to incorporate specialized modes of verification within tools such as ISP and DAMPI. In these modes, one can have a static analyzer that deter-

mines whether data decoding is going on; and in the absence of data decoding (true for many large examples), deploy MSPOE to often obtain orders of magnitude reduction in the number of interleavings.

4.2 Preliminaries

Let P be a concurrent MPI program and P_i is the i^{th} sequential process executing P where $i \in PID$ and $PID = \{0, 1, ..., n\}$. We assume the program is executed with finite many processes. Each P_i is L_i instructions long. Let l denote the program counter(PC) array; thus, $l_i \in l$ denotes the PC value for the i^{th} process. The jth MPI command in the ith process is denoted $p_{i,j}$ where $j \in L_i$.

As explained in Section 2.1.1, a non-blocking send call issued by the process P_i with a program counter j with a destination as P_k is denoted as $S_{i,l_i}(k)$. Similarly, a non-blocking receive call is written as $R_{i,l_i}(k)$. If the receive is a wild-card move then its denoted as $R_{i,l_i}(*)$. An MPI Barrier operation by process i is represented as $B_{i,j}$ where j is the l_i for that process. Let Op be the set of MPI operations, i.e.,

$$Op = \cup_{i,j,k,m} \{S_{i,j}(k),\ R_{i,j}(k),\ R_{i,j}(*),\ B_{i,j}\}$$

. In out presentation, we will *suppress* all *wait*[1] calls and show the *IntraMB* ordering appropriately. Note that an operation belonging to Op is a *visible* operation and all other operations are *invisible*. A visible operation is one that is intercepted by the ISP scheduler.

The state of the system is represented as $\sigma = \langle I, P, M, l \rangle$ that consists of *issued* ($I \subseteq Op$) instructions, *persistent-set* (P) set, *matched* ($M \subseteq I$) instructions, and the PC array l. It is really the state of the ISP scheduler since knowing the precise state of MPI runtime is very hard. We keep an approximate track of the MPI runtime via maintaining the scheduler state. We refer these states as system's states. The set of all states of the system is denoted by $\mathcal{S}$.

Set of instructions that are issued (*i.e.*, instructions in I) but not completed in a state σ are the *enabled* instructions sitting ready to be matched. Persistent-set P at a state $\sigma \in \mathcal{S}$ (denoted by P_σ) is a set of *match-set* moves (as explained in Section 2.2.2.3) Since match-set transitions the system from one state to a subsequent state, we view match-set moves as the *transitions* of the MPI program. The terms match-sets and transitions in

[1]To simplify the presentation we take such a step although synchronous/asynchronous operations are both handled by the MSPOE algorithm

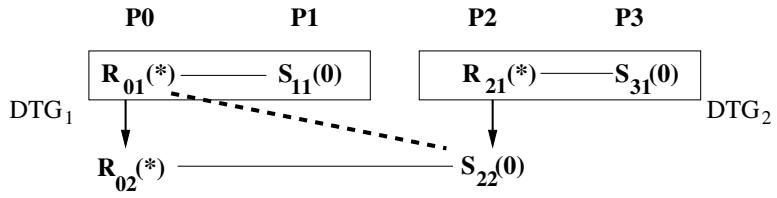


Figure 4.5. Dependence among DTG transitions

this dissertation would be used interchangeably. Thus, when a send call $S_{i,l_i}(k)$ matches a receive call $R_{k,l_k}(i)$ at σ, the associated transition $t \in P_\sigma$ is represented by $\langle S_{i,l_i}(k),\ R_{k,l_k}(i)\rangle$. Completed instructions are those that have found a match and have been signalled into the runtime by the ISP scheduler.

Let $\mathcal{T}$ denote the set of all transitions of the system. A $t \in \mathcal{T}$ enabled at state s which when executed results in a unique successor state s', written as $s \xrightarrow{t} s'$. The successor state is also represented by the following: $s' = t(s)$. We define the whole MPI program as a state transition system $A_G = (\mathcal{S}, \delta, so)$ where $\delta \subseteq \mathcal{S} \times \mathcal{S}$ is the *transition relation* defined by:

$$(s, s') \in \delta \text{ iff } \exists t \in \mathcal{T} : s \xrightarrow{t} s'$$

and s_0 is the starting state of the system. A_G of the example in Figure 4.1. is shown in Figure 4.2.

4.2.1 Nature of Transitions in a Persistent-set

A persistent-set at a state can have multiple transitions. Persistent-sets are constructed in a prioritized manner as discussed in Section 2.2.2.3. The only possibility of a persistent-set containing multiple transitions is when there is a wildcard receive involved. When all the potential senders to a wildcard receive are completely determined at a state we observe that ISP takes each sender and forms a transition with the wildcard receive call. The work in [67] views all resulting transitions as *dependent* and designates the collection of such transitions as *dependence transition group* (**DTG**). For instance, in Figure 4.2 the DTG w.r.t the receive $R_{2,1}$ has the following transitions: $t_1 = \langle S_{0,1}, R_{2,1}\rangle$ and $t_2 = \langle S_{1,1}, R_{2,1}\rangle$. We define a function $Dtg(s) \restriction_{R_{i,l}}$ that returns a set of transitions that belong to the DTG w.r.t. to the non-deterministic receive $R_{i,l}$ that are enabled at a state s.

Notice, however, multiple DTGs can co-exist at a state. Example shown in Figure 4.5 illustrates such a scenario. Figure 4.5 shows one trace of the program. Note that the

solid un-directed arrows were the match-sets fired in the execution. The dotted un-directed arrow represents another possible match-set. The solid directed arrows capture the IntraMB ordering[2]. Observe, that augmentation of the DTG_1 can happen only when transition in DTG_2 is fired before the transition in DTG_1. This would result in $S_{2,2}$ be enabled with $S_{1,1}$ and $R_{0,1}$. The result is the following: DTG_1 is augment from containing the transition $\langle S_{1,1}, R_{0,1}\rangle$ to containing two transitions $\langle S_{1,1}, R_{0,1}\rangle$ and $\langle S_{2,2}, R_{0,1}\rangle$.

POE_{OPT} (optimized POE) in [67] takes an optimistic stand and tags any two transitions belonging to separate DTGs that are enabled at a given state as initially *independent* and if it discovers later that ordering a transition in one DTG may lead to the augmentation of another DTG (discovery of an additional send that can match the wildcard receive) then it adds transitions of both the DTGs in the persistent-set of that state.

The MSPOE algorithm, on the other hand, aspires to optimize the working of POE_{OPT} within a single DTG. The whole exercise of MSPOE is to optimistically treat transitions within a DTG in σ as *independent* and operate on a reduced persistent-set. Only when a transition later is discovered to be dependent, we accordingly augment the persistent-set in σ where the dependent transitions are concurrently enabled.

Why is it important to discover dependent/independent transitions? Every POR method leverages on the *independence* among transitions. If by changing the order of execution of concurrent *independent* transitions we witness no effective change in the state of the system then it suffices to explore just one such interleaving order among transitions.

4.3 Formal Definition of Independent Transitions

In order to first define *independent transitions*, we first introduce the notion of *commuting sends* that are part of the transitions within a single DTG.

Definition 4 (Commuting Sends) *: Sends $S_{i,l}(k)$ and $S_{j,m}(k)$ are commuting sends iff the following conditions hold:*

- *Let $t_1 = \langle S_{i,l}(k), R_{k,n}(*)\rangle$ and $t_2 = \langle S_{j,m}(k), R_{k,n}(*)\rangle$ such that $t_1, t_2 \in Enabled(s)$.*
- *$S_{j,m}(k) \in t_2'$ and $S_{i,l}(k) \in t_1'$ where $t_2' = t_1(s)$ and $t_1' = t_2(s)$.*

[2]The edge between $R_{2,1}$ and $S_{2,2}$ indicates that their must be a wait operation fo $R_{2,1}$ in between which has been suppressed but the effects are appropriately captured in the IntraMB edge.

P_0	P_1	P_2
$R_{01}(*)$	$S_{21}(0)$	$S_{11}(0)$
$R_{02}(*)$		

Figure 4.6. Commuting example

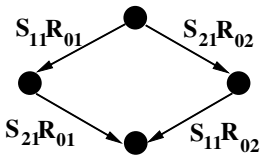


Figure 4.7. Transition independence

Observe that in Definition 4, two sends, S_i and S_j can commute only when they are enabled and part of transitions t_1 and t_2 in a state s and firing one send at s should not leave the other send *disabled or unmatched* in the subsequent state. The $\mathcal{C}$ be the relation for commuting sends. We not define independent relation as:

Definition 5 (Independent Relation) *$I \subseteq \mathcal{T} \times \mathcal{T}$ is an* independence relation *iff for each $\langle t_1, t_2 \rangle \in I$ following conditions hold:*

1. **Enabledness:** *t_1 and $t_2 \in$ Enabled(s) and there exists a $R_{k,n}(*)$ such that $t_1, t_2 \in Dtg(s) \restriction_{R_{k,n}}$.*
2. **Commutativity:** *If $S_{i,l}(k) \in t_1$ and $S_{j,m}(k) \in t_2$ then $(S_{i,l}, S_{j,m}) \in \mathcal{C}$.*

Thus, with the independent relation, we now can say two transitions t_1 and t_2 are dependent when the send operations in t_1 and t_2 do not commute. Consider the example and its corresponding state graph shown in Figure 4.6 and Figure 4.7. The initial state s_0 has two enabled transitions, viz.: $t_1 = \langle S_{1,1}, R_{0,1} \rangle$ and $t_2 = \langle S_{2,1}, R_{0,1} \rangle$. Note that transitions commute since they lead to the same final state. Firing t_1 disables t_2 in the next state, however, the transition enabled at $t_1(s)$ is $t_2' = \langle S_{2,1}, R_{0,2} \rangle$ and $t_2 \equiv_c t_2'$. Thus, t_1 and t_2 are independent.

If send calls in t_1 and t_2 do not commute (assuming t_1 was fired from s) then following can be the only reasons:

- The send from t_2 is disabled at $t_1(s)$.
- The operation available at $t_1(s)$ is not a receive t_2's send can match with. If the operation enabled at $t_1(s)$ is a receive, then it must be a deterministic receive which is sourcing from a process other than the process that issued t_2's send.

We discuss in detail the ability of MSPOE to compute the independence of transitions in Section 4.6. We stick to the same *persistent* definition that is defined in [28].

Definition 6 (Persistent in s) *A set T of transitions enabled in a state s is* persistent in s *iff, for all non empty sequences of transitions from s in A_G*

$$s = s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 ... \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1}$$

and including only transitions $t_i \notin T$, $1 \leq i \leq n$, t_n is independent in s_n with all transitions in T.

Informally, this means that when a transition sequence is generated from a state s by choosing only transitions that are independent with transitions in T then the final state reached cannot have a transition that is dependent with any of the transitions in T. The interleavings obtained by only executing the entries in the persistent-set at every state are the *representative* interleavings and result a quotient state graph denoted as A_R. Such representative interleavings are also called as *Mazurkiewicz traces*[41].

Let's revisit the state graph shown in Figure 4.2. Using Definition 5), we now can reason about the example. Notice that for the states shown in the dotted box, the DTGs at those states have only independent transitions. Thus, for the purpose of verification of safety properties (such as absence of deadlocks), or FIB detection, examining only *one representative* interleaving would suffice.

4.4 MSPOE Algorithm

Algorithm 3 presents the MSPOE algorithm in detail. The match-set move (or the transition) selected at a particular state s in an interleaving is denoted by $Curr(s) \in P_s$ where P_s is the persistent-set at state s. RP_s is the *reduced* persistent-set at state s. We also maintain a stack St of states that have been visited but not completely explored. Algorithm 4 presents ISP scheduler's functioning to generate the interleaving of the program. Algorithm 5 depicts the prioritized match-set selection policy of POE which remains the same for MSPOE. Algorithm statements tagged with $*$ are additions to POE which transform POE into MSPOE.

MSPOE starts with the initial state s_0 in the stack. We generate a complete interleaving by calling the function *GenerateInterleaving* (line 6 in Algorithm 3) We repeat the following steps from this point forwards until the state stack (St) becomes empty:

- Select the last state s from the trace and remove the match-set entry explored in the trace from P_s and RP_s. If RP_s becomes empty then pop the state off from the state stack St.

Algorithm 3 MSPOE Algorithm

```
1: Input:
2:    Stack of State:  St                          ▷ St has s_0; initial state
3:    Vector of Set:  P                            ▷ Persistent-set for each state
4:    Vector of Set:  RP                           ▷ Reduced Persistent-set for each state

5: s ← First(St)                                   ▷ Get bottom of Stack St
6: St ← GenerateInterleaving(s)
7: while ∼ Empty(St) {                             ▷ continue until St becomes empty
8:    s ← Last(St)                                 ▷ Get top of Stack St
9:    RP_s ← RP_s \ {Curr(s)} *        ▷ Curr(s) returns the match-set chosen at state s
10:   P_s ← P_s \ {Curr(s)}
11:   if Empty(RP_s) { *        ▷ RP_s was singleton and was explored in the interleaving
12:      St ← St − s                               ▷ Remove state s from St
13:   } else
14:      St ← GenerateInterleaving(s)
15:   }
16: }
```

P_0	P_1	P_2
$S_{0,1}(2)$	$S_{1,1}(2)$	$R_{2,1}(*)$
		$R_{2,2}(*)$
$B_{0,2}$	$B_{1,1}$	$B_{2,3}$
	$S_{1,3}(2)$	$R_{2,4}(2)$

Figure 4.8. MSPOE with redundant exploration

- If the after executing the step the last state has non-empty RP_s then generate further interleaving from s.

Algorithm 4 takes as input a state and generates an interleaving from that state in the following manner:

- From P_s, choose a match-set m according to POE's prioritized match-set selection procedure.
- Add m to RP_s.
- If m involves a deterministic receive, then search for each state s' in the stack St and perform the following:

Algorithm 4 GenerateInterleaving from state s

```
1: Input:
2:    State:  s
3:    Stack of State:  St
4: Output:
5:    Stack of State:  St

6: while s is not NULL {                          ▷ Continue until next state can't be found
7:    m ← Choose(P_s)                             ▷ Choose a match-set to explore from s
8:    RP_s ← RP_s ∪ {m} *
9:    if m = ⟨S_{i,l}(j), R_{j,m}(i)⟩ { *                                  ▷ if m has det recv
10:      for all s' ← s − 1 until First(St) { *                             ▷ Update RP_{s'}
11:         if ∃B_{i,−} ∈ P_{s'} : B_{i,−} ≺_{lp} S_{i,l} { *
12:            goto Next_State *
13:         }
14:         if ∃m' ∈ P_{s'} : m' = ⟨S_{i,−}(j), R_{j,−}(∗)⟩ ∧ m' ∉ RP_{s'} { *
15:            RP_{s'} ← RP_{s'} ∪ {m'} *
16:         }
17:      }
18:   }
19:   Next_State: s ← Explore(s, m)               ▷ Get the next state by firing m from s
20:   St ← St + s                                 ▷ Add s to the Stack
21: }
22: return St
```

Algorithm 5 Choose P_s

```
1: Input:
2:    State:  s
3: Output:
4:    Match-set:  m

5: if ∃ m ∈ P_s : m contains barrier {
6:    return m
7: else if ∃ m ∈ P_s : m contains wait {
8:    return m
9: else if ∃ m ∈ P_s : m contains deterministic recv {
10:   return m
11: else if ∃ m ∈ P_s : m contains non-deterministic recv {
12:   return m
13: }
```

1. If $P_{s'}$ contains a match-set m' involving a send from the same process whose send is a part of m at P_s then add m' to $RP_{s'}$.

				Interleavings		Time(sec)
Benchmark	Buffering	# of procs	Deadlocks?	ISP	MSPOE	MSPOE
Mat-Multiply	0	4	No	54	1	0.001
		8	No	120	1	0.002
	∞	4	No	54	1	0.3
		8	No	120	1	0.3
2D-Diffusion	0	4	Yes	1	1 $\surd$	-
	∞	4	No	90	1	0.314
		8	No	> 10,500	1	0.442
Pi- Monte-Carlo	0	4	No	36	1	0.002
		8	No	5040	1	0.003
	∞	4	No	36	1	0.24
		8	No	5040	1	0.3
Integrate_mw	0	4	No	81	81	-
		8	No	2401	2401	-
Madre	0	4	Yes	1	1 $\surd$	-
	∞	4	No	>8000	1	1.48
		8	No	>8000	1	3.09
Parmetis	0	4	No	1	1	128.933
Heat-Diffusion	0	4	Yes	7	5	1.611
		8	Yes	5041	23	11.95
Gaussian Elimination	0	4	No	1	1	0.24
		8	No	1	1	0.276
	∞	4	No	180	1	0.31
		8	No	>20,000	1	0.324

Table 4.1. Interleaving results for deadlock detection

2. However, if $P_{s'}$ contains a barrier operation MB ordered with the send that is part of m then move terminate $RP_{s'}$ update and move-on to explore the next state in the interleaving. Consider the example shown in Figure 4.8. Notice that no matter which interleaving is explored, $S_{1,3}$ can never be enabled and be a potential match for receive calls $R_{2,1}$ and $R_{2,2}$ since such a match is restricted by the presence of barriers. We avoid such unnecessary augmentation of persistent states by adding the barrier check (lines 12-13) to the MSPOE algorithm.

- Repeat all the step until no more states can be explored.

4.5 Experimental Results

All the experiments were run on Intel Core 2 Duo 2 Ghz with 3 GB of RAM. We set a time limit of 2 hours to verify the benchmarks. We abort the verification process if the it did not complete within the time-limit. The results pertaining to the reductions obtained are documented in Table 4.1.

P_0	P_1	P_2
$R_{0,1}(*)$	$S_{1,1}(0)$	$S_{2,1}(0)$
$R_{0,2}(*)$	$S_{1,2}(0)$	$S_{2,2}(0)$
$R_{0,3}(*)$		
$R_{0,4}(*)$		
$B_{0,5}$	$B_{1,3}$	$B_{2,3}$
$S_{0,7}(1)$	$S_{1,4}(2)$	$S_{2,4}(0)$
$\cdots$	$\cdots$	

Figure 4.9. Communication in 2D-Diff

2D-Diffusion: We tested ISP's POE and MSPOE algorithm on *2D-Diffusion* [22] example. The code has a deadlock when evaluated in *zero buffering mode*. In this mode, the send calls act as synchronous operations. The deadlock was caught by ISP and MSPOE right in the first interleaving. The sign $\surd$ in the MSPOE column next to the number of interleavings examined illustrates that MSPOE also caught the same deadlock. When the same code is run on *infinite buffering mode*, the code becomes deadlock free. The code was modified to run with a single time-step. Its communication pattern is shown in the Figure 4.9. Note that if sends were treated as synchronous then after barriers each process is blocked on their respective sends causing a deadlock.

Integrate: `Integrate_mw` [22] is another benchmark that uses heavy non-determinism to compute an integral of sin function over the interval $[0, Pi]$. `Integrate` has a master-slave pattern where the root process divides the interval in a certain number of tasks. The root process then delegates to each worker process a single task and then waits for results from them by posting wildcard receive calls. Workers that finish early with their work are provided with more tasks until all tasks are distributed (as detailed in the high level code below)

```
Worker i: while(1) {
            R(from 0, any-tag); // Recv task
            if(work-tag)
              S(master, result-tag);
            else break;
          }

Master: for(i = 1 to nprocs-1) {
```

```
  Send(i, work-tag); // send to each worker the task
  tasks++;
}
while(tasks <totalTasks){
   Recv(*, result-tag); // recv result
   S(S.S, work-tag);  // assign more task
   tasks++;
}
for(i = 1 to nprocs-1) {
   Recv(i, result-tag); // recv result
   S(i, terminate-tag);  // terminate signal to worker i
}
```

This benchmark does not have a deadlock. Notice that MSPOE does not demonstrate any savings over ISP while exploring the schedule space. This is because, the master process finally posts deterministic receive calls targeting each worker before it sends termination signals to each worker. This causes the MSPOE to fully expand the persistent-sets of each prior wildcard receive.

