Precise Dynamic Analysis for Slack Elasticity: Adding Buffering Without Adding Bugs

Sarvani Vakkalanka, Anh Vo, Ganesh Gopalakrishnan, and Robert M. Kirby

School of Computing, Univ. of Utah, Salt Lake City, UT 84112, USA

Abstract. Increasing the amount of buffering for MPI sends is an effective way to improve the performance of MPI programs. However, for programs containing non-deterministic operations, this can result in *new* deadlocks or other safety assertion violations. Previous work did not provide any characterization of the space of *slack elastic* programs: those for which buffering can be safely added. In this paper, we offer a precise characterization of slack elasticity based on our formulation of MPI's *happens before* relation. We show how to efficiently locate *potential culprit sends* in such programs: MPI sends for which adding buffering can increase overall program non-determinism and cause new bugs. We present a procedure to minimally enumerate potential culprit sends and efficiently check for slack elasticity. Our results demonstrate that our new algorithm called POE_{MSE} which is incorporated into our dynamic verifier ISP can efficiently run this new analysis on large MPI programs.

1 Introduction

A common myth is that if an MPI program does not deadlock under zero buffering for the sends, it will not deadlock under increased buffering. This myth has been expressed in many places [3, 4, 6]. Previous work [10, 16] shows that while *deterministic* message passing programs enjoy this property, non-deterministic ones do not: they can exhibit new deadlocks or new non-deadlock safety violations when buffering is added. Those programs unaffected by increased buffering are called *slack elastic* [10]. Therefore, any developer wanting to improve MPI program performance by increasing runtime buffering needs to first make sure that the program is slack elastic.

Importance of detecting slack inelasticity: It is expected that MPI will continue to be an API of choice for at least another decade for programming large-scale scientific simulations. Large-scale MPI implementations must be formally verified to avoid insidious errors suddenly cropping up during field operation. Of all the formal verification methods applicable to MPI, dynamic methods have shown the most promise in terms of being able to instrument and analyze large programs. Examples range from scalable semi-formal approaches such as [5] that analyze large MPI programs for specific properties such as deadlocks, and formal *dynamic partial order* reduction methods such as in our ISP tool [18–21].

A relatively neglected aspect of previous MPI program formal analysis methods is how buffering affects correctness. A dynamic verifier for MPI must not only verify a given MPI program on a given platform – it must also ideally verify it *across all future* platforms. In other words, these tools must conduct *verification for safe portability*. Users of an MPI program must be allowed to switch to an MPI runtime that employs aggressive buffer allocation strategies or port the program to a larger machine and increase the `MP_EAGER_LIMIT` environment variable value without worrying about new bugs; for instance:

• Some MPI runtimes [7] perform credit-based buffer allocation where a program can perceive time-varying eager limits. This can create situations where the eager limit varies even for different instances of the same MPI send call.

• Future MPI programs may be generated by program transformation systems [2], thus creating 'unusual' MPI call patterns. Therefore, simple syntactic rules that guard against slack inelastic behavior are insufficient.

• Misunderstanding about buffering are still prevalent. Here are examples: *If you set* `MP_EAGER_LIMIT` *to zero, then this will test the validity of your MPI calls* [4]. A similar statement is made in [6]. The paper on the Intel Message Checker [3] shows this as the recommended approach to detect deadlocks.

Past work [10, 16] did not offer a precise characterization of slack elastic programs nor a practical analysis method to detect its violation. Many important MPI programs (*e.g.*, MPI-BLAST [12], and ADLB [9]) exploit non-deterministic receives and probes to opportunistically detect the completion of a previous task to launch new work. Therefore, avoiding non-determinism is not a practical option, as it can lead to code complexity and loss of performance. We offer the first precise characterization and an efficient dynamic analysis algorithm for slack inelasticity.

• We describe a new dynamic analysis algorithm called POE_{MSE} that can efficiently enumerate what we term *potential culprit sends*. These are MPI sends occurring in an MPI program to which adding buffering can increase the overall program non-determinism. This increased non-determinism potentially results in unexplored behaviors and bugs.

