Formal Verification of Practical MPI Programs

Anh Vo   Sarvani Vakkalanka
Michael DeLisi   Ganesh Gopalakrishnan
Robert M. Kirby [*]

School of Computing, University of Utah
{avo,sarvani,delisi,ganesh,kirby}@cs.utah.edu

Rajeev Thakur [†]

Mathematics and Computer Science Division
Argonne National Laboratory
thakur@mcs.anl.gov

Abstract

This paper considers the problem of *formal verification* of MPI programs operating under a fixed test harness for safety properties *without* building verification models. In our approach, we directly model-check the MPI/C source code, executing its interleavings with the help of a verification scheduler. Unfortunately, the total feasible number of interleavings is exponential, and impractical to examine even for our modest goals. Our earlier publications formalized and implemented a partial order reduction approach that avoided exploring equivalent interleavings, and presented a verification tool called ISP. This paper presents algorithmic and engineering innovations to ISP, including the use of OpenMP parallelization, that now enables it to handle practical MPI programs, including: (i) ParMETIS - a widely used hypergraph partitioner, and (ii) MADRE - a Memory Aware Data Re-distribution Engine, both developed outside our group. Over these benchmarks, ISP has automatically verified up to 14K lines of MPI/C code, producing error traces of deadlocks and assertion violations within seconds.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program VerificationModel checking

General Terms Verification

Keywords MPI, Message Passing Interface, distributed programming, model checking, dynamic partial order reduction [1]

1. Introduction

The Message Passing Interface (MPI) [7] library remains one of the most widely used APIs for implementing distributed message passing programs. Its projected usage in critical, future applications such as Petascale computing [4] makes it imperative that MPI programs be free of programming logic bugs. While techniques such as model checking [2] have made great advances in verifying concurrent system *models*, they are yet to make a discernible dent on

code level verification, especially for MPI programs. The goal of code level (model checking based) verification is to verify a concurrent program by replaying the interleavings of the actual program *without* building verification models. Code level verification is essential because the act of building and maintaining verification models from real programs – whether 'by hand' or through the use of automated tools – is impractical. Verification model building has to be repeated each time the code is tuned. MPI programs, consisting of layers of user library and MPI library function invocations, are written in languages such as C, C++, and Fortran whose semantics are difficult to mirror in verification models. These semantics are further affected by compilers and runtimes. Thus, the runnable object code is often its own best verification model.

In all our discussions of code level verification, we are assuming that the environment of the code is provided by *one* test harness. The test harness defines the initial state of the program together with subsequent input (if any) needed by the program during execution. Verification that covers all test harnesses is future work.[2] We keep our verification goal modest, not aiming for general temporal logic verification, but only for deadlock and local assertion violation detection. Even with these assumptions, the number of process/thread interleavings grows exponentially, requiring efficient verification techniques. In practice, however, many of the thread interleavings are equivalent, consisting merely of commutations of independent actions.

One of the earliest tools exploiting independence between thread actions and performing code level model checking for reactive C programs was Verisoft [6]. Verisoft employed a scheduler that replayed the program interleavings following a *stateless* model checking approach. In the stateless model checking approach, it is assumed that the program terminates – *i.e.*, the product state space of the thread executions is acyclic. (If this assumption is not met, one can still achieve a reasonable degree of debugging by fixing a depth bound, and conducting verification up to this bound.) In a stateless model checker, each reachable state of the program is discovered by executing (the code of) thread transitions.

Suppose a global execution state s is reached, and two thread actions t_1 and t_2 are enabled at s. Depending on the particular implementation method chosen, a stateless model checker may do one of two things: (i) fork the execution into one that considers t_1 first followed by t_2, and another that considers t_2 first, (ii) employ an external scheduler that forces an arbitrary interleaving (say, it picks t_1), while noting down alternate interleavings possible at each point (t_2 in our example), and later replaying the execution from the beginning where it now forces the alternative execution (t_2) when s is re-encountered.

[2] Although, for many MPI programs, we have observed that the kind of bugs we are interested in detecting would be revealed by a fixed test harness.

As noted earlier, without exploiting independence between thread actions, this strategy results in an exponential number of interleavings for p processes executing n steps each.[3] Verisoft's interleaving reduction approach was based on classifying thread actions statically into *global* and *local*. Based on this classification, it employed a method for interleaving reduction called *persistent set based partial order reduction* [5]. Given an execution state s and a set of thread actions $T = \{t_1 \ldots t_k\}$ ($k \leq p$) enabled at s, a persistent set $T_p \subseteq T$ is one that guarantees that any sequence σ of process moves that stays *outside* T_p leaves T_p enabled along the execution. For example, if t_1 is a *local variable* update of thread τ_1 and the remaining members of T are arbitrary, then one can choose $T_p = \{t_1\}$. As another example, (i) if t_1 and t_2 of threads τ_1 and τ_2 are statements of the form if $pred$ then $action$ involving global variables in $pred$ and $action$, and (ii) the firing of t_1 disables t_2 and vice versa, then one must include both t_1 and t_2 within T_p. Now the idea behind partial order reduction (POR) is this: at state s, recursively explore (say, through forking) *only* the actions within T_p. In other words, while at state s, do not pursue all actions within T. Clearly, if we can ensure that $\mid T_p \mid \lll \mid T \mid$, then a vast reduction in the number of states explored is possible. Such reduction guarantees *soundness* in that whatever bugs were detectable by the full search would still be detected during reduced search. In essence, for every sequence explored in the full search, a *representative* sequence is explored by the reduced search.