MADRE: MADRE [59], a memory aware data redistribution engine, is a library written in MPI which mainly performs load balancing tasks in an efficient manner. MADRE moves the data blocks across nodes in a distributed system within the bounds of memory available to each of the application's process. We tested MADRE with its *unitBred* algorithm on various data-sets. *unitBred* algorithm is of particular interest to us because it uses MPI_ANY_SOURCE and MPI_ANY_TAGS. MADRE has no bugs provided normal MPI send calls are not treated as blocking calls. We ran ISP's POE and then MSPOE algorithm with sbt9 dataset with unitBred algorithm and the results are documented in the Table 4.1.

Parmetis: Parmetis [37] is a parallel hypergraph partitioning code-base. Since, Parmetis only uses deterministic calls, ISP and MSPOE complete the verification process in a single interleaving. Parmetis was selected as a benchmark despite the absence of non-determinism because the application issues a lot of MPI calls which served as a basis to evaluate the scalability of the data-structures used in MSPOE. When run on 4 processes, Parmetis issues

Benchmark	ISP	MSPOE
2D-Diffusion	1 FRB (171), 1 FIB(169)	1 FRB (171), 1 FIB(169)
Gaussian Elimination	6 FIB (51, 243, 302, 306, 312, 317)	6 FIB (51, 243, 302, 306, 312, 317)
Madre	2 FIB (455, 502)	2 FIB (455, 502)

Table 4.2. FIB results with MSPOE

$\sim 55,000$ calls.

Heat Diffusion: Heat diffusion is an MPI example borrowed from the SC 2011 tutorial presented by G. Gopalakrishnan etal. that has a deep-seated deadlock. ISP discovers the deadlock in 5041^{th} intereaving when the benchmark is run on 8 processes. MSPOE, on the other hand, discovers the same deadlock in 23^{rd} interleaving.

4.5.1 MSPOE for Identifying FIBs

We ran the same becnhmarks in Table 4.1 for the FIB analysis. Among all the benchmarks, the only ones that have either FIBs or FRBs are listed in Table 4.2. Note that all benchmarks were evaluated for 4 processes under *infinite buffering mode*. In each case MSPOE returned with the exact set of FRBs and FIBs that ISP reported. In Table 4.2 notice that numbers in curved brackets are the line number of Barrier calls issued from the source program.

4.6 Discussion

An important question pertaining to the working of MSPOE is the following: Does MSPOE precisely compute all the dependent actions in an MPI program? Notice that MSPOE only augments the persistent-set of a prior state (at which a wildcard move took place) only when a deterministic receive is witnessed later in the trace. It is by no means a complete criterion to discover all dependent transitions.

Consider, for instance, the example shown in Figure 4.10. In this example, if $S_{3,1}$ matched $R_{1,1}$ then $S_{1,2}$ and $S_{2,1}$ would engage in a cyclic wait on each other causing a deadlock. Notice that $S_{1,2}$ can't match unless $S_{2,1}$ successfully completes since $R_{2,2}$ is the only match of $S_{2,1}$ and $S_{2,1}$ is an enabler operation for $R_{2,2}$. Notice that MSPOE will fail to discover such a deadlock. However, a pertinent question that will underscore the usability of MSPOE is the following: how often such coding patterns are employed in applications, if at all? In real MPI codes that we have assessed, we did not witness such a coding style. Typically, a deterministic communication from a process following a wildcard receive is

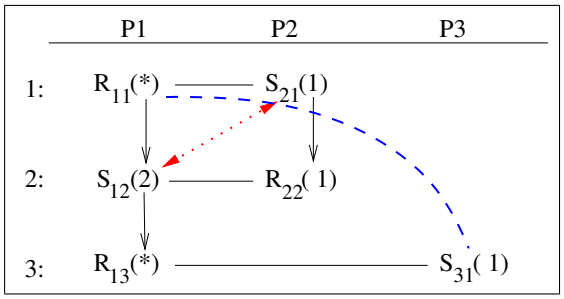


Figure 4.10. Deadlock because cyclic dependency between $S_{1,2}$ and $S_{2,1}$

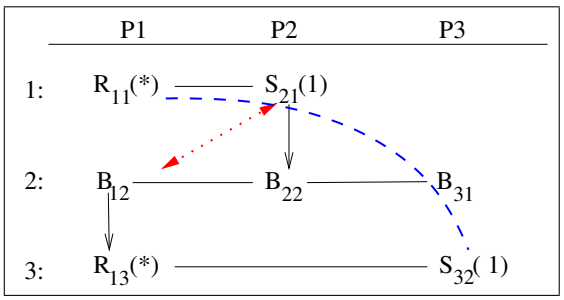


Figure 4.11. Deadlock because barriers do not discharge

accomplished by *reply channels*. Processes often employ reply channels to perform dynamic load balancing duties by sending data/task to the sender that matched the prior wildcard receive. Thus, in our opinion, it is rare (almost to none) to observe that applications issue hard-wired deterministic receives/sends following a wildcard receive operation. Notice that in Figure 4.10, if $S_{1,2}(2)$ is re-written as $S_{2,1}(status.Source)$ (indicating a reply-channel) then the deadlock in the code disappears.

Figure 4.11 is another example where MSPOE will fail to detect a deadlock. In Figure 4.11, note that the barriers would not discharge if $S_{3,2}$ were to match $R_{1,1}$ thereby causing the deadlock. Notice that $S_{3,2}$ is unordered w.r.t. $B_{3,1}$. This can happen only when $S_{3,2}$ is issued before $B_{3,1}$ however the wait associated with $S_{3,2}$ is issued after the barrier. Again, such a coding practice is flawed and we have not witnessed any real MPI program so far that employs such a coding style. Typically, global fence operations (such as barriers) are issued only *after* the local fence operations such as waits are successfully discharged. If such were to be the programming style then the wait calls for both $R_{1,3}$ and $S_{3,2}$ should have been issued before the respective process barriers. In which case, the match-set $\langle B_{1,2}, B_{2,2}, B_{3,1} \rangle$ would be issued only after the completion of $\langle S_{3,2}, R_{1,3} \rangle$. Even in alternate trace when $S_{3,2}$ pairs-up with $R_{1,1}$, notice that $S_{2,1}$ will now find a match in $R_{1,3}$. Hence, the deadlock will disappear.

In all our benchmarks, none of above mentioned coding styles were employed except the deterministic receive calls following a wildcard receive. MSPOE, thus, as a result of such observations, despite being in-complete works extremely well (in other words, appears complete) in practice. Constructing a methodology that is complete forms the basis of our next work, detailed in the subsequent chapters.

4.7 Conclusions

We have presented a novel algorithm MSPOE that demonstrates significant savings in the exploration space of programs for the purpose of communication deadlock detection and FIB detection. In many cases the the reductions were from tens of thousands of interleavings to just one interleaving. We document the MSPOE reduction results observed over several benchmarks. We further present evidence on the criticality of the match-set selection in avoiding redundant explorations and for early detection of bugs.

Future work: Conditional communication flow pattern is sill not tackled by MSPOE. However, MSPOE algorithm can be notified of the causal receive calls whose buffers when decoded would result in a conditional communication flow. Such information can be statically mined and provided to the dynamic verification scheduler. To gather the afore-said information, we would require a MPI specific control flow graph (CFG). Work in [7] presents *p-cfg* which is a CFG for MPI programs. Our future work would therefore lie in modifying the *p-cfg* work to handle non-deterministic MPI operations. Furthermore, we will develop flow-sensitive static analysis methods on top of the improved *p-cfg* to analyze conditional communication patterns.

CHAPTER 5

GENERALIZED MATCHES-BEFORE RELATION

We discuss the inconclusiveness of InterMB ordering because it is a weak ordering relation. We then present a generalization of Matches-Before (MB) relation by providing a tighter Wait-for ordering among operations from the distinct processes. Finally we present the rules to construct the desired relation. We further present the importance of generalizing the existing MB ordering (detailed in Chapter 3) and its criticality in our *predictive verification* effort.

5.1 Introduction

In earlier chapters, we presented the MB ordering and its utility in the FIB and the MSPOE algorithms. However, the central question is whether the MB ordering is general enough to provide us with the savings that we are seeking in the exploration space? The answer is in negative. It is because, the InterMB ordering (defined in Section 3.3) is *interleaving-aware*. Thus, two operations that are InterMB ordered in one interleaving may no longer be ordered the same way in an alternate interleaving. Consider the example shown in Figure 5.1. A complete interleaving is demonstrated in the figure. Matches that took place in the interleaving are shown by solid un-directed lines and possible matches in alternate interleavings are shown by dotted un-directed lines. The solid directed lines are capturing IntraMB ordering and the red dotted directed arrow shows the InterMB ordering. Note that w.r.t the InterMB rule shown in Figure 3.1(b), we have that $R_{0,2} \prec_{ip} S_{2,2}$. However, in an alternate interleaving when $S_{2,1}$ matches with $R_{0,1}$ and $R_{0,2}$ matches with $S_{2,2}$ then $R_{0,2} \not\prec_{ip} S_{2,2}$. Does it mean that the FIB algorithm that utilizes InterMB ordering is broken? Notice that FIB explores **all** the interleavings that ISP generates. The FIB decision is settled only after all the relevant interleavings are examined. Hence, the afore-mentioned abnormality in InterMB relation does not affect FIB analysis. Besides we have also presented the soundness and completeness proof of the FIB algorithm. However,

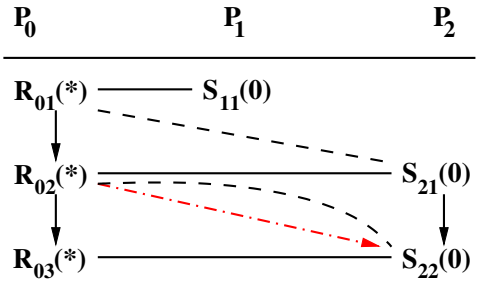


Figure 5.1. Example illustrating inconclusiveness of InterMB ordering

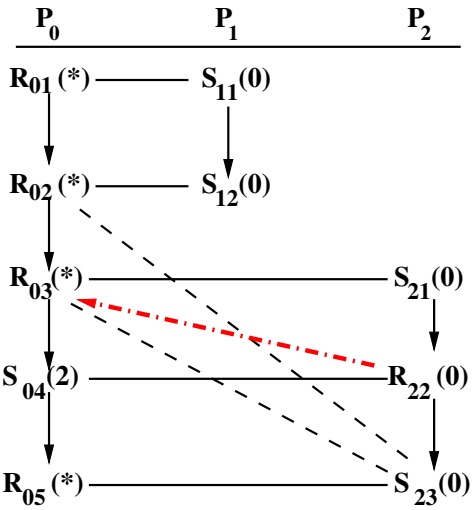


Figure 5.2. Example illustrating ordering enforced by deterministic operations

this discussion puts an important point across the table that we cannot, in general, depend on *interleaving-aware* constructs when the goal is to prune the interleaving space.

The larger question is: Can the *interleaving-oblivious* global dependencies among communication operations be established and computed precisely? Consider the same example in Figure 5.1. Note that regardless of whichever interleaving is explored, $S_{2,2}$ can never match $R_{0,1}$. This is because, $S_{2,1} \prec_{lp} S_{2,2}$ and $R_{0,1}$ being the first receive from process P_0 must match $S_{2,1}$ first (*non-overtaking ordering* detailed in Section 2.2.1). Such an ordering is based on FIFO ordering among sends and thus can be computed precisely.

The above discuss illustrates that a global dependency can be established among operations,however, is the non-overtaking ordering the only ordering that forms the basis of global dependency among operations? Consider the example shown in Figure 5.2. In this example, if we strictly adhere to the non-overtaking rules of matching then $S_{2,3}$ can match

$R_{0,3}$ or $R_{0,2}$. However, due to an additional ordering constraint, such matches would never manifest in reality as we demonstrate in the ensuing discussion. Observe that there is a deterministic receive $R_{2,2}$ ordered in the following manner: $R_{2,2} \prec_{lp} S_{2,3}$. Thus, in any interleaving $R_{2,2}$ must match before $S_{2,3}$. Operation $R_{2,2}$'s first and only match is $S_{0,4}$. Hence, $R_{2,2}$ cannot match any operation earlier than $S_{0,4}$. In such a situation, we can view $R_{0,3}$ as an *enabler* for $S_{0,4}$. Hence, we can safely say that matching $R_{2,2}$ is globally dependent on $R_{0,3}$ and regardless of whichever interleaving is examined, $R_{2,2}$ will match **only after** $S_{0,4}$ has matched.

Equipped with the observations discussed above, we present in this chapter the details to comprehensively construct global dependencies among operations which we also connote by *Wait-for* (W) ordering. The W ordering is constructed by examining a single run of the program. In order to construct W relation by only observing a single interleaving, we must ensure that the program communication flow is not dependent on choice of sender that a particular wildcard receive matches. In other words, the communication actions issued by a process P after it engages in a non-deterministic receive is unaffected by which of the vying senders it chooses to match with (P does not decode the identity of the sender nor the data payload and change its future program paths). We identify such "forgetfulness" property of these programs which we call *sender oblivious message matching* (SOMM).

Notice further that to precisely compute the W relation we must know all the potential match possibilities of the operations involved. For instance, in Figure 5.2, the W ordering from $R_{2,2}$ to $R_{0,3}$ could be established only after ascertaining that $S_{0,4}$ is the first and the only legal match for $R_{2,2}$. Hence, we also present the rules to construct the match possibilities of the communication operations in MPI programs. We denote the relation capturing the match possibilities of MPI operations by M. Finally, we revisit the MB ordering (presented in Section 3.4) and modify it with the newly constructed W relation.

5.2 Preliminaries

We define the notion of *type equality* (denoted by $\equiv_t$) among MPI operations. Two operations Op_1 and Op_2 are type equal, *i.e.*, $Op_1 \equiv_t Op_2$ when the following holds:

- Either $Op_2 \prec_{lp} Op_1$ or $Op_1 \prec_{lp} Op_2$ (*i.e.*, both operations are issued by the same process)
- If Op_1 is a send(or recv) then Op_2 is also a send (or recv).

The first condition reveals that not only Op_1 and Op_2 are issued by the same process, they are also ordered in a certain way in the trace. Either Op_2 matches before Op_1 or the other way around in any observed trace. We extend the notion of *type equality* by adding *target equality* to it (denoted by $\equiv_{t,d}$) and term it by *type-target equality*. Thus, two operations Op_1 and Op_2 are type-target equal, *i.e.*, $Op_1 \equiv_{t,d} Op_2$, when the following holds:

- $Op_1 \equiv_t Op_2$
- Op_1 and Op_2 has the **same destination** process. If Op_1 is a send to process j then Op_2 is also a send to process j. If Op_1 is a receive sourcing from process j then Op_2 is either a receive sourcing from j or a wildcard. If Op_1 is a wildcard receive that matched a send from process j in the observed trace then Op_2 is either a deterministic receive sourcing from process j or a wildcard.

We extend the notation $Op^{\ll}$ defined on Page 20. Let $Op^{\ll_k}$ return a set of k many ancestors of the operation Op. Similarly, $Op^{\gg_k}$ return a set of k many descendants of the operation Op. Further $Op^{\ll_{k,p}}$ return a set of k many ancestors of the operation Op that satisfy the predicate p. Similarly, $Op^{\gg_{k,p}}$ return a set of k many descendants of the operation Op that satisfy the predicate p. The implementation of $Op^{\ll_{k,p}}$ is illustrated in Algorithm 6. The function $GetImmAncs(Op)$ returns the immediate IntraMB ancestors of the operation Op. The implementation of $Op^{\gg_{k,p}}$ can be similarly constructed.

Example: Consider the example shown in Figure 5.2. Let the predicate p be: $\exists x : R_{0,5} \equiv_{t,d} x$. Then $R_{0,5}^{\ll_{2,p}} = \{R_{0,3}, R_{0,2}\}$.

We further define C as a function mapping an ordered pair of integers to an ordered triple of integers - $C : N \times N \rightarrow N \times N \times N$ where N is the set of natural numbers. Let C_k return the entire relation C at the event τ_k in the sequence τ. Note that a trace of the program as a sequence of match-sets is represented as τ. The k^{th} event of this trace sequence is represented by τ_k. Let $C_k(i,j)$ return an ordered triple $\langle S_i^{cnt}(j), R_j^{cnt}(i), R_j^{cnt}(*)\rangle$ for the process pair (i,j) at τ_k. This triple captures the total number of point to point communication events between P_i and P_j until τ_k. More specifically, if P_i is the sender and P_j is the receiver then the ordered triple captures the following information:

- total count of send calls from P_i to P_j
- total count of deterministic receive calls from P_j sourcing P_i, and

Algorithm 6 ComputeKPAncs

```
1: Input:
2:     Operation : Op
3:     Integer: k
4:     Property: p
5: Output:
6:     Set of Operation: res

7:   Ancs ← {Op_i}
8: while k > 0 ∧ ∼ Empty(Ancs) {
9:     ImmAncs ← GetImmAncs(Ancs)
10:    for all x ∈ ImmAncs {
11:        if x satisfies p {
12:            res ← res ∪ {x}
13:            k ← k − 1
14:        }
15:    }
16:    Ancs ← ImmAncs
17: }
18:   return res;
```

- total count of non-deterministic receive calls from P_j.

Let $C_k[(i,j) \leftarrow e]$ represent an update the entry $C_k(i,j)$ by e. Let $C_k(i,j).fst$, $C_k(i,j).sec$, and $C_k(i,j).trd$ denote the first, second, and third fields of the ordered triple $C_k(i,j)$ respectively. We inductively build the C relation by executing the following rules.

Init Condition:

$$\frac{true}{C_0(i,j) = \langle 0,0,0 \rangle}$$

Rule 1:

$$\frac{e \in [1,n], \tau_e = \langle S_{i,l}(j), R_{j,m}(i) \rangle}{\texttt{Let } prev = C_{e-1}(i,j) \texttt{ in } C_e[(i,j) \leftarrow \langle prev.fst\texttt{++}, prev.sec\texttt{++}, prev.trd \rangle]}$$

Rule 2:

$$\frac{e \in [1,n], \tau_e = \langle S_{i,l}(j), R_{j,m}(*) \rangle}{\begin{array}{l}\texttt{Let } prev = C_{e-1}(i,j) \texttt{ in } C_e[(i,j) \leftarrow \langle prev.fst\texttt{++}, prev.sec, prev.trd\texttt{++} \rangle], \\ \forall k : k \neq i, k \neq j, \texttt{Let } prev = C_{e-1}(k,j) \texttt{ in} \\ \qquad\qquad C_e[(k,j) \leftarrow \langle prev.fst, prev.sec, prev.trd\texttt{++} \rangle]\end{array}}$$

The explanation for rule 1 is fairly evident. If the match-set τ_e involves a send from P_i and deterministic receive from P_j then increment the send count and the deterministic receive count of $C(i,j)$ maintainted at event τ_e. Rule 2 is slightly involved. If τ_e involved a

non-deterministic receive from P_j then the rule not only updates the entry for $C_e(i,j)$ (the entry for communication processes) but also all the non-deterministic receive count for all other entries in C_e.

5.3 Potential Match (M^o) Relation

Consider two MPI operations Op_i and Op_j such that they have not matched each other in a trace τ. Op_i is a potential match of Op_j if there exists an alternate execution trace in which they are *legally* matched by the MPI runtime. Let the potential match relation (M) be the symmetric set of all such pairs (Op_i, Op_j). However, looking at a single execution trace we cannot initially conclude that a pair of operations form a legal match in an alternate trace. Therefore, we initially construct an over-approximation of M (which we label by M^o). We present the definitions of some of the helper functions which we will later use to present the M^o construction rules.

Let $E(C_e(i,j)) = C_e(i,j).sec + C_e(i,j).trd - C_e\ (i,j).fst$. The E function computes the number of prior wildcard receive calls (until τ_e) issued by P_j that did not match the sends from P_i. In other words, E at τ_e captures the number of prior receive calls that a send (that matched in τ_e) can potentially match in alternate interleavings.