• We search for potential culprit sends over the graphs defined by MPI's *happens before*. We also contribute this happens-before relation (a significant extension of the *intra happens-before* [21]) as a new result. As we shall show, happens-before incorporates MPI's message non-overtaking and wait semantics.

• The new version of ISP containing POE_{MSE} is shown to perform efficiently on non-trivially sized MPI programs. Given the limited amount of space, we opt to present our results intuitively, with some background § 2, present the POE_{MSE} algorithm § 3, and results § 4. Formal details are presented in [17].

2 Motivating Examples

We begin with the familiar 'head to head' sends example Figure 1(a). To save space and leave out incidental details of MPI programming, we adopt the following abbreviations (explained through examples). We prefer to illustrate our

ideas using non-blocking send/receive operations just to be able to discuss the effects of buffering on `MPI_Wait`s. We use $S_{0,0}(1)$ to denote a non-blocking send (`MPI_Isend`) targeting process 1 ($P1$). The subscript $0,0$ says that this is send number 0 of $P0$ (counting from zero). Without loss of generality, we assume all these communication requests carry the same tag, happen on the same communicator, and communicate some arbitrary data. Also, any arbitrary C statement including conditionals and loops may be placed between these MPI calls. We use $W_{0,1}(h_{0,0})$ to denote the `MPI_Wait` corresponding to $S_{0,0}(1)$. Similarly, $R_{0,2}(0)$ stands for an `MPI_Irecv` sourcing $P0$, and $W_{0,3}(h_{0,2})$ is its corresponding `MPI_Wait`. It is clear that this example will deadlock under zero buffering but not under infinite buffering. This is a well-known MPI example underlying almost all "zero buffer" deadlock detection tests mentioned earlier.

Deadlock Detection by Zeroing Buffering is Inconclusive: The MPI program in Figure 1(b) will deadlock when either $S_{0,0}$ or $S_{1,0}$ or both are buffered. There is no deadlock when both $S_{0,0}$ and $S_{1,0}$ have buffering.

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

(a) Zeroed buffering finds deadlocks)

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

(b) Zeroed buffering misses deadlocks

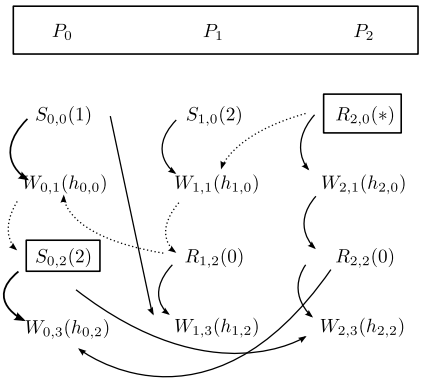
Fig. 1: Where zeroing buffering helps, and where it does not

If both $S_{0,0}(1)$ and $S_{1,0}(2)$ have zero buffering, their corresponding `MPI_Wait` operations $W_{0,1}(h_{0,0})$ and $W_{1,1}(h_{1,0})$ remain blocked until (at least) the corresponding receives are issued. Of the two sends, only $S_{1,0}(2)$ can proceed by matching $R_{2,0}(*)$ (standing for a wildcard receive, or a receive with argument `MPI_ANY_SOURCE`). This causes the waits $W_{1,1}(h_{1,0})$ to unblock, allowing $R_{1,2}(0)$ to be posted. This allows $S_{0,0}(1)$ to complete and hence $W_{0,1}(h_{0,0})$ will return, allowing $S_{0,2}(2)$ to issue. Since $W_{2,1}(h_{2,0})$ has unblocked after $R_{2,0}(*)$ matches, $R_{2,2}(0)$ can be issued, and then the whole program finishes without deadlocks. Now, if $S_{0,0}(1)$ were to be buffered, the following execution can happen: First, $S_{0,0}(1)$ can be issued. The corresponding wait, *i.e.* $W_{0,1}(h_{0,0})$ can return regardless of whether P_2 has even posted its $R_{2,0}(*)$. This can now result in $S_{0,2}(2)$ to be issued, and this send competes with $S_{1,0}(2)$ to match with $R_{2,0}(*)$. Suppose $S_{0,2}(2)$ is the winner of the competition, *i.e.* it matches $R_{2,0}(*)$. This now leads to a deadlock with respect to $R_{2,2}(0)$ because the only process able to satisfy this request is P_0, but unfortunately this process has no more sends.