The size of the persistent set T_p explored at each state s depends on the degree of dependence between thread actions at state s. For instance, if t_1 is `a[x]++` and t_2 is `a[x]--` for global variable `x`, then both t_1 and t_2 must be in T_p. This is because a third thread τ_3 may trigger a bug depending on whether t_1 fires first or t_2. However, dependences may not be statically apparent. As an example, we may have actions `a[x]++` and `a[y]--` instead of `a[x]++` and `a[x]--`, and it may be case that `x` and `y` may alias depending on the outcome of a complex conditional statement. Since it is in general very difficult (formally undecidable) to determine such aliases during static analysis, algorithms such as [6] errored on the side of caution and assumed that `x` and `y` could be aliases (i.e., t_1 and t_2 depended on each other) in case they could not prove through static analysis that they *did not*. This approach is the so called *static* POR approach. While conservative, static POR is pessimistic, often giving no reductions.

These ideas can also be applied to message passing (MPI) programs. For example, in MPI programs, operation `MPI_Send(target1,communicator1,data1)` issued by process 1, and operation `MPI_Send(target2,communicator2,data2)` issued by process 2 become dependent with respect to [4] operation `MPI_Recv(ANY,communicator3,var)` issued by process 3 if `target1` and `target2` evaluate to 3, and all three `communicatori` (for `i` between 1 and 3) evaluate to the same value. If these evaluation outcomes cannot be determined statically, then a POR algorithm will conservatively attempt to match the sends with the receives in both orders – and this may be highly wasteful if indeed the evaluations do not meet the conditions described above. (Note: In this paper, due to space limitations, we offer only this brief definition of dependence - see [2] for details: two actions a and b are dependent iff whenever they are both enabled at state s, (i) the firing of one does not disable the other, and (ii) their firing sequentially, i.e., $a;b$ or $b;a$ results in the same state.)

To alleviate the drawback with static POR, Flanagan and Godefroid [3] introduced the idea of *dynamic* POR (DPOR). We explain this idea assuming that the interleaving replays are forced by a scheduler. Consider a scheduler that has produced a trace σ and now is attempting step t_j (say action `a[y]--` of some thread j). The scheduler examines σ to see whether it contains a state $s1$ from where an action t_i (say `a[x]++` of another thread i) was executed such that t_i and t_j are dependent. Unlike during static POR, the scheduler, now knowing the concrete values of `x` and `y`, can conclude precisely whether $x == y$, and determine the possibility of dependence accurately. If determined to be dependent, the scheduler adds an auxiliary piece of information (called the *backtrack set* in [3]) to $s1$ indicating that when the execution is replayed and $s1$ is reached, thread t_j must be scheduled. This idea, carried till no more traces are generated, results in every state having expanded a tight superset (in practice better than achievable through static POR) of its most optimal persistent set.

While these ideas are promising, and have been successfully implemented in the context of PThread/C programs [26, 27] and Java programs [15], the development of a dynamic partial order reduction algorithm along the lines proposed in [3] **proved impossible for MPI programs**. Briefly (Section 2 has the details), the reasons are as follows. First, in MPI, the runtime decides which sends match which receives. Coming to our earlier example, merely firing operations `MPI_Send(target1,comm1,data1)` from process 1 followed by firing `MPI_Send(target2,comm2,data2)` from process 2 (and firing operation `MPI_Recv(ANY,comm3,var)` from process 3 at any time) does not guarantee that process 1's send will match the receive. Depending on the cluster as well as the MPI runtime, one or the other of these operations may preferentially match the receive. To wrest control away from these elements and back to our scheduler, what we do is to (i) determine the *maximal* set of sends that can match a wildcard receive, and (ii) once this maximal set has been determined (say P), *rewrite* `MPI_Recv(ANY,communicator3,var)`, in turn, to `MPI_Recv(i,communicator3,var)` for every `i` within P, and explore all these interleavings (in MPI, such a 'specific receive' is guaranteed to match the next unmatched send issued by `i`). Second, MPI barriers have an unusual semantics in that sometimes a 'send' issued after a barrier can match a receive issued before the barrier. After solving these and many other aspects pertaining to MPI, we finally obtained our algorithm called POE (Partial Order avoiding Elusive interleavings) implemented in our tool ISP (In-Situ Partial order), with its features reported in [22–24].

1.1 Our Contributions

Our main contributions in this paper are the algorithmic improvements that now allow ISP to handle two practical MPI programs that serve as our case studies. The first case study, ParMETIS [8], is a widely used hypergraph partitioner. We can verify the V3ParkKWay function of ParMETIS (over 14,000 lines of MPI/C code) for 16 different input drivers (tests). For each of these tests, the *total* number of interleavings of the constituent MPI processes would be impossible to examine. However, employing the notion of independence, the ISP algorithm discovered that *just one* relevant interleaving exists per test, and these can be run in just under 12 minutes on a high-end workstation. The algorithmic improvements required for the success of handling this case study are: (i) improvements to the data structures used by the POE algorithm, and (ii) loop parallelism provided by OpenMP employed within the execution engine of POE.

Our second case study, MADRE, is a recently described Memory Aware Data Re-distribution Engine [21]. The reason why MADRE serves as an interesting data point is as follows. Many MPI programs are written without the use of any non-deterministic

---

[3] The number of possible interleavings is $(np)!/((n!)^p)$; but since the thread actions can have global effects, the number of distinct executions is even higher.

[4] More precisely, one should regard the send/receive match as one "operation" and consider dependences among these match possibilities.

construct such as a wildcard receive, `MPI_Waitany`, etc. For all such MPI programs, ISP guarantees to generate *exactly one interleaving*. However, the converse is not true: even for some MPI programs that use non-deterministic MPI operations, it is possible that ISP will generate only one interleaving (for instance, if this non-deterministic possibility never materializes at runtime). However, for a program such as MADRE, it was felt that indeed the non-deterministic possibility will materialize, and we wanted to discover whether the improved POE algorithm can handle it. In summary, the number of interleavings for the UnitBred test of MADRE is actually quite high: $n!$ interleavings for n processes. We discovered that

- ISP could comfortably handle this many interleavings
- ISP automatically detected a deadlock within the UnitBred test. This was later discovered (by us) to be a well-documented deadlock. However the fact that we did not know of a deadlock possibility lurking within MADRE, and that we revealed it in less than one second by performing push-button model checking is a highly encouraging result.