Lemma 1 (E preserves non-overtaking ordering) *The function* E *respects the per-process based non-overtaking ordering.*

Notice that while computing the extra receives that can potentially match a send from process i (which was part of match-set at τ_e) we remove the number of sends already witnessed and matched from process i pior to τ_e. Thus, E function avoids the addition of superfluous edges that violate the non-overtaking ordering.

Function $D_{i,j}(k)$ returns k if from the trace event where the function $D_{i,j}$ is invoked there exist k many instances of a deterministic receive issued by P_j that sources from P_i. If k many instances do not exist then `INT_MAX` is returned. This function is required to discover the match ordering enforced by deterministic receive calls. The E function is oblivious to such an ordering. The higher level intuition behind the existence of such a function is the following: n^{th} instance of send from P_i targeting P_j can slide down from its current match and match later receive calls from P_j sourcing P_i, however, only up to (and including) the n^{th} instance of deterministic receive call . Figure 5.3 provides a trace of an example that clarifies the intuition. In the example, the non-overtaking ordering constraint

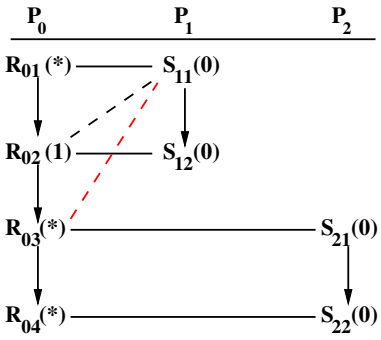


Figure 5.3. Deterministic Recv pinning a send

as implemented in E function, will restrict the matching of $R_{0,4}$ with $S_{1,1}$. However, the E function will be unable to discover that $S_{1,1}$ cannot match any receive appearing after $R_{0,2}$. The deterministic receive $R_{0,2}$ pins the ability of $S_{1,1}$ to match with any later receive. $S_{1,1}$ is the first send from P_1 and cannot match any receive appearing after the 1^{st} instance of the deterministic receive from P_0. Such a matching constraint will be captured by the D function.

Init Condition:

$$\frac{true}{e \in [1,n], \tau_e = \langle S_{i,l}(j), R_{j,m}(i/*)\rangle, \tau_e \in M^o}$$

Upward-M rules:

$$\frac{e \in [1,n], \tau_e = \langle S_{i,l}(j), R_{j,m}(i/*)\rangle, E(C_e(i,j)) > 0}{\begin{array}{l}\texttt{Let } \phi = \forall Op : R_{j,m} \equiv_{t,d} Op,\ \ K = E(C_e(i,j)) \texttt{ in} \\ \quad \{\langle S_{i,k}(j), x\rangle |\ x \in R_{j,m}^{\ll_{K,\phi}}\} \subset M^o\end{array}}$$

Downward-M

$$\frac{\begin{array}{l}e \in [1,n], \tau_e = \langle S_{i,l}(j), R_{j,m}(i/*)\rangle, C_e(i,j).fst > C_e(i,j).sec, \\ \texttt{Let } K = E(C_n(i,j)) - E(C_e(i,j)) \texttt{ in } K > 0\end{array}}{\begin{array}{l}\texttt{Let } \phi = \forall Op : R_{j,m} \equiv_{t,d} Op, \texttt{ in} \\ \quad \texttt{Let } D = D_{i,j}(C_e(i,j).fst - C_e(i,j).snd) \texttt{ in} \\ \quad\quad \texttt{Let } K' = min\{D, K\} \texttt{ in } \{\langle S_{i,l}(j), x\rangle \mid x \in R_{j,m}^{\gg_{K',\phi}}\} \subset M^o\end{array}}$$

The initial condition adds all the matched events in the trace to the M^o set. Assuming we have already computed the C relation by applying the rules presented in Section 5.2,

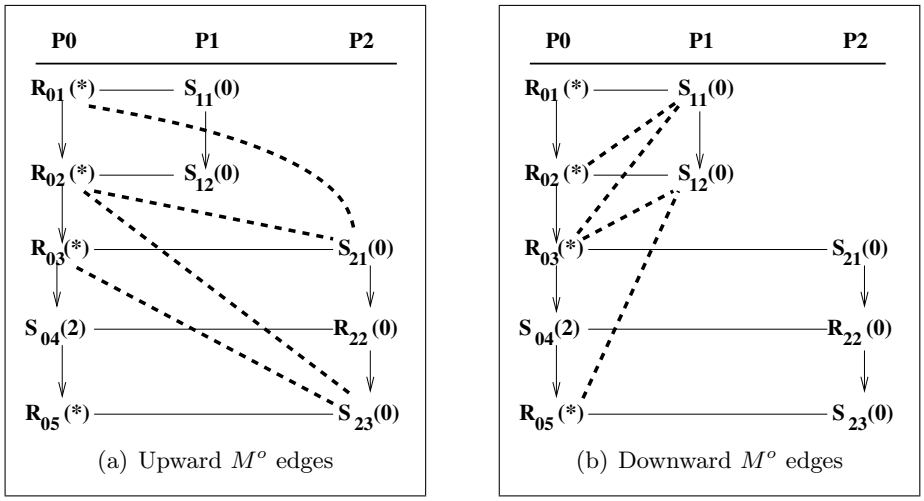


Figure 5.4. Upward and downward M^o edges

we now present the details of the *Upward-M* and *Downward-M* rules. The *Upward-M* rules construct the M^o edges for a send that can find possible matches in receives that appear prior to the receive that matched the send in the trace. Similarly, the *Downward-M* rules construct the M^o edges for a send that can potentially match receives appearing later than the receive that matched the send in the observed trace. In other words, upward and downward rules capture the upward and downward matching mobility of sends.

Consider the trace of the example shown in Figure 5.4(a). In this figure, the sends $S_{1,1}$ and $S_{1,2}$ matched in trace events τ_1 and τ_2 respectively cannot move up and match prior receives because there aren't any prior receive operations. The $E(C_1(1,0)) = 0$ and $E(C_2(1,0)) = 0$ captures that information. However, $E(C_3(2,0)) = E(C_5(2,0)) = 2$ and $R_{0,3}^{\ll_{2,\phi}} = \{R_{0,2}, R_{0,1}\}$, $R_{0,5}^{\ll_{2,\phi}} = \{R_{0,3}, R_{0,2}\}$ with appropriate ϕ predicates. This suggests that for sends $S_{2,1}$ and $S_{2,3}$ there are two prior receive operations with which these sends can potentially match.

The downward mobility of sends is computed in a similar manner. Consider the same example in Figure 5.4(b). Notice that for trace event τ_3 we have the following: $K = E(C_5(2,0)) - E(C_3(2,0)) = 2 - 2 = 0$. Thus, there are no later receive operations from $R_{0,3}$ with which $S_{2,1}$ can potentially match with. The same reasoning is applied to the send $S_{2,3}$. In the Downward-M rule, $E(C_n(i,j))$ captures the total number of receive operations from P_j that did not match sends from P_i even though they were a compatible match for such sends. Similarly, $E(C_e(i,j))$ captures the number of receive operations up to (and including) the trace event τ_e such that they were compatible for a match with send operations from

	C			E			K			D		
	(1,0)	(2,0)	(0,2)	(1,0)	(2,0)	(0,2)	(1,0)	(2,0)	(0,2)	(1,0)	(2,0)	(0,2)
τ_0	$\langle 0,0,0\rangle$	$\langle 0,0,0\rangle$	$\langle 0,0,0\rangle$	-	-	-	-	-	-	-	-	-
τ_1	$\langle 1,0,1\rangle$	$\langle 0,0,1\rangle$	$\langle 0,0,0\rangle$	0	1	0	2	1	0	∞	∞	∞
τ_2	$\langle 2,0,2\rangle$	$\langle 0,0,2\rangle$	$\langle 0,0,0\rangle$	0	2	0	2	0	0	∞	∞	∞
τ_3	$\langle 2,0,3\rangle$	$\langle 1,0,3\rangle$	$\langle 0,0,0\rangle$	1	2	0	1	0	0	∞	∞	∞
τ_4	$\langle 2,0,3\rangle$	$\langle 1,0,3\rangle$	$\langle 1,1,0\rangle$	1	2	0	1	0	0	∞	∞	∞
τ_5	$\langle 2,0,4\rangle$	$\langle 2,0,4\rangle$	$\langle 1,1,0\rangle$	2	2	0	0	0	0	∞	∞	∞

Table 5.1. Computation of C, E, K and D details

P_i but did not match those sends. Then $E(C_n(i,j)) - E(C_e(i,j))$ denotes the number receive operations from the trace event τ_e onwards up to τ_n that are a compatible match for sends from P_i but did not match sends from P_i. For send operation $S_{1,1}$, note that $K = E(C_5(1,0)) - E(C_1(1,0)) = 2 - 0 = 2$. Also note that $D = D_{1,0}(1) = INT\_MAX$ since there are no deterministic receive operations posted by P_0. This results in $K' = min\{D, K\} = 2$. Hence, the send $S_{1,1}$ can match with two receive operations appearing immediately after $R_{0,1}$. Those receive operations are in the set $R_{0,1}^{\gg_{K',\phi}} = \{R_{0,2}, R_{0,3}\}$. WE apply similar reasoning for the send $S_{1,2}$. The M^o edges computed are shown in the figure with dotted undirected lines. We provide complete details for the M^o relation computation for the example in Figure 5.4(a) in the Table 5.1.

Let's revisit example in Figure 5.4(a). From the construction rules, we have discovered that $S_{2,3}$ can potentially match $R_{0,2}$ and $R_{0,3}$ in alternate interleavings. However, notice that $R_{2,2}$ is a deterministic receive whose only match is $S_{0,4}$. Since $R_{2,2} \prec_{lp} S_{2,3}$, we deduce that $S_{2,3}$ cannot match any operation that is MB ordered with $S_{0,4}$. Hence, we must refine such false M^o edges from the M^o relation. In order to refine such edges we first define the following:

$$M^o(Op_i) = \{Op_j \mid (Op_i, Op_j) \in M^o \vee (Op_j, Op_i) \in M^o\}$$

$$F(Op_i) = \{x \mid x \in M^o(Op_i) \wedge \forall y \in M^o(Op_i) : x \in y^{\ll}\}$$

$$L(Op_i) = \{x \mid x \in M^o(Op_i) \wedge \forall y \in M^o(Op_i) : x \in y^{\gg}\}$$

$F(Op_i)$ is the set of *first* IntraMB ordered operations from each process in the set $M^o(Op_i)$ and $L(Op_i)$ is the set of *last* IntraMB ordered operations from each process in the set $M^o(Op_i)$. When Op_i is a send or a deterministic receive then $F(Op_i)$ and $L(Op_i)$ are singleton sets . However, when Op_i is a wildcard receive then $F(Op_i)$ and $L_(Op_i)$ can be non-singleton sets in which case we define the set projections w.r.t process IDs. $F^j(Op_i)$

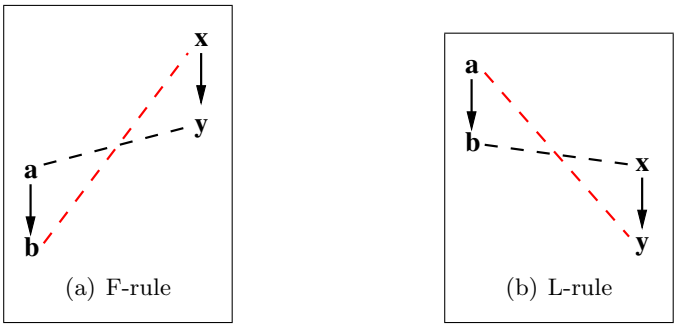


Figure 5.5. F-rule and L-rule

is first IntraMB ordered operation from P_j that belongs to the set $M^o(Op_i)$. Similarly, $L^j(Op_i)$ is the last IntraMB ordered operation from P_j that belongs to the set $M^o(Op_i)$.

We now define the M^o *consistency rule.*

Definition 7 (M^o Consistency Rule) *For any two operations, a and b, such that $a \prec_{lp} b$ then the following relation holds:*

$$\forall j: \ (F^j(a) \prec_{lp} F^j(b)) \ \vee \ (F^j(a) = F^j(b)),$$
$$\wedge \ (L^j(a) \prec_{lp} L^j(b)) \ \vee \ (L^j(a) = L^j(b))$$

Definition 7 articulates the following fact: if there exists an IntraMB ordering between two operations a and b then w.r.t each process j, $F^j(a)$ and $F^j(b)$ must either be IntraMB ordered or the same operation. Similarly, $L^j(a)$ and $L^j(b)$ must also either be IntraMB ordered or the same operation. We refine the set M^o by removing all those edges that violate the above property. The following rule presents this refinement formally:

F-rule:

$$\frac{\forall a, b: a \prec_{lp} b, \quad \forall j, \exists x, y: x \in F^j(b), y \in F^j(a), x \prec_{lp} y}{M^o \setminus \{\langle b, z\rangle | z \in y^{\ll}\}}$$

L-rule:

$$\frac{\forall a, b: a \prec_{lp} b, \quad \textit{Let} \quad \forall j, \exists x, y: x \in L^j(b), y \in L^j(a), x \prec_{lp} y}{M^o \setminus \{\langle a, z\rangle | z \in x^{\gg}\}}$$

The F-rule and L-rule are illustrated in Figure 5.5(a) and Figure 5.5(b) respectively. The red dotted arrows illustrate the false M^o edges that must be removed. Consider again the example in Figure 5.4(a). Notice that $R_{2,2} \prec_{lp} S_{2,3}$. Furthermore, $F^0(R_{2,2}) = S_{0,4}$ and $F^0(S_{2,3}) = R_{0,2}$. Thus, any operation prior to $S_{0,4}$ cannot belong to the $M^o(S_{2,3})$. This

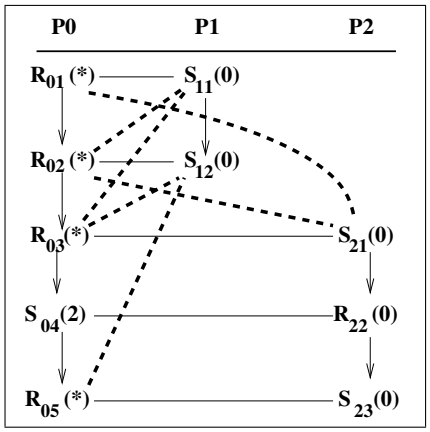


Figure 5.6. Complete M^o graph of example in Figure 5.4(a)

leads to the removal of the following edges from the M^o relation: $(S_{2,3}, R_{0,3})$ and $(S_{2,3}, R_{0,2})$. Figure 5.6 shows the complete M^o graph of the example in Figure 5.4(a) after applying the F and L rules.

5.4 Wait-for (W) Relation

We now formally discuss the Wait-for (or W) relation that captures the global interleaving-oblivious orderings on communication operations of a program. An operation Op_i is Wait-for dependent on Op_j only when Op_j is either an *enabler* to all the operations belonging to the set $M^o(Op_i)$ or an enabler for Op_i itself. We use M^o to construct Wait-for dependencies. Since, M^o is over-approximate, we, thus, construct an under-approximation of W which we represent as W^u. We present the operational semantic rules for the computation of W^u below.

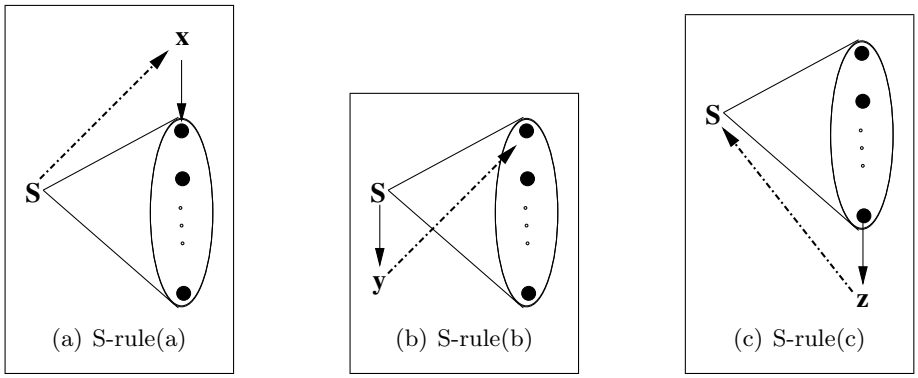
S-rule:

$$\frac{e \in [1,n], \tau_e = \langle S_{i,l}(j), R_{j,m}(i) \rangle}{\begin{array}{l}\{(S_{i,l}(j), x) |\ x \in F^j(S_{i,l})^<\} \cup \\ \{(y, F^j(S_{i,l})) |\ y \in S^>_{i,l}\} \cup \\ \{(z, S_{i,l}(j)) |\ z \in L(S_{i,l})^>\} \ \subset W^u\end{array}}$$

R-rule:

$$\frac{e \in [1,n], \tau_e = \langle S_{i,l}(j), R_{j,m}(i) \rangle}{\begin{array}{l}\{(R_{j,m}(i), x) | x \in F^i(R_{j,m})^<\} \cup \\ \{(y, F^i(R_{j,m})) | y \in R^>_{j,m}\} \cup \\ \{(z, R_{j,m}) |\ z \in L^i(R_{j,m})^>\} \ \subset W^u\end{array}}$$

We provide an additional rule for the barriers. Barrier match-sets are ordered w.r.t to prior



(a) S-rule(a) (b) S-rule(b) (c) S-rule(c)

Figure 5.7. Figures demonstrating the three parts in S-rule

IntraMB ordered operations from each process.

B-rule:

$$\frac{e \in [1,n], \tau_e = \langle \bigcup_{i \in [1,n]} B_{i,\_} \rangle}{\{\langle B_i, x \rangle | x \in B_j^{<} : j \neq i\} \subset W^u}$$

The explanation of the S-rule can be best understood by the Figure 5.7. In the figures, the cone represents the M^o of a send and the variables have the same meaning as described in the rule parts of S-rule. At each trace event, the rules are applied depending on the kind of operations involved in the event. Furthermore, these rules are applied after the phase of M^o construction has finished.

S-rule and R-rule are exactly the same. We will explain the S-rule here in detail. Note that a send call (`s`) is always targeted and therefore $M^o(s)$ is a set of operations from a single process. Figure 5.7 demonstrates the Wait-for dependencies introduced by the S-rule. It is evident that `s` must match after `x` has matched since `x` is an ancestor operation to all possible matches of `s`. Similarly, `z`, being descendant to all possible matches of `s`, must match after `s` has matched. The only Wait-for dependency in S-rule that has a subtle explanation is the edge from $s^{>}$ to $y = F^j(s)$. If `y` could find a match in the descendant of `s` in some interleaving then, by definition, `y` will not be the $F^j(s)$. Thus, under no execution interleaving `y` can match any send later than `s`. Hence, the Wait-for dependency from $s^{>}$ to y is correct.

A natural question that a reader may ask is: how does the Wait-for edges introduced for wildcard receives? A Wait-for edge can only be introduced starting from a wildcard receive $R_{j,m}(*)$ when the following holds:

$$\exists x : (R_{j,m}, x) \in W^u \text{ iff } R_{j,m} \in L^j(x)^{>}$$

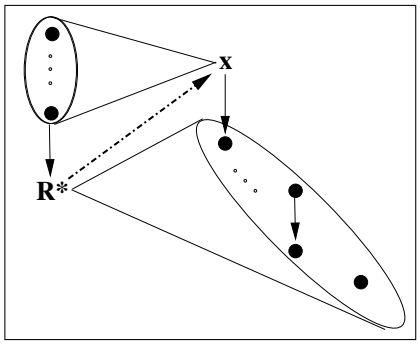


Figure 5.8. Condition for introducing Wait-for from a wildcard recv

This condition is pictorially represented in Figure 5.8 Note that this condition is accommodated within S-rule(c) and R-rule(c), thus, we do not require a separate rule for wildcard receives.

Theorem 2 *If* $(Op_i, Op_j) \in W^u$ *then* $\forall x \in Op_j^{\ll *}, (Op_i, x) \in W^u$ *and* $\forall y \in Op_i^{\gg *}, (y, Op_j) \in W^u$

Proof : It is straightforward to observe that if Op_i is waiting for Op_j to match then Op_i is waiting for every operation that is IntraMB ordered before Op_j. The proof follows from the definition of IntraMB ordering. Similar reasoning applies for an IntraMB ordered descendant of Op_i.