Neither Zero Nor Infinite Buffering Helps: In the example of Figure 1(b), even if all sends are given infinite buffering, we will run into the deadlock with respect to $R_{2,2}(0)$ described earlier. This may give us the impression that either zero buffering or infinite buffering will catch all deadlocks. This is not so! The example in Figure 2(a) will not deadlock when none of the sends are buffered or

P_0	P_1	P_2
$S_{0,0}(1)$	$S_{1,0}(2)$	$R_{2,0}(*)$
$W_{0,1}(h_{0,0})$	$W_{1,1}(h_{1,0})$	$W_{2,1}(h_{2,0})$
$S_{0,2}(2)$	$R_{1,2}(0)$	$R_{2,2}(0)$
$W_{0,3}(h_{0,2})$	$W_{1,3}(h_{1,2})$	$W_{2,3}(h_{2,2})$
$S_{0,4}(2)$	$R_{1,4}(2)$	$S_{2,4}(1)$
$W_{0,5}(h_{0,4})$	$W_{1,5}(h_{1,4})$	$W_{2,5}(h_{2,4})$
$R_{0,6}(1)$	$S_{1,6}(0)$	$R_{2,6}(*)$
$W_{0,7}(h_{0,6})$	$W_{1,7}(h_{1,6})$	$W_{2,7}(h_{2,6})$

(a) Specific buffering causes deadlock



(b) Slack Inelasticity modeled as Happens-before path breaking

Fig. 2: More Buffering-related Deadlocks

all the sends are buffered. However, it will deadlock only when (i) $S_{0,0}$ is buffered and (ii) both $S_{1,0}$ and $S_{2,4}$ are not buffered, as detailed in § 3.

There is one common simple explanation for all the behaviors seen so far. To present that, we now introduce, through examples, the *happens before* (HB) relation underlying MPI that we have discovered. We adapt Lamport's happens-before [8] – widely used in programming to study concurrency and partial order semantics – for MPI (for full explanations, please see [17, 21]). We will now use Figure 2(b), which essentially redraws Figure 1(b) with the HB edges added, to illustrate the precedences MPI happens-before. There is a HB edge between: (i) Every non-blocking send/receive and its wait. This can be seen in Figure 2(b) as the solid arrows from, say $S_{0,0}(1)$ to $W_{0,1}(h_{0,0})$. (ii) MPI waits and their successive instructions. We show this as, for example, $W_{2,1}$ to $R_{2,2}$. Notice that $W_{1,1}$ to $R_{1,2}$ and $W_{0,1}$ to $S_{0,2}$ are also HB ordered; these are shown dotted because they happen to fall on a HB path (*HB-path*) which we highlight via dotted edges. (iii) Collective operations (including barriers) and their successive MPI instructions, two non-blocking sends targeting the same destination (MPI non-overtaking), and two receives sourcing from the same source (exhaustive list is available [17]).

So far we introduced *intra* HB; the rule for inter HB is roughly as follows: if two instructions can match, their successors are in the *inter* HB relation with respect to each other. For example, since $S_{0,0}$ and $R_{1,2}$ can match in the execution of this program under zero-buffering, we have an inter HB edge from $S_{0,0}$ to $W_{1,3}$ and another from $R_{1,2}$ to $W_{0,1}$. (These happen to be shown dotted because they lie on the HB-path.)