ISP verified many schemes/tests in MADRE (cycleShiftBred, parkBred, etc.) in one interleaving and found no deadlock.

Our own past work on ISP is as follows: [24] a mathematical description of POE, [23] provides details on handling collectives, [22] documents how we dealt with MPI's progress engine, and [16] presents a redundant MPI barrier removal algorithm. This paper for the first time answers in the affirmative that ISP can apply to practical MPI programs, and describes the developments leading to this result.

Roadmap: The remainder of the paper is organized as follows. Section 2 discusses some related work. In Section 3, we provide an overview of ISP and the POE Algorithm. Section 4 documents our verification effort for MADRE followed by a brief over view of ParMETIS and its use of MPI in Section 5. Section 6 deals with the challenges arising when verifying ParMETIS with ISP. We describe step by step improvements made to ISP to better handle large applications and provide the experimental results of this study in Section 7. In Section 8, we conclude the report with the key lessons that were learned along the experience, as well as provide several ideas for future research.

2. Related Work

Over the past several years, considerable effort has been put into building verification tools for MPI programs (e.g., [10,13,25], with a survey of such tools provided in [17]). However, these tools are still unable to reliably verify MPI programs. For example, tools such as Marmot [10] attempt to perturb schedules by inserting padding delays, but end up missing even extremely simple deadlock scenarios, as shown in [24]. While MPI-SPIN [19,20], a model checker based on SPIN, can detect the kind of errors that ISP does and also exhaustively explores all the interleavings of the program, this approach depends on model building effort on part of the user which we consider impractical. MPI-SPIN does provide a reduction algorithm called the *Urgent Algorithm* that allows all MPI send/receive channels to be treated as rendezvous channels. However, this algorithm applies only to programs that do not use wildcard receives. In general, MPI-SPIN relies on SPIN's POR algorithm which, unfortunately, does not "understand" the commuting properties of MPI calls. In its favor, MPI-SPIN supports a symbolic execution facility to compare a sequential algorithm against an MPI implementation of the very same algorithm to detect numerical inaccuracies - a feature not supported by ISP.

A dynamic debugging tool for multithreaded programs called CHESS [14] was recently proposed. Currently, CHESS employs a context-bounded search approach, as opposed to DPOR. In addition, CHESS does not run into issues similar to those discussed in Section 1 concerning MPI's wildcard receives and barriers. Nevertheless, the success that CHESS has had in debugging several real-world programs is another indication of the pragmatic advantages of dynamic code-level model checking over model-based model checking.

3. ISP Overview

ISP works by intercepting the MPI calls made by the target program and making decisions on when to send these MPI calls to the MPI library. This is accomplished by the two main components of ISP: the Profiler and the Scheduler. An overview of ISP's components and their interaction with the program as well as the MPI library is provided in Figure 1.

The Profiler: The interception of MPI calls is accomplished by compiling the ISP profiler together with the target program's source code. The profiler makes use of MPI's profiling mechanism (PMPI). It provides its own version of *MPI_f* for each corresponding MPI function f. Within each of these MPI_f, the profiler communicates with the scheduler using TCP sockets to send information about the MPI call the process wants to make. It will then wait for the scheduler to make a decision on whether to send the MPI call to the MPI library or to postpone it until later. When the permission to fire f is given from the scheduler, the corresponding *PMPI_f* will be issued to the MPI run-time. Since all MPI libraries come with functions such as *PMPI_f* for every MPI function f, this approach provides a portable and light-weight instrumentation mechanism for MPI programs being verified.

The ISP Scheduler: The ISP scheduler helps carry out the POE algorithm. The scheduler has to meet the objectives indicated in Section 1, namely: G1: the maximal set of sends that can match a wildcard receive must be determined, and G2: the unusual MPI barrier semantics must be respected. To help explain these issues, we first discuss a few examples in Section 3.1. We then begin explaining the POE algorithm in Section 3.2. As it turns out, the POE algorithm is based on exploiting MPI's out-of-order completion semantics. In other words, (i) not all MPI operations issued by a process complete in that order, and (ii) a proper modeling of this out-of-order completion semantics is essential in order to meet goals G1 and G2. For example, two `MPI_Isend` commands issued in succession by an MPI process P1 to the same target process (say P2) are forced to match in order, whereas if these `MPI_Isend`s are targeted to two *different* MPI processes, then they *may match contrary to the issue order*. As another example, any operation following an `MPI_Barrier` must complete only after the barrier has completed, while an operation issued *before* the barrier may linger across the barrier, and actually complete *after* the barrier! These ideas will be brought out in the examples in Section 3.1.

These discussions will lead us to observe that the high level state maintained by the ISP scheduler must include (i) the current process, (ii) the next "PC" of each process, (iii) whether a process is at a *fence point*, (iv) the intra-completes graph of all actions encountered in each process, and (v) whether an action has been encountered but not issued into the MPI runtime. These ideas will be discussed in Sections 3.3 and 3.4. The rest of the paper will then discuss how the implementation of the POE algorithm was optimized to help ISP handle practical MPI programs.