□

Let $\prec_w$ be the operator that defines Wait-for ordering among two operations. Thus, $Op_j \prec_w Op_i$ would mean that Op_j can match only after Op_i has matched. In other words, $Op_j \prec_w Op_i \equiv Op_i \prec_{mb} Op_j$. We re-define the MB ordering (presented in Section 3.4) which is now a union of IntraMB and Wait-for relation instead of the union of IntraMB and InterMB relations.

Consider the example from Figure 5.6. Applying the S and R rules for Wait-for construction we obtain the Wait-for edges as illustrated in Figure 5.9 by directed red dotted arrows.

5.5 Potential Match (M^o) Relation Refinement

Like we mentioned before, M^o is an over-approximate construction. In this section, we provide rules for *refining* the M^o relation. We locate false M^o edges and remove them by

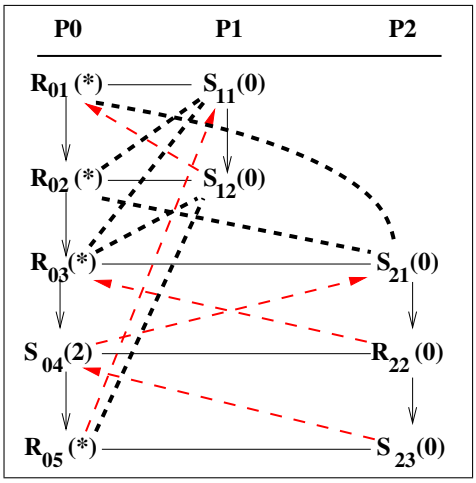


Figure 5.9. Wait-for edges introduced due to S and R rules with complete M^o graph

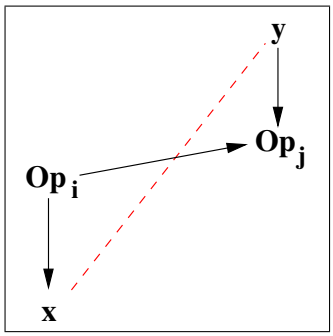


Figure 5.10. Refinement due to a Wait-for edge

the application of following rules:

WfRefinement rule:

$$\frac{Op_i \prec_w Op_j, \exists x \in Op_i^{\gg *}, y \in Op_j^{\ll *} : (x,y) \in M^o}{M^o \setminus \{(x,y)\}}$$

WfRefinement rule removes M^o edges which are positioned such as illustrated in Figure 5.10.

However, removing an M^o edge from the M^o relation may cause certain other M^o edges to be not feasible anymore. For such scenarios we define a predicate *balance(t)* where `t` is an M^o entry.

$$\begin{aligned}\texttt{balance}((S_{i,l}(j), R_{j,m}(i/*))) = \quad & \texttt{Let } R = \{x | x \in R^{\ll}_{j,m} \wedge x \equiv_t R_{j,m}\} \texttt{ in} \\ & \texttt{Let } S_x = \exists x \in R : \{y | y \in M^o(x), S_{i,l} \not\prec_{mb} y, S_{i,l} \neq S_{k,-}\} \texttt{ in} \\ & \texttt{Let } S = \bigcup_{x \in R} S_x \texttt{ in} \\ & \texttt{if}(\forall x \in R, S_x \neq \{\} \wedge |S| \geq |R|)\end{aligned}$$

The predicate *Balance* returns true when each receive in the ancestor of $R_{j,m}$ that either sources from P_i or is a wildcard has a unique potential matching send such that $S_{i,l}$ is not MB ordered w.r.t that potential matching send. When *balance* returns false, we have ascertained that M^o edge $\langle S_{i,l}, R_{j,m} \rangle$ is infeasible at runtime since one of the ancestors will be left without a single match which is a clear violation of MPI runtime operational semantics since, all receives sourcing from a single process must finish in FIFO order. The *balance* in the communication structure is violated because of the existence of Wait-for edges. We use the *balance* predicate in our rule 2 as follows:

Imbalance rule:

$$\frac{\exists m, m' \in \tau : a \in m, b \in m', (a,b) \in M^o, \neg balance((a,b)), m \prec m'}{M^o \setminus \{\langle a, b \rangle\}, (b,a) \in W^u}$$

In the *Imbalance* rule notice that we assume that event a matched before event b ($m \prec m'$) in a global timeline. This assumption comes handy in introducing the Wait-for edge from b to a. We call the M^o graph of a program to be *balanced* when all M^o edges satisfy the balance predicate.

Algorithm 7 gives the procedural view of applying the M^o, W^u construction rules and the refinement rules. Notice, that the algorithm maintains a transitive closure of M and W relations at all times. At the termination of this procedure we assert that $M = M^o$ and $W = W^u$.

Consider the example shown in Figure 5.9. Notice that after the removal of M^o edges $(S_{2,3}, R_{0,3})$ and $(S_{2,3}, R_{0,2})$, the M^o graph gets imbalanced. Observe that $(S_{1,1}, R_{0,3})$ and $(S_{1,2}, R_{0,5})$ are imbalance edges since if such matches were to manifest in an interleaving then $R_{0,1}$ will remain orphaned. Hence, after correctly removing the imbalanced edges, the correct M^o graph of the example is illustrated in Figure 5.11.

5.6 Proof of Correctness

We first show that the fix-point reached (the final M graph) in M^o graph refinement process is unqiue.

Algorithm 7 M-W Construction

1: **Input:**
2: `Trace:` τ
3: **Output:**
4: `Relation:` `M`
5: `Relation:` `W`

6: $C \leftarrow Compute(\tau)$ ▷ Apply C construction rules
7: $M^o \leftarrow Compute(C, M^o)$ ▷ Apply M^o construction rules
8: $W^u \leftarrow Compute(M^o, W^u)$ ▷ Apply W^u construction rules
9: $M \leftarrow M^o$
10: $W \leftarrow W^u$
11: **repeat**
12: $M^o \leftarrow M$
13: $W^u \leftarrow W$
14: $M, W \leftarrow Refine(M^o, W^u)$ ▷ Apply WfRefinement and Imbalance rule
15: $M, W \leftarrow TransitiveClosure(M, W)$
16: **until** $W^u = W$

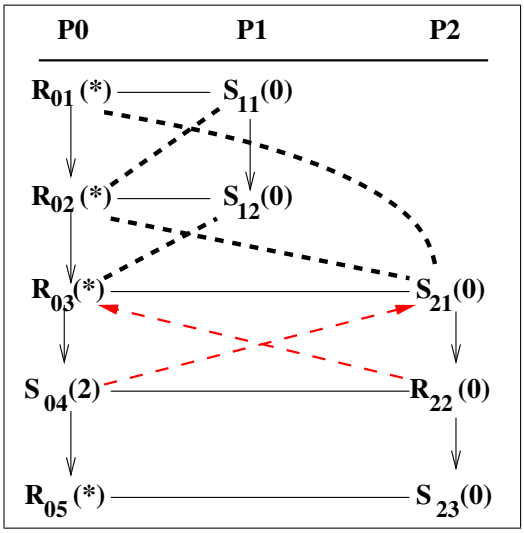


Figure 5.11. Final M^o of the example from Figure 5.9

Theorem 3 (Unique fix-point) *A unqiue fix-point,* i.e. *M graph, is computed after the termination of M^o graph refinement process.*

Proof : Notice that in all there are only three reduction rules to refine the imprecise M^o graph, viz., (i) The M^o consistency related rules F-rule and L-rule (on page 59), (ii) the WfRefinement rule (on page 63), and (iii) the Imbalance rule (on page 64). Notice, that the F-rule and the L-rule are part of the M^o construction process and thus are not concurrently

enabled with the WfRefinement and Imbalance rules. After the M^o refinement by M^o consistency rules, the resulting M^o graph may be imbalance. At this point, notice that both the WfRefinement rule and the Imbalance rule may be applicable. However, observe that Imabalance rule is non-conflicting with the WfRefinement rule. The only way these rules would conflict is when Wait-for edges are removed by either of these rules. Removal of Wait-for edges may make either of these rules in-applicable at a certain verification state of the rule-system. Since, these rules clearly do not remove any Wait-for edges the W^u relation, we can safely say that the two distinct reduction rules commute at all steps of the verification system.

Recall the according to the Church-Rosser theorem [8], a single normal form term is reached via a (possibily empty) sequence of reductions from a starting term when two distinct reduction rules are applicable from the starting term. In our case, the starting term is the starting state with an imprecise M^o graph and the reduction rules are the afore-mentioned 2 reduction rules. Then according to the Church-Rosser Theorem, we must have a single final term (M graph) reachable from the intial term (the imprecise M^o graph).

□

From the preceding discussion, we now know that there is only single M graph reachable from a our reductions. All we need to illustrate is that the final M graph, which is unique, does not have any false potential match edges. Lets start by identifying ways in which a certain M^o edge can be false under our construction set-up in a balanced M^o graph. Assuming that $ms, ms' \in \tau$ (where τ is a trace for an MPI program) and $ms \prec ms'$. Further, assume that $Op_{i,l} \in ms, Op_{j,m} \in ms'$ then if $(Op_{i,l}, Op_{j,m}) \in M^o \setminus M$ iff one of the following reasons are satisfied:

1. Exists $x \prec_w y$ such that $x \in Op_{j,m}^{\ll *}$ and $y \in Op_{i,l}^{\gg *}$.
2. Exists $Op_{j,m} \prec_w x$ such that $Op_{i,l} = F^i(x)$.

It is quite evident that if $(Op_{i,l}, Op_{j,m})$ is false then there must exist a Wait-for ordering. Such a Wait-for ordering can result either due to $Op_{j,m} \prec_w Op_{i,l}$ or due to $Op_{j,m} \prec_w x$ where x is another match for $Op_{i,l}$, however, with an extra constraint that $F^i(x) = Op_{i,l}$. It becomes evident that when $Op_{i,l}$ matches $Op_{j,m}$, x would be left orphaned as there is no match prior to $Op_{i,l}$. Since, $x \prec_w Op_{j,m}$, MPI runtime will always match x before $Op_{j,m}$

and hence the match $(Op_{i,l}, Op_{j,m})$ would never arise in any interleaving of the program. We show that *WfRefinement* refinement rule is sufficient to refine and remove all such false edges that arise in a balanced M^o graph. Furthermore, any false M^o edge in an imbalanced graph is rightly discovered by the *Imbalance* refinement rule.

Lemma 4 *Let in trace τ of the program there be $ms, ms' \in \tau$ such that $ms \prec ms'$. Assume that $Op_{i,l} \in ms$ and $Op_{j,m} \in ms'$. If the M^o edge $(Op_{i,l}, Op_{j,m})$ in a balanced M^o graph is false then there must exist the following relation in W^u: $Op_{j,m} \prec_w Op_{i,l}$.*

Proof : Assume $(Op_{i,l}, Op_{j,m})$ is the first false edge so far in the trace. If the edge is false then there must exist a Wait-for dependency from $Op_{j,m}$ to $Op_{i,l}$ (since $Op_{i,l}$ matched before $Op_{j,m}$) by definition. The question is whether the W^u construction rules can discover such a Wait-for dependency? As espoused earlier in this section, there are only two cases for the Wait-for dependency to exist. Let's evaluate the first case where in order for Wait-for dependency to exist from $Op_{j,m}$ to $Op_{i,l}$ there must exist operations $x \in Op_{j,m}^{\ll *}$ and $y \in Op_{i,l}^{\gg *}$ such that $x \prec_w y$. Under what scenarios such operations, x and y, can exist and what are the nature of these operations? The following text explains this in detail.

- **Direct Wait-for**: There is a direct communication between processes P_i and P_j such that the Wait-for dependency from x to y is a result of that direct communication, *i.e.*, x has matched the operation from P_i in the observed trace. Now for $x \prec_w y$, either $M^o(x) \subseteq y^{\gg}$ or $M^o(y) \subseteq x^{\ll}$. In either case, $y \prec_{lp} F^i(x)$. Thus, from Definition 7, it follows that $y \prec_{lp} F^i(Op_{j,m})$. Hence, the F-rule and L-rule of M^o construction and refinement will remove the edge $(Op_{i,l}, Op_{j,m})$ from M^o to begin with. If on the other hand, we could not establish $x \prec_w y$ despite the presence of a Wait-for from x to y then we know that $M^o(x) \not\subseteq y^{\gg}$ or $M^o(y) \not\subseteq x^{\ll}$. This makes it evident that an earlier M^o edge is false and $(Op_{i,l}, Op_{j,m})$ is not the first false M^o edge. We apply the same reasoning for all earlier false M^o edges.

- **Transitive Wait-for**: There is no direct communication that has taken place between P_i and P_j so far (*i.e.*, until ms' in the trace τ). However P_i and P_j have interacted transitively by engaging in communication with other processes. Thus, $x \prec_w w$ is a Wait-for edge that is transitively established from a series of direct Wait-for edges $x \prec_w y_1, y_1 \prec_w y_2, \cdots y_n \prec_w y$. Each of $y_1, y_2, \cdots, y_n$ is from a separate process involved in the transitive communication thread. For discovering each such direct

Wait-for and consequently the associated false M^o edge, we re-apply the same lemma (that is under discussion). For demonstration purposes, let's take a simpler instance where $x \prec_w y_1$ and $y_1 \prec_w y$ such that y_1 is an operation from P_k. Furthermore let $y_1 \in ms''$. If Wait-for dependencies were to exist the way we have assumed then $ms \prec ms'' \prec ms'$. If for some reason, the W^u construction rules could not establish $x \prec_w y_1$ or $y_1 \prec_w y$ then it is evident that $F^k(x) \in y_1^{\ll}$ and $F^i(y_1) \in y^{\ll}$. As long as the match-sets containing $F^k(x)$ and $F^i(y_1)$ have matched **later** than ms, we have correctly simulated $x \prec_w y_1 \prec_w y$ and will be able to successfully discover $(Op_{i,l}, Op_{j,m})$ to be false. On the other hand if that is not the case, then there is an earlier false edge waiting to be discovered by applying the exactly same reasoning.

Let's evaluate the other case when there is no genuine Wait-for ordering from x to y, yet $(Op_{i,l}, Op_{j,m})$ is false. Observe that we are still working with a *balanced* M^o graph. If we stick to the definition of Wait-for dependency then discovering that $Op_{j,m}$ and $Op_{i,l}$ are not Wait-for dependent implies there exists an interleaving where there are *co-enabled*. If there exists such a interleaving where they are co-enabled then there must exist a state in the execution where they match. However, knowing that $Op_{i,l}$ and $Op_{j,m}$ cannot match implies that there must exist some other ordering that disallows the match between $Op_{i,l}$ and $Op_{j,m}$. Notice that if $Op_{i,l}$ and $Op_{j,m}$ were a compatible match then one of these two operations must be a receive. If the receive is deterministic then only Wait-for ordering between them can be the direct Wait-for ordering. Therefore, if neither direct nor transitive Wait-for ordering is present between $Op_{i,l}$ and $Op_{j,m}$ then the receive must be a non-deterministic receive. Without losing generality, lets assume that $Op_{i,l}$ is the wildcard receive then $Op_{j,m}$ is a send targeting P_i. In such a case there is possibility that there exists an operation x from P_k such that $Op_{i,l} = F^i(x)$. It is also perfectly feasible that $Op_{j,m} \prec_w x$ because of direct or transitive communication between P_j and P_k. If such a Wait-for dependency is established from $Op_{j,m}$ to x then we have witnessed what we call **inter-process non-overtaking** ordering among sends that target the same destination process. Notice that in such a scenario even though $Op_{j,m}$ is not waiting on $Op_{i,l}$ it still cannot match $Op_{i,l}$ because if $Op_{j,m}$ were to match $Op_{i,l}$ then x must find a match prior to $Op_{i,l}$. However, $Op_{i,l}$ being the first match for x, operation x will remain orphaned. MPI runtime since always matches sends in a non-overtaking order thus would disallow the match between $Op_{i,l}$ and $Op_{j,m}$ in every interleaving. Notice that in order to establish $Op_{j,m} \prec_w x$, we again resort to the reasoning presented to discover direct or transitive Wait-for dependencies earlier.

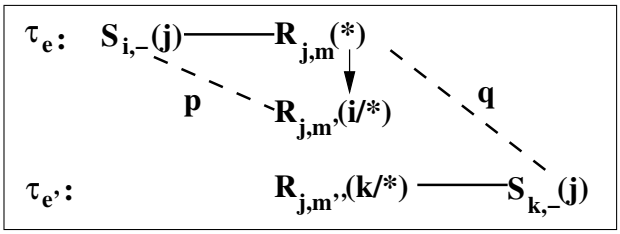


Figure 5.12. Corealizability of M^o edges

Thus, we have demonstrated that in a balanced M^o graph, the refinement rules will discover all the false M^o edges by first discovering the right W^u dependencies.
□

Lemma 5 (Imbalance M^o) *An M^o edge e may violate the balance predicate only after a false edge has been removed from the M^o graph by a prior application of* **WfRefinement** *rule.* **Imbalance** *rule is sufficient to discover such an imbalance.*

Proof : Assume that there is imbalance in the M^o graph with out any prior successful application of WfRefinement rule. This implies that no Wait-for dependency was discovered and the M^o graph constructed right after the first iteration was imbalanced which clearly violates the M^o construction rules. Hence, it a contradiction to the starting assumption. Thus, an imbalance in M^o graph can result only after a false M^o edge is removed. We now present the sufficiency of Imbalance rule to discover such an imbalance. MPI runtime allows a sends in the program to freely match receive calls as long as the *co-realizability* property is maintained. The M^o construction rules also respect this co-realizability property when constructing M^o edges.

Co-realizability property:

$$\tau_e \in \tau : \tau_e = (S_{i,-}(j), R_{j,m}(*)) \wedge (S_{i,-}(j), R_{j,m}(i/*)) \in M^o \wedge m' > m \Rightarrow$$
$$\exists \tau_{e'} \in \tau : \tau_e = (S_{k,-}, R_{j,m''}(k/*)) \wedge m' < m'' \wedge k \neq i \wedge (S_{k,-}, R_{j,m}(*)) \in M^o$$

Figure 5.12 illustrates this property. In the Figure, edge p is realizable iff edge q is realizable. Notice that when application of WfRefinement rule removes an edge like q then it is evident that edge p can no longer belong to M^o since $R_{j,m}$ will be left orphaned. Thus, removing p creates an imbalance which the Imbalance refinement rule correctly captures by comparing the cardinality of the set containing potential senders for receives prior to $R_{j,m'}$ (when we consider $S_{i,-}$ to be a match for $R_{j,m'}$) with the cardinality of the set of receives prior to $R_{j,m'}$.
□

Conjecture 6 (M^o completeness) *The M^o graph of the program obtained after the termination of Algorithm 7 has no false M^o edges,* i.e., *$M^o = M$.*

Proof : From Lemma 4 and Lemma 5 it is evident that when the fix-point is reached M^o will have no false edges and W^u will have no omissions. Since, certain parts of the proof fr Lemma 4 and Lemma 5 have yet to be formalized, we present this theorem as a conjecture which we strongly believe to be true from the partial proof of Lemmas 4, 5.

Corollary 7 (W^u Completeness) *The W^u relation of the program obtained after the termination of Algorithm 7 has no omissions,* i.e., *$W^u = W$.*

5.7 Conclusion

In this chapter we have presented the preliminaries and the rules to construct the potential match relation for each send and receive in the program after evaluating a single trace of the program. Furthermore, we demonstrate the inconclusiveness of *InterMB* ordering for predictive verification and present the rules to construct the global *interleaving-oblivious* orderings in the form of Wait-for dependencies and supplement them with additional rules to refine potential match relation and the Wait-for relation.

Discussion: We use the M^o and W^u relation to construct a deadlock detection strategy that operates in polynomial time. We present deadlock detection strategy details in the subsequent chapter. We further discuss the usefulness of these constructs to detect FIBs for the SOMM class of programs in Chapter 9.