3.1 Examples Illustrating ISP

3.1.1 Wildcard Receives

In the short MPI program example in Figure 2, processes P0 and P2 are targeting P1 which entertains a 'wildcard match,' i.e., can receive from *any* process that has a concurrently enabled MPI_Isend
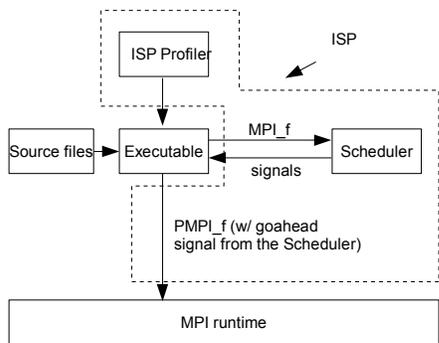
Figure 1. Overview of ISP

```
P0: MPI_Isend(to P1, data = 42); ...
P1: MPI_Irecv(*, x); if (x==42) then error1 else ...
P2: MPI_Isend(to P1, data = 21); ...
```

Figure 2. Simple MPI Example Illustrating Wildcard Receives

```
P0: S0(to P1, h0) ; B0;              W(h0);
P1: R(*, h1)      ; B1;              W(h1);
P2:                 B2; S2(to P1, h2);W(h2);
```

Figure 3. Illustration of Barrier Semantics and the POE Algorithm

targeting P1. As soon as one such send is chosen (for example, P0's), the other send is disabled. It is not eligible to match with this receive of P1 (it has to match another receive of P1 coming later). As pointed out in Section 1, this 'disabling' behavior characterizes dependence. This can be vividly seen from the fact that the particular send that matches may or may not cause error1 to be triggered. Consider given $i < j < k$, and a trace t where the ith action of t, namely t_i, is P2's send, and similarly t_j is P1's receive, and t_k is P0's send. In this trace, it is not necessary that P1's receive is matched with P2's send just because t_i is executed before t_k. MPI implements its own buffering mechanism that can cause one send to race ahead of the other send. Formally, unlike in other works (e.g., [3]), the program order of MPI calls does not imply *happens-before* [11] of the MPI call executions. Hence, it is possible that t_j is matched with t_k. There is no way in an MPI run-time (short of making intrusive modifications to the MPI library, which is often impossible because of the proprietary nature of the libraries) to force a match either way (both sends matching the receive in turn) by just changing the order of executing sends from P2 and P0. As explained earlier, we solve this problem by performing dynamic rewriting of wildcard receive commands.

3.1.2 MPI Barriers

According to the MPI library semantics, no MPI process can issue an instruction past its barrier unless all other processes have issued their barrier calls. Therefore, an MPI program must be designed in such a way that when an MPI process reaches a barrier call, all other MPI processes also reach their barrier calls (in the MPI parlance, these are *collective operations*); a failure to do so deadlocks the execution. While these rules match the rules followed by other languages and libraries in supporting their barrier operations, in the case of MPI, it is possible for a process P_i to have an operation OP

before its barrier call, and another process P_j to have an operation OP' after P_j's matching barrier call where OP can observe the execution of OP'. This means that OP can, in effect, complete after P_i's barrier has been invoked. This shows that the program ordering from an operation to a following barrier operation need not be obeyed during execution. To ensure higher performance, this behavior is allowed in MPI, as shown by the example in Figure 3, and requires special considerations in the design of POE. In this example, one MPI_Isend issued by P0, shown as S0(to P1, h0), and another issued by P2, shown as S2(to P1, h2), target a wildcard Irecv issued by P1, shown as R(*,h1).[5] The following execution is possible: (i) S0(to P1, h0) is issued, (ii) R(*, h1) is issued, (iii) each process *fully* executes its own barrier, (B0, B1, or B2 are abbreviations for MPI_Barrier), and this "collective operation" finishes (all the B's indeed form an atomic set of events), (iv) S2(to P1, h2) is issued, (v) now both sends and the receive are alive, and hence S0 and S2 become dependent, requiring a dynamic algorithm to pursue both matches. Notice that S0 can finish after B0 and R can finish after B1.

3.2 POE Algorithm Overview

The POE algorithm works as follows. There are two classes of statements to be executed: (i) those statements of the embedding programming language (C/C++/Fortran) that do not invoke MPI commands, (ii) the MPI function calls. The embedding statements in an MPI process are local in the sense that they have no interactions with those of another process. Hence, under POE, they are executed in program order. When an MPI call f is encountered, the scheduler records it in its state; however, it does not (necessarily) issue this call into the MPI run-time. (Note: When we say that the scheduler issues/executes MPI call f, we mean that the scheduler grants permission to the process to issue the corresponding PMPI_f call to the MPI run-time). This process continues until the scheduler arrives at a *fence*, where a fence is defined as an MPI operation that cannot complete after any other MPI operation following it. The list of such fences include MPI_Wait, MPI_Barrier, etc., and are formally defined in [24]. When all MPI processes are at their own fences, the full extent of all senders that can match a wildcard receive becomes known, and dynamic rewriting can be performed with respect to these senders. In the example in Figure 2, Irecv(*) is written into Irecv(from P0) and Irecv(from P2).

The POE algorithm now forms *match-sets*. Each match-set is either a single big-step move (as in operational semantics) or a set of big-step moves. A *set* of big-step moves results from dynamically rewriting a wildcard receive. Each big-step move is a set of actions that are issued collectively into the MPI run-time by the POE-scheduler (we enclose them in $\langle\langle .. \rangle\rangle$). In the example of Figure 3, the match-sets are:

● The set of big-step moves
{ $\langle\langle$ S0(to P1), R(from P0) $\rangle\rangle$,
$\langle\langle$ S2(to P1), R(from P2) $\rangle\rangle$ }
● The single big-step move
$\langle\langle$ B0,B1,B2 $\rangle\rangle$

The POE algorithm executes all the big-step moves (match sets). The execution of a match-set consists of executing all its constituent MPI operations (firing the PMPI versions of these operations into the MPI runtime). The *set of big-step moves* (set of match sets) is executed only when no ordinary big-step moves are left. In our example, the big-step move of barriers is executed first. This priority order guarantees that a representative sequence exists for each possible interleaving (see [24] for details).

---

[5] While not central to our current example, we also take the opportunity to illustrate how the handles h0 through h2, and MPI_Wait(W) are used.

Once only a set of big-step moves are left, each member of this set (a big-step move) is fired. The POE algorithm then resumes from the resulting state.

In our example, each big-step moves in the set
{ ⟨⟨ S0(to P1), R(from P0) ⟩⟩,
⟨⟨ S2(to P1), R(from P2) ⟩⟩ }
is executed, and the POE algorithm invoked after each such big-step move.

Thus, one can notice that the POE scheduler never actually issues into the MPI run-time any wildcard receive operations it encounters. It always dynamically rewrites these operations into receives with specific sources, and pursues each specific receive paired with the corresponding matching send as a match-set in a depth-first manner.

The following sections discuss the complete-before orderings of MPI and match-set formations, which are the key areas that were improved in ISP to handle ParMETIS.

3.3 Completes-Before Ordering

The Completes-Before (CB) ordering accurately captures when two MPI operations x and y issued from the same process in program order are guaranteed to complete in that order. For example, if an MPI process P1 issues an `MPI_Isend` that ships a large message to P2 and then issues `MPI_Isend` that ships a small message to P3, it is possible for the second `MPI_Isend` to finish first. A summary of the completes-before order of MPI is as follows: (i) **Send Order**: Two `Isend`s sending data to the same destination complete in issue order. (ii) **Receive Order**: Two `Irecv`s receiving data from the same source complete in issue order. (iii) **Wildcard Receive Order**: If a wildcard `Irecv` is followed by another `Irecv` (wildcard or not), the issue order is respected by the completion order. (iv) **Wait Order**: A `Wait` and another MPI operation following it complete in issue order. For a formal description of the CB relation, please see [24].

The POE algorithm represents this ordering between two MPI operations x and y (in which x completes before y) by storing an `IntraCB` edge from x to y.

3.4 Match Sets

First we define *match sets* and *ancestors*. An MPI operation x is defined as a *fence* if and only if x completes before all other MPI operations that appeared after x in program order. For two MPI operations x, y: x is the *ancestor* of y if and only if x completes before y.

Match sets play a vital role in the POE algorithm. An MPI operation instruction is considered *matched* when it has been given permission by the scheduler to be fired into the MPI runtime. For example, a collective operation is matched when POE has collected all the corresponding collective calls from all the processes that are supposed to participate in the call. Semi-formally defined, a match set is a set of big-step moves as described in Section 3.2. For example, a match set of type `Irecv` will contain exactly one `Isend` and its matching non-wildcard `Irecv`. However, the tricky part is to form the match set of type `Irecv(*)`, which represents a wildcard receive and all of its possible matching sends. POE does this by starting with a set containing just the wildcard receive in question. It then seeks the maximal number of additional sends that can be added to this set, without hitting a fence. Finally the wildcard `*` is rewritten into specific instances of process IDs. In addition, in order for an operation MPI_f to be a part of any match-set, all of f's ancestors must have been issued to the MPI run-time (in order words, all of f's ancestors must have been matched earlier). The exact formation of match-sets is thus governed by two aspects: (i) which operations 'go together' (e.g., a receive statement and a send that it is sourcing from), and (ii) whether all the ancestors

of a match-set have been issued (to preserve the completes-before semantics).

We now present our case studies and explain how POE was optimized in order to model check them.

4. MADRE

MADRE is a collection of memory aware parallel redistribution algorithms addressing the problem of efficiently moving data blocks across processes without exceeding the allotted memory of each process. MADRE is an interesting target for ISP because it belongs to a class of MPI programs that makes use of wildcard `Irecv`s,[6] which potentially could result in deadlocks that can easily go undetected due to programming timings (and they sometimes do; in Section 7.3, we explain one such scenario).

Other than the two algorithms cycleShiftBred and parkBred mentioned in [21], MADRE also includes several variations of other data redistribution algorithms, including unitBred, phBred, and phaseBred. For more details concerning these implementations, please consult the MADRE manual [18]. Except for unitBred, which we will go into further details later, ISP successfully verified the whole library to be free of deadlocks under the provided test inputs that come with MADRE distribution 0.3. ISP also finished the verification process of each of the aforementioned tests within seconds, and none of them required more than one interleaving.

4.1 unitBred

unitBred is a very simple implementation for moving data across processes. However, it is an interesting test target because of its use of `MPI_ANY_SOURCE` and `MPI_ANY_TAG`, which requires multiple interleavings to completely test all the possible message matchings. In fact, for n processes, ISP needs to explore n! interleavings to verify unitBred completely (This is because of ISP's overapproximation of potential senders that can match wildcard receives. This issue is addressed again in Section 8). ISP was also able to detect the following unsafe MPI usage pattern within unitBred:

```
P0: Isend; Wait(); Irecv; Wait();
P1: Isend; Wait(); Irecv; Wait();
```

Under typical testing conditions where the message sizes are not large enough, the above pattern usually does not cause any problem due to the fact that `Isend`s are usually buffered by most MPI libraries (MPICH2, for example, buffers messages up to 256K). However, if the MPI implementation does not allow the sends to buffer and forces the sends to synchronize, the above section of code will deadlock. This behavior happens when the size of the messages are larger than the buffer or in the case when the implementation does not buffer sends at all. Because ISP tries to form the match-sets when both processes encounter the fence instructions (`Wait` in this case) and the `Irecv`s are not yet issued by the processes, this unsafe pattern is easily detected. Although this bug was already known to the MADRE authors, the fact that ISP could discover it without knowing its existence in advance is a very encouraging result.