CHAPTER 6

A PREDICTIVE POLYNOMIAL TIME DEADLOCK DETECTION ALGORITHM FOR MESSAGE PASSING APPLICATIONS

We present in this chapter the details of a deadlock detection strategy that operates in polynomial time. The deadlock detection strategy builds upon the work that is presented in Chapter 5. The strategy, as we present in the chapter, is sound and complete for a class of MPI programs that fall under the SOMM category (introduced in Section 5.1). We also illustrate the generality of this algorithm to be applied to any message passing system. Finally we present the results of this algorithm as a part of the tool MAAPED (Messaging Applications Analysis with Predictive Error Discovery) on several benchmarks.

6.1 Introduction

Deadlocks in MPI programs can occur because of variety of reasons. A significant number of these reasons cause an MPI program to deadlock in the first run of the program. For instance, supplying incorrect number of sends and receives in the program, passing incorrect arguments to the send/recv calls thereby leaving a certain communication operation orphaned, or having unsynchronized collective operations in the code are few reasons that cause the deadlock to manifest in the first run of the program. Any debugger would suffice to discover such type of deadlocks. Then there is whole another class of deadlocks which do no manifest on the first run or repeated runs of the program. The moment the program is ported to a different machine architecture the deadlock suddenly appears. The reasons for such a deadlock could be the following:

- The code is written with certain buffering assumptions which may no longer hold true when the program is ported on a different machine architecture. Consider the program in Figure 6.1. This program will deadlock when run on a machine that does not provide system buffering. This is because in the absence of system buffering both the sends act as a blocking receives.

P_0	P_1
$S_{0,1}(1)$	$S_{1,1}(0)$
$W_{0,2}(h_{0,1})$	$W_{1,2}(h_{1,1})$
$R_{0,3}(1)$	$R_{1,3}(0)$

Figure 6.1. Example with buffer dependent deadlock

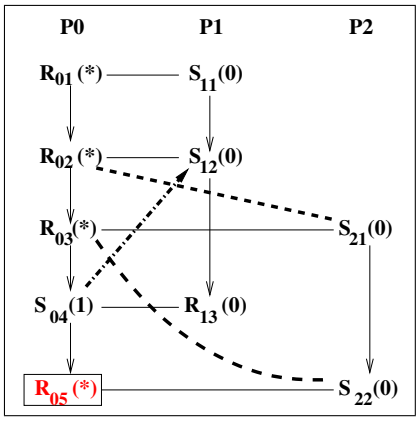


Figure 6.2. Deadlock due to Wait-for on Send

- Code has a non-deterministic receive that ends up consuming the sends meant for a deterministic receive appearing later thereby orphaning the deterministic receive call.
- Code has convoluted Wait-for dependencies that interact with certain non-deterministic receive calls causing a send to suddenly get disabled causing deadlocks that are deep seated in the schedule space. Consider the Figure 6.2. In this example, $S_{1,2}$ has matched $R_{0,2}$, however, there exists an alternate interleaving where $S_{1,2}$ can be successfully delayed until the control of process P_0 reaches $S_{0,4}$ at which point we witness a cyclic progress dependency creating a deadlock.

Notice that for such class of deadlocks, debugging technology would be inconclusive and fall far short of the goal. While there have been schedule perturbation solutions such as [81], however, such techniques rely on the right perturbation of the schedule to catch the deadlock, thus, they lack the completeness guarantee. Dynamic verifiers such as ISP and DAMPI, on the other hand, which rely on exhaustive verification of the schedule space, do detect such deadlocks though the time they take to find such deadlocks or otherwise prove deadlock freedom of the program is very high because of their innate strategy to examine a huge schedule space. We assert that for most MPI programs, we do not need such an expensive strategy to discover deadlocks. We provide an alternate strategy which precisely predicts the presence of a deadlock after evaluating a single schedule of the program. Such a predictive deadlock detection strategy relies on two important artifacts that we presented in the Chapter 5, namely, the potential match (M) relation and the Wait-for (W) relation.

6.2 Deadlock Detection Rules

We now present the rules for deadlock detection. The deadlock detection analysis proceeds by ascertaining whether the *communication opportunities are preserved*. We mean the following in regard to the preservation of communication opportunities:

- Under any execution scenario, each receive/send must find at least one matching send/receive. In other words, a deadlock is a state in an execution of the program where a receive/send from a certain process has not found a match irrespective of the progress of other processes in the system.

Note that any other deadlock (for instance, mismatched collective calls, wrong arguments to send and receive calls, incorrect number of send/receive calls) would be discovered in the first run of the program. Our focus is on "deep seated" deadlocks (deadlocks that do not manifest in the first run of the program). We have a two-step mechanism to detect deadlock due to the violation of preservation of communication opportunities.

1. A deadlock in the program due to orphaning of a deterministic receive which can be ascertained if the *last ordered potential matching send* in the M image of the deterministic receive can find a match in a receive preceding the deterministic receive under focus. Rule 1 formally captures this condition. Figure 6.3 illustrates such a deadlock scenario pictorially.

2. A deadlock in the program due to the orphaning of a deterministic/non-deterministic receive ($R_{j,m}$) which can be discovered by Rule 2. Rule 2 can be understood in a following manner: Assume the focal (orphaned) $R_{j,m}$ operation matched with $S_{i,l}(j)$ at the event τ_k of the observed execution trace. Further, assume $M_{low} \subset M(R_{j,m})$ is a set of sends that have matched with receive calls appearing later (in global time) than $R_{j,m}$. If $S_{i,l}$ and all the elements of M_{low} could be consumed by receives prior to $R_{j,m}$ then by pigeon-hole principle there must exist send calls to P_j which in the observed trace matched earlier than $R_{j,m}$ that must now find a match in $R_{j,m}$ and receive calls appearing later. However, if $R_{j,m}$ were to be orphaned then it implies that there exists at least one earlier send that cannot find a match in $R_{j,m}$ or later receives. This send becomes disabled for $R_{j,m}$ or later receives and therefore, must be a target of a Wait-for dependency. Hence, the deadlock due to an orphaned $R_{j,m}$ can precisely be caught by identifying such an orphaned matchable send earlier in the trace. This

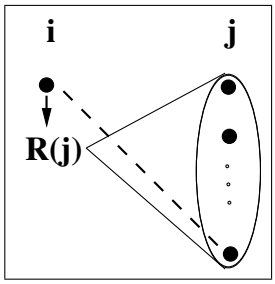


Figure 6.3. Orphaned deterministic Receive scenario

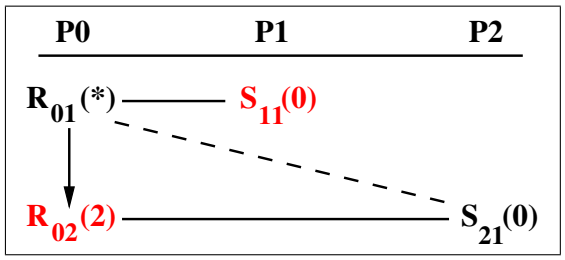


Figure 6.4. Example illustrating a deadlock despite No Wait-for dependencies

must imply that the orphaned send to P_j must be a target of a Wait-for dependency which disallows the send to match freely with $R_{j,m}$ or any receive appearing later than $R_{j,m}$.

A natural question after following the above discussion is: why scenario 2 is not sufficient since it appears that scenario 2 covers scenario 1 too? Notice that scenario 2 does not really cover scenario 1. Consider the example shown in Figure 6.4. In this example, none of the sends are target of a Wait-for dependency. They are enabled from the start state until they are consumed. At no point in the program they get disabled. However, a deadlock still results purely because of a deterministic receive's potential only match $S_{2,1}$ can be consumed by $R_{0,1}$ in an alternate interleaving. This example fits the description of scenario 1 only. The example thus asserts the need to separately deal with the two scenarios.

We now present the rules to discover such deadlocks.

Rule 1:

$$\frac{e \in [0,n] : \tau_e = \langle S_{i,l}(j), R_{j,m}(i) \rangle, \exists x \in L(R_{j,m}) : (x, R_{j,m'}) \in M : m' < m}{deadlock = true}$$

Rule 2:

$$\frac{\begin{array}{c} e \in [0,n] : \tau_e = \langle S_{i,l}(j), R_{j,m}(i/*) \rangle, (Op, S_{i,l}) \in W : Op \in R^{\gg}_{j,m}, \\ Discharge(S_{i,l}, R_{j,m}, Op, \hat{V}) \end{array}}{deadlock = true}$$

Rule 1 is fairly obvious to grap. In rule 2, note that we have used boolean function *Discharge*. This function returns true (indicating presence of a deadlock) when all receive calls (`R`) from $R_{j,m}$ onwards until Op (source of the Wait-for dependency) can be successfully be matched

Algorithm 8 Discharge Algorithm

1: **Input:**
2: Operation: S, R ▷ Send and Recv
3: Operation: Op ▷ Source of Wait-for edge
4: List: $\hat{V}$ ▷ List of matched sends up to (including) S
5: **Output:**
6: Boolean

7: **for all** R' from R until Op: $R \equiv_t R'$ **{**
8: $M' \leftarrow FindEnabledSend(M(R'), S, \hat{V})$
9: $S' \leftarrow Choose(M')$ ▷ Randomly choose a send
10: $result \leftarrow true$
11: **if** S' is Null **{** ▷ $Choose(M')$ that there is no send available
12: **return** false ▷ No deadlock until S'
13: **} else**
14: $\hat{V} \leftarrow \hat{V} \cup \{S'\}$
15: **}**
16: **}**
17: **return** true

Algorithm 9 FindEnabledSends Algorithm

1: **Input:**
2: Set: $M(R)$
3: List: $\hat{V}$
4: Send Operation: S
5: **Output:**
6: Set: M'

7: **for all** $s \in M(R)$ **{**
8: **if** $s \notin \hat{V} \wedge s \not\prec_w S \wedge \nexists(x \in M(R) : x \prec_{lp} s)$ **{**
9: $M' \leftarrow M' \cup \{s\}$
10: **}**
11: **}**
12: **return** M'

to sends other than $S_{i,k}$, else it returns false. When *Discharge* function returns true, it means that the process control of P_i and P_j have reached $S_{i,l}$ and Op respectively and $S_{i,l}$ has not found a match leading to cyclic Wait-for dependency between P_i, P_j.

In Algorithm 8, the procedure *FindEnabledSends* (line 7) refines the $M(R)$ by removing all sends from it that are (i) present in $\hat{V}$, (ii) Wait-for dependent on S, and (iii) from the same process barring the first ordered send from that process. The above procedure is responsible for discovering concurrently enabled sends that can match a certain focal

receive. Once the concurrent sends are discovered, we *arbitrarily choose* one send other than `S` (line 8) , add it to $\hat{V}$, and move on to the next MB ordered receive `R'` and treating `R'` as the focal receive.

6.3 Correctness Proof

We show that the deadlock detection rules presented in Section 6.2 are sound and complete.

Soundness: A deadlock is discovered by our algorithm under following situations:

- There exists a deterministic receive, $R_{j,m}(i)$ whose $L(R_{j,m})$ has a potential match to a prior receive $R_{j,m'}$. From Theorem 6, it follows that such a match is a true match. From the definition of potential match M, it follows that there must exist an execution, say e', where such a match is realized. Since, all sends targeting j are MB ordered, we infer that all sends $s \in S^{\ll} : S \in L(R_{j,m}) \wedge s \equiv_{t,d} S$ also match to receives prior to $R_{j,m}$ leading to a real deadlock.

- There exists an execution where a wildcard receive is orphaned which implies that there must also exist a matchable send that gets orphaned in the same execution. Such a send is a target of a Wait-for dependency which can be delayed sufficiently so that source of that Wait-for dependency is issued leading to a cyclic progress dependency. Note that *Discharge* function plays an actual partial trace (keeping the prefix of the trace fixed). At each step, the *Discharge* function only considers sends enabled for a particular receive by appropriately removing all the choices that were either already taken in the trace prefix, or are just not *enabled* with a focal receive. Thus, the deadlock discovered is a real deadlock.

Completeness: There are two parts to the completeness argument. Firstly, we have to show that our definition of deadlock covers all possible deadlocks in the program that has executed successfully in the first run. Secondly, the deadlock detection rules precisely covers this space of deadlocks.

Since the program ran successfully in the first run, we infer that program is well-formed. In a well-formed program, the only cause of deadlock is when a certain operation is orphaned (a send/receive in our case). Our definition of deadlock (at the start of the Section 6.2) precisely states that. Thus, we need to show that our algorithm covers this definition of deadlock in a complete manner. Note that whenever a send is orphaned in a well-formed

program implies there is a certain receive that is left orphaned. Our algorithm discovers such orphaned sends and have a specialized algorithm for deterministic receives. The proof of completeness, therefore, reduces to showing that our strategy for deadlock detection doesn't miss any deadlocks.

- From Theorem 7, we know that $W^u = W$. Thus, any Wait-for dependency targeting a Send is already computed as such. Rule 2 is, therefore, applied to each send that is a target of a Wait-for dependency. The question is then, whether the random choice of send (in the rule) to match a receive (line 8 in Algorithm 8) can mask a deadlock? Notice that a choice among sends to match a receive would matter only when that choice is no longer available for later receive operations. In other words, choice of send matters at a particular state in verification only if that send gets disabled and remains disabled for the rest of the states of the execution. Assume that a send $S_{k,l'}$ where $k \neq i$ is such a send where choice plays a role. Furthermore, lets assume during random selection of sends,$S_{k,l'}$, that may get disabled after certain state, was not selected by the *Choose* function. The fact that $S_{k,l'}$ gets disabled implies that it is a target of Wait-for dependency. Let the source of the Wait-for dependency is Op'. If $Op' \prec_{lp} Op$, *i.e.*, the source of Wait-for dependency to $S_{k,l'}$ precedes the source of Wait-for dependency to $S_{i,l}$ then the deadlock due to the orphaning of $S_{k,l'}$ is caught when Rule 2 is applied again with $S_{k,l'}$ as the focal send. If $Op \prec_{lp} Op'$ and if the deadlock is present then it will be $S_{i,l}$ which will be orphaned since the process issuing Op and Op' will issue Op prior to Op' and will wait for it to get matched. Any cyclic progress dependency would be discovered right at this point. Thus, we demonstrate that random selection of sends in the *Choose* function will not mask a deadlock.

6.4 Complexity Analysis

Assume that an MPI program is run on P many processes. Each process issue K many calls. Notice that in rule 1 (refer Section 6.2) $L(R_{j,m})$ can be obtained in constant time. The check whether $L(R_{j,m})$ matches with any ancestor of $R_{j,m}$ takes no greater time than $O(|M(L(R_{j,m}))|)$. Rule 1 can be applicable to $R_{j,m}$ only when it is a deterministic receive. Thus, $|M(L(R_{j,m}))| < K$ since every entry of the M set of the operation $L(R_{j,m})$ is from a single process. Therefore for $P \times K$ many instructions, rule 1's asymptotic upper bound time complexity is $O(P \times K^2)$. In rule 2, ascertaining the condition $Op \in R^{\gg}_{j,m}$ takes constant time. The only function that would consume a non-trivial time is the *Discharge*

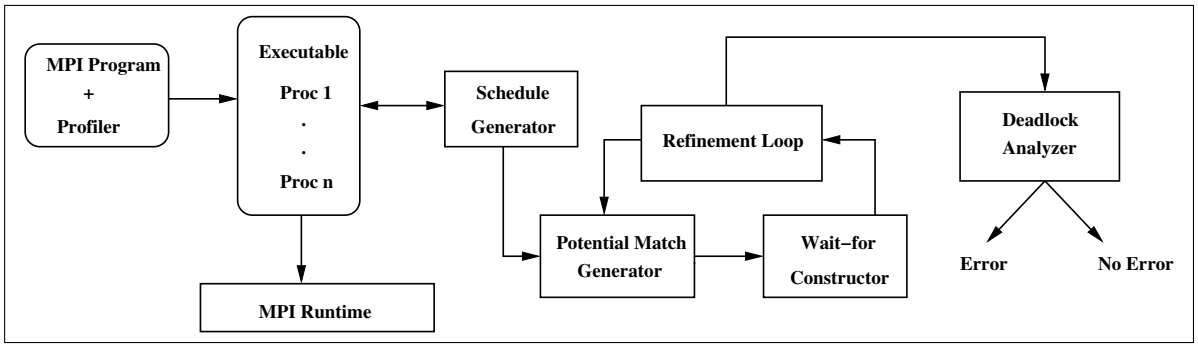


Figure 6.5. MAAPED Workflow

function. Within the Discharge function, the maximum size that M' can attain is K. Thus, the Discharge function would be called for at most K many times. For $P \times K$ many instructions, rule 2 would be fired for $\frac{P \times K}{2}$ many times. Furthermore, the function *FindEnabledSends* called with Discharge has a time complexity of $O(K^2)$. Hence, the total time worst case complexity for applying rule 2 for all the instructions in the program is $O(P \times K^3)$. Hence, the deadlock detection strategy is still polynomial in the number of processes and number of instructions per process.

6.5 Messaging Application Analysis with Predictive Error Discovery (MAPPED)

We present the tool flow of our predictive verification framework which we call as MAAPED. Figure 6.5 illustrates the components of the tool. The component *Scheduler generator* generates the first canonical interleaving exactly like ISP. The *Potential match generator* and *Wait-for constructor* apply the potential match relation and Wait-for rules respectively (presented in Section 5.3 and Section 5.4) on the trace. The *Refinement loop* is responsible for firing the refinement rules and finally after reaching the fixed point, the *Deadlock analyzer* fires the deadlock detection rules.

6.6 Results

The experiments were executed on Intel Core i7 quad-core with 8 GB of memory. We set a time limit of 2 hours to verify the benchmarks. We abort the verification process if the it did not complete within the time-limit. The benchmarks considered to demonstrate the notions discussed in Chapter 6 and Chapter 5 are the same that were used to testify the FIB work. However, we modified some of the benchmarks where dynamic load balancing

			Interleavings		Time(sec)
Benchmark	# of procs	Deadlocks?	ISP	MAAPED	MAAPED
Heat-diffusion	4	Yes	$0\times^{\dagger}$	$1\ \surd$	2.911
DTG-deadlock	5	Yes	$1\ \times^{\dagger}$	$1\ \surd$	0.009
Integrate_mw*	8	No	> 3500	1	1.669
Matrix Multiply*	8	No	120	1	4.564
Gaussian Elimination	8	No	$> 20,000$	1	2.68
Floyd Warshall	8	No	$> 20,000$	1	9.14

Table 6.1. Results for deadlock detection via predictive verification

takes place. This is because, MAAPED does not support dynamic load balancing based communication structure yet. The modified benchmarks are marked with the asterix (Matrix multiply and Integrate). We removed all reply-channel based communication (which is indicative of dynamic load balancing) from the codes and replace them with static work load assignments. Notice that the results shown in Table 6.1, some numbers are marked with †. In those experiments, ISP failed to catch the deadlock, however MAAPED discovered the deadlock.

Heat-Diffusion: It is a benchmark obtained from the SuperComputing 2011 tutorial presented by T. Hilbrich, G. Gopalakrishnan and others. The benchmarks solves the heat equation on a 2-D grid. Observe that ISP failed to execute even a single run for the benchmark Heat-Diffusion. However, when a certain optimization of ISP (*persistent-set* optimization for distinct DTGs) was turned off then ISP discovered the same deadlock that MAAPED discovered. When the example was executed on 4 processes, ISP discovered the deadlock in 7 interleavings and when the same process was run on 8 processes, ISP took over 2 hours and discovered the deadlock in 5041^{th} interleaving. MAAPED discovers the same deadlock by taking far lesser time (3 seconds when examined with 4 processes and 92 seconds with 8 processes).

DTG-Deadlock: This benchmark is a simplified version of the communication structure that exists in Parmetis [37].We also introduced a deadlock in to such a simplified example. A successful execution of this example is illustrated in Figure **??**. The deadlock will manifest only when seemingly two independent DTGs are both explored by ISP from a certain state. Exploring DTG2 before DTG1 one will enable $S_{2,2}$ to match $R_{0,2}$ leading to a cyclic progress cycle between $S_{0,2}$ and $S_{1,1}$. Like before, ISP could only discover the deadlock after the persistent-set optimization was turned off. MAAPED detected the same deadlock after evaluating a single run of the program in far fewer time.