5. ParMETIS

In our verification of ParMETIS, we had to understand the ParMETIS code only at a superficial level. This supports our point about push-button verification: that one does not need to spend a huge amount

---

[6] Wildcard receives can be either through the use of `MPI_ANY_SOURCE`, or `MPI_ANY_TAG`, or any combination of the two. ISP is capable of handling both of them.

of time learning the inner working of the target program to be able to verify it using ISP.

ParMETIS is a parallel library that provides implementations of several effective graph partitioning algorithms. Besides being a parallel extension of METIS [8], ParMETIS also provides several parallel routines that are especially suitable for graph partitioning in parallel computing environment. Graph partitioning is the problem of partitioning a given graph into k roughly equal subsets, such that the total cost of edges in each subset is minimized. While graph partitioning is an NP-complete problem, most graph partitioning routines of ParMETIS belong to the multilevel graph partition algorithms class, which offer excellent partitioning at an acceptable level of computational complexity. At a high level of description, multilevel graph partitioning works by collapsing vertices and edges of the original graph into a much smaller graph, a process called coarsening. The algorithm then attempts to partition this coarsened graph, which is a much easier task. Finally, this partitioning is gradually refined through some refinement heuristics as it is projected into the bigger graphs through the uncoarsening phase.

Parellization of a multilevel graph partition algorithm is a difficult task. The coarsening/uncoarsening phase requires that nodes and edges be merged/unmerged. Since vertices and edges are distributed randomly across multiple processes, this requires a lot of communications. Most refinement heuristics are also hard to parallelize [9]. Despite these difficulties, ParMETIS manages to implement some very efficient parallel routines for graph partitioning. More details on ParMETIS and its performance can be obtained in [8].

6. Model Checking ParMETIS

We attempted to apply ISP on ParMETIS to verify the routines for freedom of deadlocks as well as resource leaks. After several improvements and optimizations (detailed below), ISP was able to verify all routines of ParMETIS without the need of any manual modeling effort. ParMETIS was verified to be free of deadlocks and resource leaks in just one single interleaving. We also tried to inject several bugs into ParMETIS (see Section 7.3, and ISP was also able to discover all these errors.

6.1 The Challenges

Verifying ParMETIS is a challenging task, not only because of its scale (AdaptiveRepart, one re-partition routine provided by ParMETIS, has more than 12,000 lines of code between itself and its helper functions), but also because of the enormous number of MPI calls involved. In some of our tests, the number of MPI calls recorded by our POE scheduler exceeded 1.3 million. This class of applications stresses both the memory usage overhead and the processing overhead of the scheduler.

Our attempt to improve ISP while working on the large code base of ParMETIS introduced several challenges at a pragmatic level. Since we did not have another MPI program debugger – and especially one that understands the semantics of our ISP scheduler that was itself being tweaked – we had to spend considerable effort employing low level debugging methods based on `printf`s and similar methods.

6.2 Memory Consumption

In order to replay the execution of the processes and correctly *skipped over* all the previous matching of sends/receives, ISP has to store all the transitions (i.e., the MPI calls) for each processes. This consumes a considerable amount of memory. The problem was not very apparent when we tested ISP with the Umpire test suite and the Game of Life program (reported in [24]), which made fewer
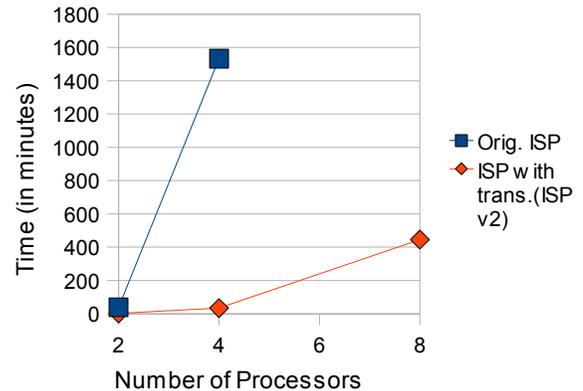

Improvements with Transitive Relation

Figure 4. Improvements based on transitive reduction

than a hundred MPI calls in our testing. In our several first runs, ParMETIS exceeded all available memory allocations.

The problem was attributed to the storage taken by ISP's *Node* structure which maintains the list of transitions for each process. In addition, each transition maintained a list of ancestors which grew quadratically. We will describe our approach to handling this problem in Section 7.1.

Forming match sets is a central aspect of POE. One source of processing overheads in match set formation was located to be the compiler's inability to inline the `.end()` call in loops such as this:

```
for (iter = list.begin(); iter != list.end();
     iter++) {
  ... do something ...
}
```

Improvements at this level had marginal effects on ISP's performance.

7. Improvements to POE

7.1 Transitivity of CB relation

It became obvious that searching through hundreds of thousands of intraCB edges was having a huge effect on ISP's performance. We either needed to store less intraCB edges, or search through less intraCB edges. First, we exploit the fact that *ancestor* is a transitive binary relation, and store only the *immediate ancestor* relation. As the name suggests, immediate ancestor is the *transitive reduction* of the *ancestor* relation – i.e., the smallest binary relation whose transitive closure is *ancestor*. We then realized that the POE algorithm remained correct even if it employed immediate ancestors in match-set formation. The intuitive reason for this lies in the fact that whenever x is an ancestor of y and y is an ancestor of z, a match set involving y would be formed (and fired) before one involving z is formed (and fired).

The graph in Figure 4 shows the improvement of ISP in handling ParMETIS after switching over to the use of immediate ancestors. The testing setup we employed is similar to the `ptest` script provided in ParMETIS 3.1 distribution. To be more specific, our tests involve running rotor.graph, a file that represents a graph with about 100,000 nodes and 600,000 edges, through V3_RefineKWay, followed by a partitioning routine called V3_PartKway, then the graph is repartitioned again with Adap-
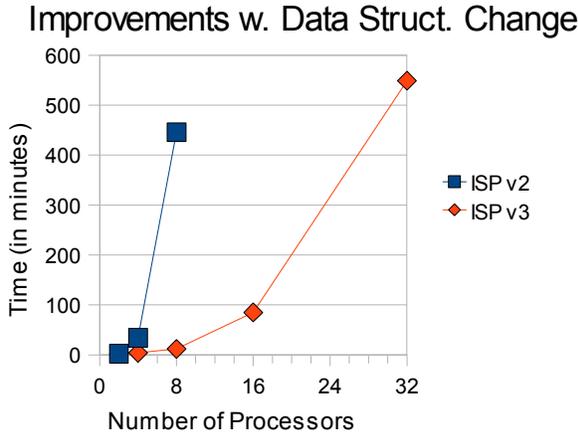
Figure 5. Improvements made by data structures changes

Figure 6. Improvements made by parallelization

tiveRepart. The test completes by running through the same routine again with a different set of options. All the tests were carried out on a dual Opteron 2.8 GHz (each itself is a dual-core), with 4 GB of memory. We also show in Table 1 the number of MPI calls this test setup makes (collectively by all the processes) as the number of processes increase.

The comparison between the original ISP and the modified ISP (dubbed ISP v2 in this study) shows a huge improvement in ISP's processing time. In fact, without the use of immediate ancestors, ISP was not able to complete the test when running with eight processes. Even running one test for 4 processes already took well over a day! In contrast, ISP v2 finishes the test for 4 processes in 34 minutes.

With the change over from ancestors to immediate ancestors, we also made additional data structure simplifications, whose impact is summarized in the graph of Figure 5 (this version of ISP was termed ISP v3).

Even with these improvements, ISP was still taking considerable time to complete model checking ParMETIS for 32 processes, which is almost ten hours. This led us to consider parallelizing the search for ancestors.

7.2 Parallel-ISP

The discovery of where ISP spends most of its processing time leads us to the idea of parallelizing ISP's search for ancestors while building the match-sets. Recall that the MPI calls made by each process of the target program are represented by transition lists. The formation of match sets requires searching through *all* transition lists. Fortunately, these searches are independent of each other, and can be easily parallelized. There are several ways to parallelize this process: (i) make a distributed ISP where each ISP process performs the search for each transition list, or (ii) create

num. of procs	Total MPI calls
2	15,789
4	56,618
8	171,912
16	544,114
32	1,390,260

Table 1. num. of MPI calls vs num of processes
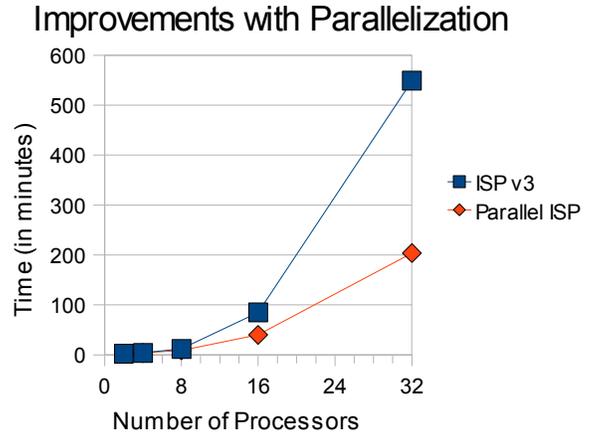
a multithreaded-ISP where each thread performs the search, or (iii) use OpenMP to parallelize the search and let the OpenMP run-time handle the thread creation. We opted for the OpenMP approach due to the fact that the POE scheduler is implemented with many `for` loops – a good candidate for parallelization using OpenMP.

We present the performance results of Parallel ISP vs ISP v3 in Figure 6. Parallelization does not help ISP much when running with a small number of processes. However, when we verify up to 16 and 32 processes, the benefits of parallelization becomes more obvious (On average, Parallel-ISP was about 3 times faster than the serial ISP).

7.3 Handling bugs in large programs

While ParMETIS was verified to be free of deadlocks and assertions violation under some fixed test inputs, we were still interested in knowing whether ISP would detect such errors should ParMETIS have them. In order to test ISP's ability to catch bugs in such a large program, we deliberately inserted some subtle deadlocks into ParMETIS's graph setup module of the RefineKWay routine. The bug is (over-simplified for illustration) described by the following pseudocode:

```
Original code in setup routine:
P0: Irecv (from P1); Irecv (from P2);
    Waitall ();
P1: Isend (to P0); Wait();
P2: Isend (to P1); Wait();
P4: ;

``Seeded'' bug :

P0: Irecv (from *); Irecv (from *);
    Irecv (from P4); Waitall ();
P1: Isend (to P0); Wait();
P2: Isend (to P1); Wait();
P4: Isend (to P0);
```

Obviously the modified code results in a deadlock if the first or second `MPI_Irecv` of P0 matches the `MPI_Isend` issued by P4. However, complex timings in a practical program can easily hide this bug, as shown in the case with this seeded bug in RefineKway: *MPICH failed to deadlock in all of our testing experiments.* This

is not surprising at all, considering how easily testing tools have been observed to miss bugs during the testing of far simpler MPI program. [24]. However, to our reassurance, our seeded bug was revealed by ISP during its exploration of the second interleaving (out of 24 possible relevant interleavings that it would explore under the partial order reduction effected by POE). We also seeded a bug similar to that discussed in Section 4.1 concerning the algorithm unitBred of MADRE. ISP caught this seeded bug during its very first interleaving.

8. Conclusions

In this paper, we have reported our efforts on applying ISP, a "push-button" dynamic model checker for MPI programs, to two practical MPI programs: ParMETIS and MADRE. Both these programs were verified for the absence of deadlocks without requiring any manual modeling effort whatsoever. This is the first formal verification tool that handles C/MPI programs, employs a customized dynamic partial order reduction algorithm to consider only the relevant interleavings of an MPI program, and prints error traces similar to a model checker when a problem is detected. Our results for the first time establish that dynamic partial order reduction based verification methods may play a significant role in verifying API based concurrent programs written in real programming languages.