Floyd-Warshall: This benchmark is obtained from [82]. It computes the all-pairs shortest

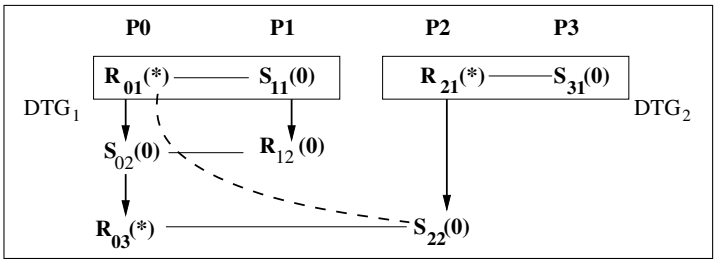


Figure 6.6. DTG-deadlock program trace

path algorithm given by Floyd and Warshall in a parallel fashion. Note that ISP did not complete in the stipulated time of 2 hours. We ran the verification separately for fewer processes to discover that code has no deadlocks. MAAPED verified the same program for all interleavings of the program in barely 10 seconds.

6.7 Discusson

What considerations/modifications we must apply to make the MAAPED framework work successfully for FIB detection? How can we relax the SOMM constraint on programs under test and develop predictive solutions utilizing the constructed artifacts M and W? We explore the latter question of relaxing the SOMM constraint in the Chapter 9.

The FIB analysis can be easily ported to the predictive analysis framework. Notice that we add Wait-for edges whenever a barrier match-set is witnessed in the trace of the program. All it requires is the slight modification of the *MB-Paths* definition (Definition 3 in Section 3.4). Instead of constructing MB paths from InterMB and IntraMB edges, we now construct the same MB paths from Wait-for edges and IntraMB edges. The rest of the FIB detection algorithm remains unchanged.

6.8 Conclusions

In this chapter, we have presented the rules to discover deep seated deadlocks in MPI programs. We further demonstrate the completeness of our deadlock detection strategy and illustrate that the rules operate in polynomial time complexity. We finally show the implementation of these rules in the framework MAAPED along with inspiring results on several benchmarks. All of the benchmarks belonged to the class of SOMM programs.

CHAPTER 7

MCC: A DYNAMIC VERIFICATION SCHEDULER FOR MCAPI APPLICATIONS

We present a dynamic direct code verification tool called MCC (MCAPI Checker) for applications written in the newly proposed Multicore Communications API (MCAPI). MCAPI provides both message passing and threading constructs, making the concurrent programming involved in MCAPI application development a non-trivial challenge. MCC intercepts MCAPI calls issued by user applications. Then, using a verification scheduler, MCC orchestrates a dependency directed replay of all relevant thread interleavings. This chapter presents the technical challenges in handling MCC's non-blocking constructs. This is the first dynamic model checker for MCAPI applications, and as such our work provides designers the opportunity to use a formal design tool in verifying MCAPI applications and evaluating MCAPI itself in the formative stages of MCAPI.

The purpose of this chapter is also to present the set of questions that every dynamic verification scheduler developer must ask before embarking on the effort to create a verification engine. We discuss some of the investigations pertaining to those set of questions in this chapter.

7.1 Introduction

It has been observed that the combined use of threading and message passing is necessary in order to create efficient multicore applications. This will require the standardization of an API for inter-core communication and synchronization. MCAPI [42] is one such effort which is under active development by a group of over 25 companies in the embedded system's market. Unlike large existing APIs like MPI [44] which target high-end compute clusters, MCAPI is designed keeping in mind the very specific needs and goals of embedded software/hardware system developers. MCAPI is aimed at programmers writing applications for embedded distributed systems employing loosely coupled cores. In particular,

MCAPI is well suited for systems that have much smaller memory footprints and are much more oriented towards reactive behaviors than computational. This paper describes the first direct code dynamic verification tool for MCAPI applications called MCC (MCAPI Checker). It takes as input a C code and verifies it directly. Therefore we resort to dynamic direct code verification methods that were originally pioneered in Verisoft [30]. Dynamic formal verification is witnessing ever growing presence in tools such as CHESS [52], Java Pathfinder [1], etc. In order to contain the thread interleaving explosion we use partial order methods that have been shown to be quite effective in software verification. MCC uses a customized version of dynamic partial order reduction (DPOR [25]) that is similar to the partial order with elusive interleavings (POE) algorithm explained in [69].

MCC builds on the strength of past projects namely ISP [2] and Inspect [**?**]. However, there are subtle differences between MCC, ISP, and Inspect. ISP is a purely an MPI verifier and Inspect is purely a shared memory thread program verifier. MCC, on the other hand, accommodates Pthread *create* and *join* calls as well as message passing based MCAPI calls. Furthermore, MCC differs from ISP in the manner in which non-determinism is handled in the input programs. ISP uses dynamic rewrite mechanism to force a deterministic match at runtime. MCAPI provides only non-deterministic receive calls, therefore, in the absence of specific receives the dynamic rewrite mechanism cannot work for MCC.

MCC supports "get/create" endpoint calls, connection-less blocking and non-blocking communication constructs and the "wait" call. The novelty of this lies in the way we enforce a deterministic match at the runtime. We discuss two solutions to enforce this determinism at runtime: (i) by intrusively modifying the MCAPI library and (ii) by inserting an implicit wait call in the instruction stream after a send and non-blocking receive pair has been given a go-ahead by the MCC scheduler.

Contribution: The contribution of this work are two fold. First, we have devised novel ways to enforce a deterministic match at the runtime; thereby avoiding the possibility of a communication race and the second is to pose a set of questions and pen our experience while building MCC which will be useful in building future dynamic verification engines.

7.2 Overview of MCAPI

The MCAPI effort traces its heritage to MPI and Socket communication libraries; however it differs from both with respect to the application domain it targets and the functionality it offers. MCAPI is less flexible than MPI (i.e., offers fewer functionalities as

compared to MPI). It is an API specification for the inter-core communication in a loosely coupled distributed embedded SoC.

MCAPI defines three communication types viz., connection-less datagrams, connection-oriented FIFO packet streams and connection-oriented FIFO scalar streams. MCAPI communication is performed by nodes which are abstract entities that could either be a process, a thread, a hardware accelerator or a processor core. Furthermore, nodes communicate with each other via endpoints that are the communication termination points. Endpoints are defined as a tuple of ⟨node id, port id⟩ pair. Each node can support multiple endpoints and every endpoint in the system is assigned a globally unique identifier. Since MCC currently supports only connection-less MCAPI constructs, we will therefore restrict the discussion in this chapter to only those API calls. The connection-less communication type of MCAPI is similar to MPI in that there is not static routing of messages. The API provides blocking and non-blocking variants of a send, receive, wait and test call to check the successful completion of non-blocking requests. An example code illustrating the usage of MCAPI calls in a C compilable code is shown in Figure 7.1. Each receiving endpoint is associated with a FIFO ordered receive queue.

7.3 Verification of MCAPI User Applications

We will stick to the same conventions for send, receive and wait calls as explained in earlier chapters with only slight modifications. A receive is denoted by $R_{i,l}(ep)$ is a receive posted by node i and expects a receive at the endpoint ep. A send denoted by $S_{i,l}(ep1, ep2)$ is a send posted by node i from endpoint $ep1$ targeting endpoint $ep2$.

Consider the example shown in Figure 7.2. While the runtime will always explore only one of the two possible execution scenarios, we must explore both the scenarios to guarantee program correctness. Any dynamic scheduler would make two match-sets for the first wildcard receive. $(S_{1,1}, R_{3,1})$ and $(S_{2,1}, R_{3,2})$. Assume that the scheduler decides to issue $(S_{1,1}, R_{3,1})$ in to the runtime. For the scheduler, the moment calls are signaled in to the runtime, the call have matched. This can be dangerous. Observe that immediately after the scheduler issues $(S_{2,1}, R_{3,2})$ in to the runtime expecting the previous match to have actually matched in the runtime. However, it is quite possible due to network latencies that the operations in the match-set $(S_{1,1}, R_{3,1})$ have really not matched by the runtime while $S_{2,2}$ is also issued to the runtime. This would initiate a communication race amopnd $S_{1,1}$ and $S_{2,2}$ in the runtime to match $R_{3,1}$. Notice that MCAPI does not provide a deterministic variant

```
1:#define NUM_THREADS 3
2:#define PORT_NUM 1

3:void* run_thread (void *t) {
4:    thread_start();
    ...
5:  mcapi_initialize(tid,&version,&status);
6:  if (tid == 2) {
7:  recv_endpt =
    pmcapi_create_endpoint (PORT_NUM,&status);
8:  pmcapi_msg_recv(recv_endpt,msg,
                  BUFF_SIZE,&recv_size,
                  &status);
9:  pmcapi_msg_recv(recv_endpt,msg
                  BUFF_SIZE, &recv_size,
                  &status);
10:   } else {
11:  send_endpt = mcapi_create_endpoint
                  (PORT_NUM,&status);
12:  recv_endpt = mcapi_get_endpoint
                  (2,PORT_NUM,&status);
13:  pmcapi_msg_send(send_endpt,recv_endpt,
                  msg,strlen(msg),
                  1,&status);
14:  }
15:  pmcapi_finalize(&status);
     ...
16:  thread_end();
     }

17:int main () {
     ...
18:  main_thread_start();
19:  for(t=0; t<NUM_THREADS; t++){
20:    rc = mcapi_thread_create(&threads[t], ...)

21:  }
22:  for (t = 0; t < NUM_THREADS; t++) {
23:    mcapi_thread_join(threads[t],NULL);
24:  }
25: main_thread_end();
    ...
    }
```

Figure 7.1. An instrumented MCAPI example C program

of a receive call. Thus, unlike ISP, we can dynamically re-write the wildcard receive calls. Observe the gravity of the situation, we have a scenario where scheduler decides a certain match-set to match in the runtime but the runtime decides to match another match-set.

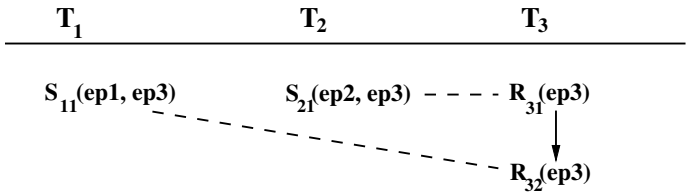


Figure 7.2. MCAPI Receive Nondeterminism

This will lead to a broken analysis of the dynamic scheduler. The big question is then the following: **How can a dynamic verification scheduler ensure that the runtime respects the order of match-sets that the scheduler decides?**

We deviced two novel ways with with the runtime determinism could be established.

- **Receive buffer probes**: We augmented the MCAPI library with an extra call `MCAPI_Probe_Endpoint(MCAPI_Endpoint,MCAPI_Status)`. This call served as a hook in to the MCAPI runtime. The function returned the endpoint pointer of the sender whose data payload is at the top of the receive queue at the endpoint supplied as the argument to this function. The MCC scheduler after signalling a match-set involving a non-deterministic receive, probes on the endpoint of that receive call until the sender belonging to the signalled match-set makes an entry in to the receive queue. Since the queue is FIFO ordered, the scheduler can safely decide to compute the next match-set and signal them to the runtime. Note, however, that for such an policy to be applicable, the scheduler should itself act as an MCAPI node, since, in order to probe it will have to issue the augmented MCAPI probe call. Secondly, this is quite an intrusive solution. We are suggesting a change in the MCAPI library with a function call that is not even a part of the standard. Furthermore, it is possible that the library's souce code may not be available to the developer of the verification engine.

- **Wait introduction in the program instruction stream**: This solution is non-intrusive as opposed to the previous solution. In this solution we remove the distinction between the calls getting matched and the calls getting completed. The scheduler after deciding to signal a match-set in to the runtime which invovles a non-blocking receive, waits until the non-blocking receive has completed. This wait is acheived by introducing an extra wait in to the instruction stream of the MCAPI application. The wrapper call of the non-blocking receive call after getting a signal from the scheduler

calls an additional wait instruction. In other words, we have implicitly transformed non-blocking receive calls in to blocking receive calls. Note that this will not affect the communication structure of the original program other than the performance hit.

The MCC scheduler adopts the second solution since it is non-intrusive.

7.4 MCAPI Checker (MCC) Overview

MCC is based on the current reference implementation of MCAPI provided by the MCA. The reference implementation uses Pthreads and a thread describes the notion of a node. Communication is performed only after a node has successfully issued MCAPI_INITIALIZE. It is an error to issue a communication call after a node has performed an MCAPI_FINALIZE. We have identified a list of safety properties that are important to ensure a correct and safe use of the API. For instance, invoking a communication call without creating valid endpoints or accessing the data buffer (passed to a non-blocking call) before the corresponding wait operation is issued are few of the conditions that violate the correctness of an MCAPI program. A list of default usage properties are compiled in [43].

Figure 7.3 describes an high level work-flow of the MCC tool. MCC has three components. The first component instruments an input MCAPI C user program at compile time. As a part of the instrumentation process all the MCAPI calls along with the Pthread create/join calls are rename (by prepending the character "p"). These instrumented calls serve as wrappers to the actual MCAPI calls. Additionally, the thread function bodies are enveloped within the calls *thread_start* and *thread_end* and the main thread is instrumented with a *main_start* and *main_end* call. Figure 7.1 shows a snippet of instrumented C code that has the same communication pattern as depicted in Figure **??**. Note that thread function body is instrumented with a *thread_start* (line 4) and a *thread_end* (line 16) call. The *thread_end* call notifies the scheduler that thread count, a piece of information noted by the scheduler before processing any instrumented call, should be decremented by one. The thread count helps the scheduler to determine when all threads have blocked. The *thread_start* call acts as a barrier (global fence) operation. In other words, all the threads (except the main thread) have to issue the *thread_start* call before any thread can proceed with its execution. The main thread is also instrumented with a *main_thread_start* and a *main_thread_end* call (lines 18, 25). These calls notify the scheduler of the start and end of the verification process. Additionally, the traditional Pthread create and join calls are also instrumented. The primary reason for create/join call instrumentation is to ascertain the

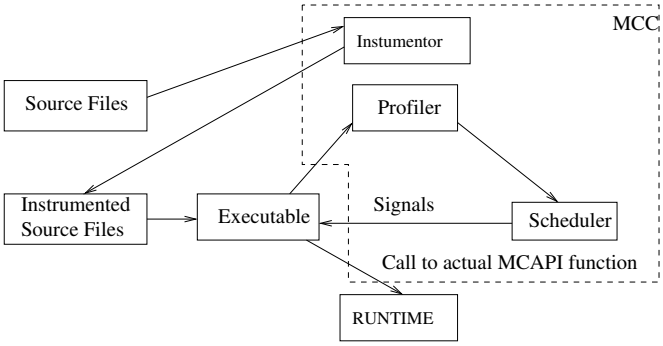


Figure 7.3. MCC workflow

total number of nodes in the system before each node starts issuing MCAPI communication calls. We assume no dynamic creation of threads. All the MCAPI related calls are replaced with the wrapper calls that are defined in the profiler component of MCC.

The second component of MCC is the profiler that has function definitions of the instrumented calls. The profiler functions perform the necessary book-keeping and communicate the information collected to the scheduler. The functions block until they receive a signal to continue with the execution from the scheduler. The profiler wrapper functions eventually issue the actual MCAPI calls to the runtime. The third component of MCC is the scheduler that ultimately decides which calls should be issued to the runtime and subsequently signals the blocked threads to unblock and execute those calls.

The scheduler explores all the independent thread steps in a single non-commutative canonical order while commuting all dependent co-enabled thread steps resulting in the exploration of a reduced state space that is a valid partial order reduction of the complete state space. The MCC scheduler accommodates receive non-determinism by delaying (dynamically re-ordering) the processing of receive calls until all sends that can potentially match the receives are dynamically discovered. Each such send-receive match is explored in separate runs of the program (these matches form the *persistent-sets*).

7.4.1 MCC Scheduler Explanation Through an Example

The MCC scheduler unlike the ISP scheduler does not perform dynamic re-writing because MCAPI does not provide specific source point receives; meaning that one cannot designate where one would like to receive from. The scheduler is able to perform dynamic re-ordering of calls by first discovering all pending calls and then issuing matched calls sequentially to the run time and inserting waits when needed in non-blocking semantics.

T_1	T_2	T_3
$R_{1,1}(e1)$	$S_{2,1}(e2, e1)$	$R_{3,1}(e3)$
$S_{1,2}(e1, e3)$	$W_{2,2}(h_{2,1})$	$S_{3,2}(e3, e1)$
$R_{1,3}(e1)$		
$W_{1,4}(h_{1,3})$		$W_{3,3}(h_{3,2})$

Figure 7.4. Re-ordering Example

While an MCAPI node (i.e. a thread w.r.t. the reference implementation) would issue the calls in program order, the MCC scheduler can permute the order of these calls without introducing any new behaviors in the program.

Consider the example shown in Figure 7.4 where the MCC scheduler re-orders the calls. Threads T_1, T_2 and T_3 are blocked at the $W_{1,3}$, $W_{2,2}$ and $W_{3,3}$ calls respectively. The match-sets formed by the scheduler at this point are $\langle S_{1,2}, R_{3,1}\rangle$ and $\langle R_{1,1}, S_{2,2}\rangle$. As the wait call for $R_{1,1}$ is not yet seen, the recv call is not obliged to finish before $S_{1,2}$ call.

Note that signaling the match-set $\langle S_{1,2}, R_{3,1}\rangle$ to runtime enables $S_{3,2}$ call which is another potential sender to the call $R_{1,1}$. Hence, signaling the match-set $\langle S_{2,1}, R_{1,1}\rangle$ to the runtime before the match-set $\langle S_{1,2}, R_{3,1}\rangle$ would lead to incorrect verification results. Noting this fact, the scheduler should signal a go-ahead to $S_{1,2}$ call first thus permuting the issue order different from the program order.