In fact, ISP's scheduler does not even need to be aware whether the MPI calls are coming from C, C++, or Fortran. The current version of ISP does not support this language neutrality, but mainly due to subtle differences in MPI calling conventions; our future work aims to provide such language neutrality.

Currently, ISP works with most standard-compliant MPI implementations, and can [7] handle 34 functions of MPI 2.0 (a full list is provided at [1]). This website also provides the source codes of all our experiments, as well as for ISP, which may be freely downloaded. Many more MPI functions can be handled straightforwardly (e.g., MPI functions pertaining to the creation and manipulation of MPI datatypes need not be trapped by ISP at all).

As for our practical case studies, while ParMETIS stresses ISP on its performance and overhead, MADRE tests ISP on its ability to catch deadlocks when non-determinism of the MPI runtime is involved (induced by wildcard receive statements). In the case of ParMETIS, ISP was able to verify and *guarantee* that all routines of ParMETIS are free from deadlock and local assertion violations under the provided test inputs (we verified up to 32 processes in our experiments). On the other hand, with MADRE, ISP detected a bug within one of the routines that could potentially deadlock the program (the bug was later discovered to be a known problem).

In Section 4.1, we indicated that for MADRE, ISP needs n! interleavings to safely over-approximate potential senders that can match a wildcard receive. In recent work, we have overcome this limitation, and are able to finish this example within a single interleaving. The details of this new development will be reported in a future publication.

To successfully finish our non-trivial case studies, several enhancements were made to the POE algorithm employed within ISP. Apart from numerous data structure and programming improvements aided by profiling studies, two of the most important advances were: (i) capitalizing on the transitivity of the completes-before relation to represent only its transitive reduction, in the form of *immediate ancestors*, (ii) justifying the correctness of POE even under this reduction, and (iii) parallelization of the POE scheduler using OpenMP, making ISP especially suitable for multi-core CPUs. Further improvements to ISP are possible to make its scheduler even more efficient and scalable. One obvious improvement would be to fork off the verification of the different elements of

a set of match sets (discussed in Section 3.1.2). This type of parallelization has already been demonstrated in a related project by our group [26]. The other aspect of heavy emphasis in our future work will be *bug preserving downscaling methods* – ways to simplify an MPI program using static analysis techniques (e.g., reduce the number of processes represented in it, or even alter the actual algorithm carried out by the program) such that the class of bugs of interest (e.g., deadlocks) are preserved. Such downscaling methods are very promising in terms of the ability to handle even larger MPI programs.

References

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

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

[3] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, pages 110–121. ACM, 2005.

[4] A. Geist. Sustained Petascale: The next MPI challenge. Invited Talk at EuroPVM/MPI 2007.

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

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

[7] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[8] G. Karypis. METIS and ParMETIS. http://glaros.dtc.umn.edu/gkhome/views/metis.

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

[10] B. Krammer, K. Bidmon, M. S. Mller, and M. M. Resch. Marmot: An MPI analysis and checking tool. In *Parallel Computing 2003*, Sept. 2003.

[11] L. Lamport. Time, clocks and ordering of events in distributed systems. *Communications of the ACM*, 21(7):558–565, July 1978.

[12] A. L. Lastovetsky, T. Kechadi, and J. Dongarra, editors. *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI Users' Group Meeting, 2008*, volume 5205 of *Lecture Notes in Computer Science*. Springer, 2008.

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

[14] M. Musuvathi and S. Qadeer. Fair stateless model checking. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 362–371, New York, NY, USA, 2008. ACM.

[15] V. Prasad. *Scalable and Accurate Approaches to Program Dependence Analysis, Slicing, and Verification of Concurrent Object Oriented Programs*. PhD thesis, Kansas State University, 2006.

[16] S. Sharma, S. Vakkalanka, G. Gopalakrishnan, R. M. Kirby, R. Thakur, and W. Gropp. A formal approach to detect functionally irrelevant barriers in mpi programs. In Lastovetsky et al. [12], pages 265–273.

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

[18] S. Siegel. The MADRE manual. http://vsl.cis.udel.edu/madre/.

[19] S. F. Siegel and G. S. Avrunin. Verification of MPI-based software for

---

scientific computation. In *In Proceedings of the 11th International SPIN Workshop on Model Checking Software*, pages 286–303, 2004.

[20] S. F. Siegel and L. F. Rossi. Analyzing BlobFlow: A case study using model checking to verify parallel scientific software. In Lastovetsky et al. [12], pages 274–282.

[21] S. F. Siegel and A. R. Siegel. MADRE: The Memory-Aware Data Redistribution Engine. In Lastovetsky et al. [12], pages 218–226.

[22] S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, and R. M. Kirby. Scheduling considerations for building dynamic verification tools for MPI. In *Parallel and Distributed Systems - Testing and Debugging (PADTAD-VI)*, July 2008.

[23] S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, R. Thakur, and W. Gropp. Implementing efficient dynamic formal verification methods for mpi programs. In Lastovetsky et al. [12], pages 248–256.

[24] S. Vakkalanka, G. Gopalakrishnan, and R. M. Kirby. Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. In *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 66–79. Springer, 2008.

[25] J. S. Vetter and B. R. de Supinski. Dynamic software testing of MPI applications with Umpire. In *Supercomputing*, pages 70–79, 2000.

[26] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Distributed dynamic partial order reduction based verification of threaded software. In *SPIN,Lecture Notes in Computer Science*, pages 58–75. Springer, 2007.

[27] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Efficient stateful dynamic partial order reduction. In *SPIN*, *Lecture Notes in Computer Science*, pages 288–305. Springer, 2008.