Figure 7.5 illustrates an interleaving scenario as a time-line based sequence of message interactions between the scheduler and the threads of an MCAPI user program (from Figure 7.4). The user program is branched off as a separate thread under the controlled environment of the scheduler. The main thread of the instrumented program issues thread create calls which when signaled to go-ahead by the scheduler, create threads T_1, T_2, and T_3. Note that the main thread blocks at the first *thread_join* call. Threads T_1, T_2, and T_3 are all blocked at their respective *thread_start* calls. The reason to have a *thread_start* call is explained in Section 7.4.2. The scheduler then unblocks the threads T_1, T_2 and T_3 after ascertaining a count of the total number of threads alive in the system. The threads continue to run and issue calls until they have hit their fence operations (blocking calls). At this point the scheduler has seen the following operations: (i) Until $W_{1,4}$ from T_1; (ii) Until $W_{2,2}$ from T_2; and (iii) Until $W_{3,3}$ from T_3. The scheduler has come across a decision point and subsequently forms match-sets from the list of enabled transitions. Scheduler issues the signals to go-ahead to the match-sets and subsequently spin-loops until the recv call in the match-set completes before signaling a go-ahead to the next match-set. The box in the

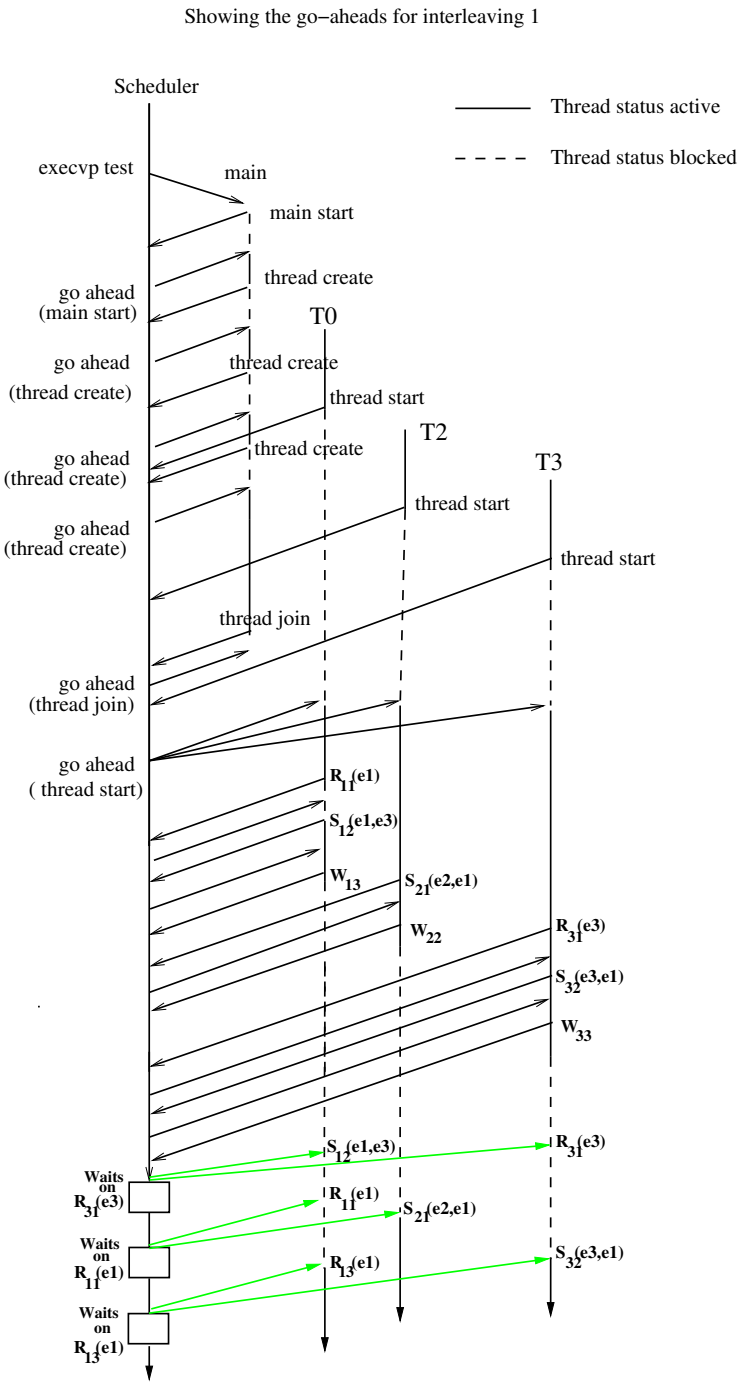


Figure 7.5. Interactions of the scheduler with the example from Figure 7.4

timing diagram of Figure 7.5 represents this spin-loop.

The main thread unblocks following the completion of the *thread_end* calls and the program runs to completion.

Algorithm 10 MCC scheduler pseudocode

```
1:  GenerateInterleaving( ) {
2:      while (1) { // Computes the total number of threads alive
3:          t_i = Obtain_transition ();
4:          if (t_i is thread_create) {
5:              num_threads++;
6:              signal go-ahead to thread_of(t_i);
7:          }
8:          if (t_i is thread_join || t_i is MCAPI communication call by thread "main") {
9:              signal go-ahead to thread i;
10:             break;
11:         }
12:         if (t_i is thread_start) {
13:             update the status of thread i to blocked;
14:         }
15:     }// while (1) ends here

16:     count = num_threads;
17:     signal go-ahead to all the blocked threads;

18:     while (count) { // till no more threads are alive
19:         for each (runnable thread i) {
20:             t_i = receive_transition from thread i;
21:             update transition_list of thread_of (t_i) in the S_curr;
22:             if (t_i is of blocking_type) {
23:                 update the status of thread i to blocked;
24:             }
25:             if (t_i is of type thread_end) {
26:                 count --;
27:             }
28:         }
            // All threads are blocked here
29:         while (no thread is runnable) {
30:             find_matchset ();
31:             unblock the threads owning transitions in the above match-set;
32:         }
33:     }// while (count) ends here
34: }

35: check_for_runtime_race( ) {
36:     if (any t_i ∈ current_match-set races with non-blocking call from prev_match-set) {
37:         while (non-blocking call is completed); {
38:         }
39:     }
40: }
```

7.4.2 MCC Scheduler Algorithm

Algorithm 10 in Section 7.4.2 explains the working of the scheduler. The MCC scheduler works under certain assumptions. It assumes that all threads of the system are created at the outset of the program. The MCC scheduler *must know the total thread count in the system to determine when all threads have blocked.* As such, MCC count threads as they

Algorithm 11 Find a suitable match-set

```
 1: find_matchset( ) {
 2:     Store the computed match-sets in ample_set of S_curr;
 3:     if (ample_set is not empty) {
 4:         for each (t_i in head_element of the ample_list) {
 5:             check_for_runtime_race();
 6:             give a go-ahead to thread_of (i);
 7:         }
 8:         remove head_element from ample_set;
 9:         copy the ample_set in S_next;
10:         return;
11:     }
12:     flag that a deadlock found;
13: }
```

are created by the main thread, and blocks them on their thread-start calls until the main thread either invokes an MCAPI call or a thread join call. At that point, MCC assumes the total number of threads to be those already created and starts all the created threads running. After ascertaining the thread count, the scheduler liberates all the blocked threads (line 17) and starts receiving transitions from all runnable threads until the next decision point is hit. Note that if a thread issues a *thread_end* call, the thread count of the system is decremented (lines 18-28).

Once a decision point is hit, the scheduler then computes the match-sets from a list of enabled transitions. It then selects one match-set and liberates the participating threads in that match-set (lines 29-32). A match-set consists of either a send-receive call pair, or a single entry comprising a wait call. The enabled transitions are computed with the help of the Intra-HB relationship that is maintained for each state of the scheduler. The priority order for evaluating these match-sets is the following: (i) enabled wait call (ii) and then the send-receive match-set.

The MCC scheduler also handles *get_endpoint* and *create_endpoint* calls. When a thread issues a *create_endpoint* call, the scheduler looks to see if any blocked thread (on *get_endpoint* call) was waiting for it. If so, the *create_endpoint* call and the blocked *get_endpoint* call are both signaled to go-ahead. If that is not the case, then the scheduler stores the created endpoint in an auxiliary table. When the scheduler encounters a *get_endpoint* call then it first looks up the table of created endpoints. It blocks the thread if the sought endpoint is not created. Otherwise, *get_endpoint* call is immediately signaled to go-ahead.

Every decision point advances the state of the scheduler. The match-sets for a state under exploration are stored in a separate data structure (*persistent-set*). Every state

has an persistent-set associated with it. One entry is selected from this ample-set for the go-ahead. Subsequently, the match-set entry that has been recently liberated is removed from the persistent-set. The updated persistent-set is then copied to the next state. Note that only the first interleaving builds the per-state persistent-set. The scheduler declares a deadlock in the code if at a state the persistent-set is found to be empty while there are still runnable threads in the system (lines 41-52).

A safety check is performed before the participating threads can be given a go-ahead. This safety check ensures that a deterministic match manifests at runtime and the transitions of the match-set in the current state (S_{curr}) do not race with the transitions from the match-set in the previous state (S_{prev}). In the case when a race is found then the scheduler spin-loops until the racing transition from S_{prev} is completed by repeatedly testing the request handle of the racing transition. Only after the completion of the racing transition is the current match-set processed (lines 35-40). Later if a wait call is observed by the scheduler for the completed racing transition, it is still issued to the runtime, however, it will return immediately.

The procedure *GenerateInterleaving* is called in a loop until there are no more replays to be performed. The decision whether to perform a replay is made by inspecting the persistent-set of the visited states in the stack. If for each state the persistent-set is found to be empty then the scheduler has explored all the relevant interleavings.

Discussion: The MCC scheduler explores all the interleavings which are resulting from the connectionless wildcard receive calls of MCAPI that are supported by MCC. Thus, Being a a dynamic strategy, MCC is guaranteed to discover deadlocks and safety violation assertions soundly. Furthermore, it also offers the completeness guarantee over the schedule space resulting from the use of wildcard receives. We now present some of the important questions that we came across while constructing dynamic verification engines for Message Passing systems.

- Should the developer of the verification scheduler insert hooks in to the API's runtime or use the API calls which may manipulate the semantics of the program?
- How can a scheduler enforce determinism in the event of a communication race? Enforcing determinism may require controlling the runtimes of multiple APIs.
- Would it be rather convenient to have a trace-based order-replay scheduler as opposed

to a stateless order-replay scheduler?

- Should the dynamic process/thread/node creation be considered important for application verification or would it suffice to verify an application with fixed nodes?

7.5 Results and Concluding Remarks

We have developed the first dynamic verification engine for MCAPI user applications that currently handles blocking and non-blocking connection-less communication constructs of the MCAPI reference implementation. Since no publicly available benchmark using MCAPI is currently available, we tested MCC successfully on small test examples constructed by ourselves. For instance, the example program from Figure 7.4 was verified in 2 interleavings in a fraction of a second. We are currently working to extend MCC to support the full set of MCAPI calls. Future works involves exploring solutions to verify programs that have subtle bugs, for instance, data-races in unison with the MCAPI non-determinism. We acknowledge Jim Holt from Freescale for his help on this work.

CHAPTER 8

RELATED WORK

In this chapter, we provide a general summary of research that has taken place in the area of MPI application verification, particularly in those areas which have a significant overlap with the solutions that we have investigated in our dissertation.

8.1 Correctness and Verification Tools in MPI

Let us first evaluate the space of correctness checking tools. These are the type of tools that check for runtime errors of an MPI program by examining only the current trace of the program which is under execution. Such tools are not sufficient to explore alternate schedules of the program. A detailed survey of correctness checking tools and debuggers can be found in [58]. We briefly list some of the correctness tools in the following text:

- **MPI-CHECK**: MPI-CHECK supports only FORTRAN 90 programs. The version that supports C/C++ is under development. MPI-CHECK does not use the MPI profiling Interface to capture the calls and analyze them; instead, using a macro-like mechanism wherein the MPI calls in the program are instrumented to have extra arguments. These arguments provide information such as line number in the source code where the call was made, the MPI function name and its arguments. The information is stored in a database known as the Program Database (PDB). The process of checking is split in to two phases. In phase one, instrumentation of MPI programs is performed followed by their compilation. In phase two, execution of the instrumented MPI code under the control of the MPI-CHECK server takes place. The errors captured by MPI-CHECK as explained in [39] are incorrect usage of MPI calls, exceeding buffer bounds, and deadlocks.

- **MARMOT**: MARMOT is a tool that analyzes MPI programs by trapping communication calls using the MPI profiling interface. It performs all argument verification like tags, communicators, ranks, etc. locally on the client side. MARMOT also detects

potential and real deadlocks. However, the mechanism employed to detect deadlocks is different from that of MPI-CHECK. In MARMOT dependency graph is not created. Instead, a time-out mechanism is used to conclude the presence of a deadlock. Some of the checks performed by MARMOT as explained [38] are: MPI type errors, resource leaks, deadlocks, erroneous use of MPI I/O.

- **UMPIRE**: UMPIRE, developed at LLNL, is another MPI program correctness checker. It is a tool that dynamically analyzes MPI programming errors using MPI profiling interface. It performs checking at two levels. Firstly, it checks at the local level where it uses all the task-local information to perform the checks. For instance, tests regarding the checksum on non-blocking send buffers can be carried out at this level. The second check is performed at a global level. It digs out more subtle errors like deadlocks, consistency errors, and type mismatches at the global level. UMPIRE uses time-out mechanism and dependency graphs to detect deadlocks. Complete operational details regarding UMPIRE can be found in [74].

- **MPIDD**: MPIDD, like UMPIRE has a central manager that traps all MPI calls using the MPI profiling interface (PMPI); however UMPIRE runs as a separate process and communicate using shared memory with different processes. MPIDD runs as another MPI process and the trapped information is sent to the central detector using MPI calls as explained in [34]. MPIDD is essentially a deadlock detection tool. It creates a dependency graph to figure out potential/real deadlocks. The detection algorithm is a Depth First Search for cycles in the dependency graph. The architecture of MPIDD suggests that it should be able to do all the argument verification tests that other tools perform.

- **MPIRace-Check**: It is a tool that identifies communication race among sends vying to match a non-deterministic receive. MPIRace-Check [48] uses vector clocks to discover such racing sends. MPIRace-Check does not have the ability to deterministically replay the program unlike the verification tools that we will discuss shortly. Since, vector clocks are used, MPIRace-Check have scalability issues.

- **Intel Message Checker**: Intel Message Checker [14] (IMC) is an MPI correctness tool which has a centralized mechanism to detect errors/deadlocks like MARMOT and UMPIRE. However, UMPIRE and MARMOT are purely runtime checking tools.

IMC, on the other hand is a post-mortem analyzer. The component of IMC called TRACE col- lector, collects information of each MPI call in a trace file using a library file libVTmc.so which is similar to the PMPI interface. This trace file is then analyzed by a checking engine after the execution. IMC also provides a visualizer to examine the output of the analyzer. IMC checks for type errors, resource leaks, deadlocks and unsafe buffer uses in the program. IMC can suffer from several impediments. The trace files generated can be large. Furthermore, the generation of trace files in the presence of an MPI error cannot be guaranteed, as the behavior after an MPI error is implementation defined.

Unlike correctness checking tools, verification tools have a scheduler that orchestrates various interleavings to exhaustively examine the relevant scenarios of the program. Verification tools provide a guaranteed coverage of MPI programs over the space of non-determinism. To the best of our knowledge, the two dynamic verification tools for MPI are ISP [69] and DAMPI [77]. MPI-SPIN [61] is the only model-checker for MPI programs which operates on user built models of MPI programs. MPI-SPIN models are written in the extended SPIN language. MPI-SPIN suffers from scalability issues and can be applied to only very small programs.

MAAPED is the only predictive verification tool in the MPI application landscape that offer similar coverage guarantees as dynamic verification tools for SOMM (Sender Oblivious Message Matching) class of programs. In future work, we discuss ways to extend the MAAPED work so that the predictive verification methodology is applicable to class of programs wider than SOMM class.

8.2 Tools for Checking MCAPI Applications

In the space of MCAPI programs, MCC [56, 55] is the first dynamic verifier. MCC is very similar to ISP in operation and borrows concepts from ISP and Inspect [**?** **?**]. The only other tool that performs deterministic replay of MCAPI programs is DR-MCAPI [17]. Its functioning is similar to that of MCC while the end goal is concerned. There are, however, operational differences in MCC and DR-MCAPI. While DR-MCAPI records the trace and performs order-replay. MCC does not record any trace. DR-MCAPI, being more recent, supports wider number of MCAPI calls (MCAPI test and MCAPI wait_any) as opposed to MCC which is not actively supported anymore.

In [16, 18], authors have presented a symbolic debugger for correctness checking of

MCAPI applications knows as CRI. The CRI tool obtains a trace of an MCAPI program execution and builds an SMT [4] formula which is then fed to popular back-end decision procedure such as Yices [**?**] check for common errors such communication races and assertion failures. It does not have the ability to replay the program and suffers from similar problems that IMC (discussed earlier) suffers from. Another work related to symbolic analysis of MCAPI applications is presented in [24]. This work is related to CRI to a certain extent. They have symbolically modeled the MCAPI program after observing a single execution trace. However, instead of restricting their reasoning to the observed trace, the work can also reason about other execution schedules where the sequence of conditional branch outcomes are same as the one observed in the execution trace. The work in [24] has a shortcoming that authors themselves have noted which is that the technique for SMT formula generation in their work is prohibitively expensive in computation time.

8.3 Related Work in Barrier Analysis

To the best of our knowledge, the work on FIB detection is the only one work in the MPI landscape that soundly and completely discovers all the collective barrier operations that are either relevant/irrelevant or a cause of deadlocks in MPI programs. The only other work that we are familiar with which deals with program written SPMD style is by Aiken etal. in [5]. This work however, is applicable to programs written in Split-C developed in Berkeley. Their algorithm is to statically infer whether or not the textually unaligned barriers in the program are correctly synchronized. A remotely related work in [50] detects barriers that are cause of a deadlock in the actual run of the program and visualizes them in the Eclipse IDE. This work is a part of the debugging facility provided by the PTP (parallel tools platform) of Eclipse.

If we move to the domain of multithreaded applications then there is a vast body of work that has investigated the problem of erroneous barriers. The work in [**?**] is one of recent efforts to statically identify mismatched barrier which are textually aligned. Notice, that the essential work that all the earlier research, regardless of the domain (shared memory or message passing), are trying to solve is the barrier matching problem in order to discover the deadlock due to ill-synchronization at compile time. Our FIB work, on the other hand, not only discovers ill-synchronization of barrier (regardless of whether they are textually aligned or unaligned), it also discovers which set of matching barriers are irrelevant.

CHAPTER 9

CONCLUSIONS AND FUTURE DIRECTIONS

Verification of programs that are constructed using message passing libraries with non-deterministic constructs is not only essential but also the only option for obtaining coverage guarantees. However, most verification tools in this domain explore the whole schedule space of programs in an indiscriminate fashion. We demonstrate in this dissertation that it is un-necessary for a large class of SPMD styled programs to explore the whole schedule space. For such a class of programs we have investigated two methodologies and shown their effectiveness to verify programs for the presence of deadlocks in far fewer interleavings and in much less time.

We first presented the MSPOE algorithm (implemented on top of ISP) and its effectiveness to prune the schedule space for several benchmarks. The MSPOE algorithm could have very well be implemented on top of other dynamic verification schedulers such as DAMPI without any changes to the algorithm. We then presented a generalized Matches-Before framework which was utilized to construct the predictive deadlock detection framework called MAAPED. We sketched the soundness and the completeness proof of the generalized Matches-Before constructor and presented the results of the polynomial time deadlock detection strategy on several benchmarks and compared them to ISP's results. As previously stated with respect to the MSPOE work, the predictive verification strategy of MAAPED can also be built on top of DAMPI verification scheduler with out significant algorithmic changes. We finally relayed some of our experiences while building dynamic verification scheduler for message passing library MCAPI.

Both MSPOE and MAAPED algorithms discover deadlocks cheapily. MSPOE has the advantage of simplicity of implementation, however, its results are incomplete. MAAPED, on the other hand, subsumes MSPOE results. MSPOE can handle reply channel based communication which MAAPED, at the moment, cannot. MSPOE, theoretically can end up exploring a exponential schedule space while MAAPED only explores a single trace

and discovers deadlocks in polynomial time. As a recommendation to a potential user of these algorithms, we suggest a portfolio approach (run both algorithms) by evaluating which algorithm can fit the constraints that the program offers. We also presented an algorithm to improve the performance of MPI applications by removing global synchronization operations (MPI barriers) that were discovered to be irrelevant. We further present the soundness and completeness proof of the algorithm and present some results on various benchmarks.

9.1 Future Research Directions

9.1.1 Proof for the Conjecture

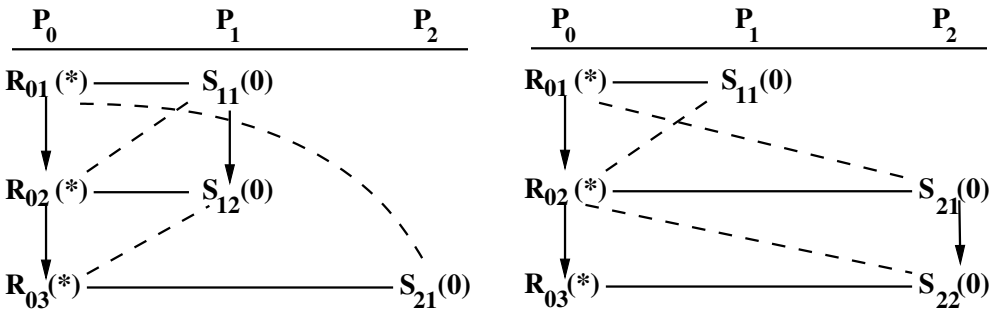
We strongly believe in the conjecture that we presented as a Theorem 6 to be true. In future, we would pursue this conjecture and try to prove it in totality. Theorem 6 not only holds value in MPI program verification but also in compiler assisted program optimizations.

9.1.2 A Static Analysis Framework for Synergistic Static-Dynamic Analysis

We believe that there is a wide variety of MPI program errors that can discovered at compile time, for instance, erroneous buffer re-use, type mismatches in the send/recv arguments, and even irrelevant barrier detection and some type of deadlocks. Most of these errors can be identified by examining the traditional CFG of the SPMD program treating it no differently than a sequential program CFG. However, for the rest of the errors where matching information among MPI operations is essential, would require a special CFG tailored to SPMD programs. The work in [65, 7] have tried to partially address that problem. We would ideally like to build a static analysis framework, especially borrowing the work from [7] to perform analysis on identifying a set of wildcard receive calls that must be examined dynamically. Such an information can then be fed to a dynamic verification scheduler which can selectively explore interleavings for a supplied input thus pruning a vast schedule space without masking any safety property violations. We can also rely on MPI specific CFGs to deal with programs where the communication flow is conditionally dependent on a particular sender a wildcard receive chooses to match.

9.1.3 Task Permutation vs Match Permutation

Consider the traces of a program shown in Figures 9.1(a) and 9.1(b). Both traces are of the same program. In these figures, P_0 is the master and P_1 and P_2 are the workers. In Figure 9.1(a), P_1 is allotted two tasks and P_2 is allotted one task. With



(a) Match Permutations

(b) Task Permutations of the same example in Figure 9.1(a)

Figure 9.1. Match Permutation vs Task Permutation

such a fixed allotment policy, we would witness that the two sends from P_1 and a single send from P_2 can permute and match any of the three receives. However, the moment task allotment policy changes (such as shown in Figure 9.1(b)), we would witness a whole different class of interleavings. Such instances arise in programs written with dynamic load balancing. The programs with dynamic load balancing are highly symmetric, however, existence of such a symmetry must be first established. The work in [15] provides a solution for discovering symmetry in Message passing programs, however, their solution revolves around approximating the NP-HARD orbit problem [20, 10]. We strongly believe that for the purpose of SPMD programs, we can have simple syntactic checks performed at compile time to discover symmetric components in the communication space of the program. Such checks can very well be on the lines of the work presented in [**?**] (which, however, is only for multi-threaded programs) and can be added as peripheral tools in the MAAPED framework.

9.1.4 Verification for Performance

Verification methodologies can and will be used to increase the performance of MPI application in forthcoming years. We have only scratched the surface with our investigations on identifying FIBs in the MPI code. Barriers are not the only synchronization operations or global fence operations in MPI libraries. It is stated in [3] that:

> *... semantics of the MPI specification purposefully does not mandate whether or not collective communication operations have the side effect of synchronizing the processes over which they operate. Thus, in one valid implementation col-*

lective communication operations may synchronize processes, while in another equally valid implementation they do not.

We would ideally want to construct a dynamic framework which operates not only on the application layer but also at the library layer. Such a framework would identify hot-spots (such as barriers/collective calls) in the application where most time is spent. Furthermore, the framework will provide a summary whether or not such synchronization points are functionally relevant and whether they can be replaced by suitable point-to-point operations.

9.1.5 Hybrid Program Verification

Consistent with the predictions of extreme scale computing report [53], the hybrid programming support has increasingly been witnessed in popular parallel programming libraries. For instance, MPI implementations exploit shared memory mechanisms for data transfer as long as the communicating processes are mapped on the separate cores of a single processor. Programs written with mixed API usage, such as CUDA and MPI have already made their way in the high performance computing world. To the best of our knowledge, inter-API interactions that would exist in applications that rely on mixed usage of APIs have not been formally studied before. The loose semantic characterization of inter-API interactions can be a source of a new class of hard-to-reproduce bugs. For instance, a benign data race caused by an erroneous use of multithreaded API calls in the program may lead to communication deadlock in the MPI specific part of the same program. We believe that a formal study of inter-API interactions in such applications (with the use of hybrid programming models) is essential. The research in to extending predictive dynamic verification methodology for such hybrid programming models would be a valuable contribution.

REFERENCES

[1] `http://javapathfinder.sourceforge.net/`.

[2] `http://www.cs.utah.edu/formal_verification/ISP`.

[3] Reference book entitled "MPI: Complete Reference" available from http://www.netlib.org/utk/papers/mpi-book/mpi-book.ps.

[4] Satisfiability modulo theories. `http://www.smtlib.org/`.

[5] Aiken, A., and Gay, D. Barrier inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1998), POPL '98, ACM, pp. 342–354.

[6] Avrunin, G. S., Siegel, S. F., and Siegel, A. R. Finite-state verification for high performance computing. In *Proceedings of the second international workshop on Software engineering for high performance computing system applications* (New York, NY, USA, 2005), SE-HPCS '05, ACM, pp. 68–72.

[7] Bronevetsky, G. Communication-Sensitive Static Dataflow for Parallel Message Passing Applications. In *CGO: Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization* (2009). ISBN: 978-0-7695-3576-0.

[8] Church, A., and Rosser, J. Some properties of conversion. In *Transactions of the American Mathematical Society, Vol. 39* (1936), American Mathematical Society, pp. 472–482.

[9] Clarke, E. M., Emerson, E. A., Jha, S., and Sistla, A. P. Symmetry reductions in model checking. In *CAV* (1998), pp. 147–158.

[10] Clarke, E. M., Emerson, E. A., Jha, S., and Sistla, A. P. Symmetry reductions in model checking. In *Computer Aided Verification* (Vancouver, BC, Canada, June 1998), A. J. Hu and M. Y. Vardi, Eds., vol. 1427 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 147–158.

[11] Clarke, E. M., Enders, R., Filkorn, T., and Jha, S. Exploiting symmetry in temporal logic model checking. *Form. Methods Syst. Des. 9* (August 1996), 77–104.

[12] Clarke, E. M., Grumberg, O., and Peled, D. *Model Checking*. MIT Press, Dec. 1999.

[13] DeSouza, J., Kuhn, B., de Supinski, B. R., Samofalov, V., Zheltov, S., and Bratanov, S. Automated, scalable debugging of MPI programs with Intel Message Checker. In *Second International Workshop on Software Engineering for High*

Performance Computing System Applications (New York, 2005), ACM, pp. 78–82.

[14] DeSouza, J., Kuhn, B., de Supinski, B. R., Samofalov, V., Zheltov, S., and Bratanov, S. Automated, scalable debugging of MPI programs with Intel® message checker. In *International Workshop on Software Engineering for High Performance Computing Applications (SE-HPCS)* (2005), pp. 78–82.

[15] Donaldson, A. F., Miller, A., and Calder, M. Finding symmetry in models of concurrent systems by static channel diagram analysis. *Electr. Notes Theor. Comput. Sci. 128*, 6 (2005), 161–177.

[16] Elwakil, M., and Yang, Z. Debugging support tool for mcapi applications. In *Proceedings of the 8th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging* (New York, NY, USA, 2010), PADTAD '10, ACM, pp. 20–25.

[17] Elwakil, M., and Yang, Z. Deterministic replay for mcapi programs. In *Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging* (New York, NY, USA, 2011), PADTAD '11, ACM, pp. 6–14.

[18] Elwakil, M., Yang, Z., and Wang, L. Cri: Symbolic debugger for mcapi applications. In *ATVA* (2010), pp. 353–358.

[19] Emerson, E. A., and Sistla, A. P. Symmetry and model checking. *Form. Methods Syst. Des. 9* (August 1996), 105–131.

[20] Emerson, E. A., and Trefler, R. J. From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. To appear.

[21] Exascale Computing Study Report. `http://users.ece.gatech.edu/\~mrichard/ExascaleComputingStudyReports/ECS\_reports.htm`.

[22] FEVS Benchmark. `http://vsl.cis.udel.edu/fevs/index.html`.

[23] FIB Webpage. `http://www.cs.utah.edu/formal\_verification/europvm-mpi08/FIB`.

[24] Fischer, T., Mercer, E., and Rungta, N. Symbolically modeling concurrent mcapi executions. In *PPOPP* (2011), pp. 307–308.

[25] Flanagan, C., and Godefroid, P. Dynamic partial-order reduction for model checking software. In *POPL* (2005), J. Palsberg and M. Abadi, Eds., ACM, pp. 110–121.

[26] Godefroid, P. Using partial orders to improve automatic verification methods. In *Computer-Aided Verification, CAV '* (New Brunswick, NJ, USA, June 1990), R. Alur and T. A. Henzinger, Eds., vol. 1102 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 176–185.

[27] Godefroid, P. *Partial-Order Methods for the Verification of Concurrent Systems: An approach to the State-Explosion Problem.* PhD thesis, Univerite De Liege, 1994–95.

[28] Godefroid, P. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, vol. 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[29] Godefroid, P. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages* (New York, NY, USA, 1997), POPL '97, ACM, pp. 174–186.

[30] Godefroid, P., Hanmer, B., and Jagadeesan, L. Systematic software testing using VeriSoft: An analysis of the 4ess heart-beat monitor. *Bell Labs Technical Journal 3*, 2 (April-June 1998).

[31] Godefroid, P., and Pirottin, D. Refining dependencies improves partial-order verification methods. In *Computer Aided Verification* (Elounda, Greece, June 1993), pp. 438–450.

[32] Godefroid, P., and Wolper, P. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Computer Aided Verification* (Berlin, Germany, July 1992), K. G. Larsen and A. Skou, Eds., vol. 575 of *LNCS*, Springer, pp. 332–342.

[33] Gray, J. Why do computers stop and what can be done about it?, 1985.

[34] Haque, W. Concurrent deadlock detection in parallel programs. *International Journal in Computer Applications 28* (January 2006), 19–25.

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

[36] Ip, C. N., and Dill, D. L. Better verification through symmetry. *Form. Methods Syst. Des. 9* (August 1996), 41–75.

[37] Karypis, G., and Kumar, V. Parallel multilevel k-way partitioning scheme for irregular graphs. In *SuperComputing (SC)* (1996).

[38] Krammer, B., Bidmon, K., Müller, M., and Resch, M. Marmot: An MPI analysis and checking tool. In *Parallel Computing - Software Technology, Algorithms, Architectures and Applications*, F. P. G.R. Joubert, W.E. Nagel and W. Walter, Eds., vol. 13 of *Advances in Parallel Computing*. North-Holland, 2004, pp. 493 – 500.

[39] Luecke, G., Chen, H., Coyle, J., Hoekstra, J., Kraeva, M., and Zou, Y. MPI-CHECK: A tool for checking Fortran 90 MPI programs. *Concurrency and Computation: Practice and Experience 15* (2003), 93–100.

[40] Mattern, F. Virtual time and global states of distributed systems. In *Proceedings Workshop on Parallel and Distributed Algorithms* (North-Holland / Elsevier, 1989), pp. 215–226.

[41] Mazurkiewicz, A. Basic notions of trace theory. In *Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency* (1989), vol. 354, Springer Verlag, Lecture Notes in Computer Science.

[42] The Multicore Communications API and Resource API. http://www.multicore-association.org.

[43] Wiki for MCAPI default usage properties. http://www.cs.utah.edu/formal_verification/mediawiki/index.php/MCAPI.

[44] Message Passing Forum. http://www.mpi-forum.org/docs.

[45] Necula, G. C., Mcpeak, S., Rahul, S. P., and Weimer, W. Cil: Intermediate language and tools for analysis and transformation of c programs. In *In International Conference on Compiler Construction* (2002), pp. 213–228.

[46] Netzer, R. H. B., and Miller, B. P. Optimal tracing and replay for debugging message-passing parallel programs. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing* (Los Alamitos, CA, USA, 1992), Supercomputing '92, IEEE Computer Society Press, pp. 502–511.

[47] Noeth, M., Ratn, P., Mueller, F., Schulz, M., and de Supinski, B. R. ScalaTrace: Scalable compression and replay of communication traces for high-performance computing. *Journal of Parallel and Distributed Computing 69*, 8 (2009), 696–710.

[48] Park, M.-Y., Shim, S., Jun, Y.-K., and Park, H.-R. MPIRace-Check: Detection of message races inMPIprograms. In *Advances in Grid and Pervasive Computing*, vol. 4459 of *Lecture Notes in Computer Science*. 2007, pp. 322–333.

[49] Pervez, S., Palmer, R., Gopalakrishnan, G., Kirby, R. M., Thakur, R., and Gropp, W. Practical model checking methods for verifying correctness of MPI programs. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 14th European PVM/MPI User's Group Meeting* (2007), F. Cappello, T. Hérault, and J. Dongarra, Eds., vol. 4757 of *LNCS*, Springer, pp. 344–353.

[50] Pldt barrier analysis in eclipse ptp tools. http://www.eclipse.org/ptp/documentation/3.0/org.eclipse.ptp.pldt.help/html/barrierxs.html.

[51] Rabenseifner, R. Automatic MPI Counter Profiling. In *n proceedings of the 42nd Cray User Group Conference, CUG SUMMIT* (2005).

[52] Research, M. CHESS: Find and reproduce Heisenbugs in concurrent programs. Accessed 11/7/10.

[53] Sarkar, V., Harrod, W., and Snavely, A. Software challenges in extreme scale systems. *SciDAC Review Special Issue on Advanced Computing: The Roadmap to Exascale* (Jan. 2010), 60–65.

[54] Sharma, S., Gopalakrishnan, G., and Bronevetsky, G. A sound reduction of persistent-sets for deadlock detection in mpi application. *under submission* (2012).

[55] Sharma, S., Gopalakrishnan, G., and Mercer, E. Dynamic verification of multicore communication applications in mcapi. In *HLDVT* (2009), pp. 100–105.

[56] Sharma, S., Gopalakrishnan, G., Mercer, E., and Holt, J. MCC - A runtime

verification tool for MCAPI user applications. In *9th International Conference Formal Methods in Computer Aided Design (FMCAD)* (Nov. 2009), IEEE, pp. 41–44.

[57] Sharma, S., Vakkalanka, S., Gopalakrishnan, G., Kirby, R. M., Thakur, R., and Gropp, W. A formal approach to detect functionally irrelevant barriers in MPI programs. pp. 265–273.

[58] Sharma, S. V., Gopalakrishnan, G., and Kirby, R. M. A survey of MPI related debuggers and tools. Tech. Rep. UUCS-07-015, University of Utah, School of Computing, 2007. `http://www.cs.utah.edu/research/techreports.shtml`.

[59] Siegel, S. F. The MADRE web page. `http://vsl.cis.udel.edu/madre`, 2008.

[60] Siegel, S. F. MPI-Spin web page. `http://vsl.cis.udel.edu/mpi-spin`, 2008.

[61] Siegel, S. F., and Avrunin, G. S. Verification of MPI-based software for scientific computation. In *International SPIN Workshop on Model Checking Software (SPIN)* (2004), pp. 286–303.

[62] Siegel, S. F., and 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)* (2005), ACM Press, pp. 95–106.

[63] Siegel, S. F., Mironova, A., Avrunin, G. S., and Clarke, L. A. Combining symbolic execution with model checking to verify parallel numerical programs. *ACM Transactions on Software Engineering and Methodology 17*, 2 (2008), Article 10, 1–34.

[64] Smaragdakis, Y., Evans, J., Sadowski, C., Yi, J., and Flanagan, C. Sound predictive race detection in polynomial time. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2012), POPL '12, ACM, pp. 387–400.

[65] Strout, M. M., Kreaseck, B., and Hovland, P. D. Data-flow analysis for mpi programs. In *Proceedings of the 2006 International Conference on Parallel Processing* (Washington, DC, USA, 2006), ICPP '06, IEEE Computer Society, pp. 175–184.

[66] TotalView concurrency tool.

[67] Vakkalanka, S. *Efficient Dynamic Verification Algorithms for MPI Applications.* PhD thesis, University of Utah, 2010.

[68] Vakkalanka, S., DeLisi, M., Gopalakrishnan, G., Kirby, R. M., Thakur, R., and Gropp, W. Implementing efficient dynamic formal verification methods for MPI programs. In *EuroPVM/MPI* (2008).

[69] Vakkalanka, S., Gopalakrishnan, G., and Kirby, R. M. Dynamic verification of mpi programs with reductions in presence of split operations and relaxed orderings. In *Proceedings of the 20th International Conference on Computer Aided Verification* (Berlin, Heidelberg, 2008), CAV '08, Springer-Verlag, pp. 66–79.

[70] Vakkalanka, S., Sharma, S. V., Gopalakrishnan, G., and Kirby, R. M. Isp:

A tool for model checking MPI programs. In *ACM Conference on Principles and Practices of Parallel Programming (PPoPP)* (2008), pp. 285–286.

[71] Vakkalanka, S., Szubzda, G., Vo, A., Gopalakrishnan, G., Kirby, R. M., and Thakur, R. Static-analysis assisted dynamic verification of MPI waitany programs (poster abstract). In *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (2009), vol. 5759 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 329–330.

[72] Vakkalanka, S., Vo, A., Gopalakrishnan, G., and Kirby, R. M. Reduced execution semantics of MPI: From theory to practice. In *FM 2009* (Nov. 2009), pp. 724–740.

[73] Vakkalanka, S., Vo, A., Gopalakrishnan, G., and Kirby, R. M. Precise dynamic analysis for slack elasticity: Adding buffering without adding bugs. In *Proceedings of the 17th European MPI Users' Group Meeting Conference on Recent advances in the Message Passing Interface* (Berlin, Heidelberg, 2010), EuroMPI'10, Springer-Verlag, pp. 152–159.

[74] Vetter, J. S., and de Supinski, B. R. Dynamic software testing of MPI applications with Umpire. In *SC* (2000).

[75] Vo, A. *Scalable Formal Dynamic Verification of MPI Programs through Distributed Causality Tracking*. PhD thesis, University of Utah, 2011.

[76] Vo, A., Aananthakrishnan, S., Gopalakrishnan, G., de Supinski, B. R., Schulz, M., and Bronevetsky, G. A scalable and distributed dynamic formal verifier for MPI programs. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC10)* (2010), ACM.

[77] Vo, A., Aananthakrishnan, S., Gopalakrishnan, G., de Supinski, B. R. d., Schulz, M., and Bronevetsky, G. A scalable and distributed dynamic formal verifier for MPI programs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (Washington, DC, USA, 2010), SC '10, IEEE Computer Society, pp. 1–10.

[78] Vo, A., Vakkalanka, S., DeLisi, M., Gopalakrishnan, G., Kirby, R. M., and Thakur, R. Formal verification of practical MPI programs. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2009), PPoPP '09, ACM, pp. 261–270.

[79] Vo, A., Vakkalanka, S. S., Williams, J., Gopalakrishnan, G., Kirby, R. M., and Thakur, R. Sound and efficient dynamic verification of MPI programs with probe non-determinism. In *EuroPVM/MPI* (2009), pp. 271–281.

[80] Vuduc, R., Schulz, M., Quinlan, D., de Supinski, B., and 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* (2006), ACM, pp. 27–36.

[81] Vuduc, R., Schulz, M., Quinlan, D., de Supinski, B., and Sornsen, A.

Improving distributed memory applications testing by message perturbation. In *Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging* (New York, NY, USA, 2006), PADTAD '06, ACM, pp. 27–36.

[82] Xue, R., Liu, X., Wu, M., Guo, Z., Chen, W., Zheng, W., Zhang, Z., and Voelker, G. MPIWiz: Subgroup reproducible replay of MPI applications. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2009), PPoPP '09, ACM, pp. 251–260.

[] Yang, Y., Chen, X., Gopalakrishnan, G., and Kirby, R. M. Distributed dynamic partial order reduction based verification of threaded software. In *SPIN* (2007), pp. 58–75.

[] Yang, Y., Chen, X., Gopalakrishnan, G., and Kirby, R. M. Efficient stateful dynamic partial order reduction. In *SPIN* (2008), Lecture Notes in Computer Science, Springer. Model Checking Software, 15th International SPIN Workshop, Los Angeles, CA.

[] Yang, Y., Chen, X., Gopalakrishnan, G., and Wang, C. Automatic discovery of transition symmetry in multithreaded programs using dynamic analysis. In *Proceedings of the 16th International SPIN Workshop on Model Checking Software* (Berlin, Heidelberg, 2009), Springer-Verlag, pp. 279–295.

[] `http://yices.csl.sri.com/`.

[] Zhang, Y., and 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* (New York, NY, USA, 2007), PPoPP '07, ACM, pp. 194–204.