Now, here is what buffering of $S_{0,0}$ does:

• It allows $W_{0,1}$ to return early (since the message is buffered, this wait can return early – or it becomes a no-op). Now consider the dotted path from $R_{2,0}(*)$ to $S_{0,2}(2)$. Before buffering was added to $S_{0,0}$, this path had all its edges. When

buffering was added to $S_{0,0}$ thus turning $W_{0,1}$ in effect to a no-op, we broke this HB-path.
• We show that when a send (such as $S_{0,2}(2)$) and a receive (such as $R_{2,0}(*)$) do not have an HB-path separating them, they become concurrent [17]. We saw this earlier because as soon as we buffered $S_{0,0}$, $S_{0,2}(2)$ became a *competing* match to $R_{2,0}(*)$. The **potential culprit send** is $S_{0,0}$ because it was by buffering this that we made $S_{0,2}(2)$ and $R_{2,0}(*)$ match, thus increasing the overall non-determinism in the program.
• Therefore the POE_{MSE} algorithm is (detailed in § 3 but briefly now): (i) execute under zero buffering; (ii) build the HB-paths separating potential competing sends such as $S_{0,2}$ with wildcard receives such as $R_{2,0}(*)$; (iii) locate potential culprit sends such as $S_{0,0}$ (if any) that can break HB-paths thus making these new sends match a wildcard receive, increasing non-determinism; (iv) if the increased non-determinism leads to bugs, the potential culprit sends are actual culprit sends.

In this example buffering $S_{0,0}$ indeed lead to a deadlock, and so we located an actual culprit send. Hence, this program is *not slack elastic*!

3 The POE_{MSE} Algorithm

The POE_{MSE} algorithm is an extension of the POE algorithm [18–21]. We present POE_{MSE} at a high level using the examples in Figure 1(a) first, Figure 1(b) next, and then that in Figure 2(a) last. All formal details are in [17].

We run the program as per our POE algorithm under zero buffering for all sends. Running the example in Figure 1(a) instantly reveals the deadlock, allowing the user to fix it. For Figure 1(b), POE will simply run through the code without finding errors in the first pass. Being a stateless dynamic verifier, ISP only keeps a stack history of the current execution. The happens-before graph is built as in Figure 2(b). Next, ISP unwinds the stack history. For each wildcard receive encountered, it will find out which sends *could have matched* should other sends (potential culprit sends) have buffering. In our case, it will find that $S_{0,2}(2)$ could have matched $R_{2,0}(*)$ if we were to break the HB-path by buffering $S_{0,0}$ as said before.

Coming to Figure 2(a), when POE_{MSE} initially executes forward, it runs the whole program under zero buffering. It would find that $R_{2,0}(*)$ matched $S_{1,0}(2)$, and $R_{2,6}(*)$ matched $S_{2,4}(2)$. Then stack unwinding proceeds as follows:
• When $R_{2,6}(*)$ is popped from the stack, POE_{MSE} will try to force another sender to match the wildcard receive $R_{2,6}(*)$. It does not find any culprit sends that can be buffered to make it happen. This is clear because if another send were to match $R_{2,6}(*)$, it has to come from $P1$ (MPI's non-overtaking). However, there is no such sender.
• Thus, POE_{MSE} will unwind more, finally popping $R_{2,0}(*)$. At this point, POE_{MSE} will find that the culprit send of $S_{0,0}(1)$ indeed works, because buffering this send immediately turns $W_{0,1}$ into a no-op, breaking the HB-path $S_{0,0}(1) \rightarrow W_{0,1} \rightarrow S_{0,2}(2) \rightarrow R_{2,0}(*)$ at $W_{0,1} \rightarrow S_{0,2}(2)$.

Number of interleavings (notice the extra necessary interleavings of POE_{MSE})	POE_{MSE}	POE
sendbuff.c	5	1
sendbuff-1a.c	2 (deadlock caught)	1
sendbuff2.c	1	1
sendbuff3.c	6	1
sendbuff4.c	3	1
Figure 2(a)	4 (dl caught)	2 (dl missed)
ParMETIS_b	2	1
Overhead of POE_{MSE} on ParMETIS / ParMETIS_* (runtime in seconds (x) denotes x interleavings)	POE_{MSE}	POE
ParMETIS (4procs)	20.9 (1)	20.5 (1)
ParMETIS (8procs)	93.4 (1)	92.6 (1)
ParMETIS_*	18.2 (2)	18.7(2)

Table 1: ParMETIS_b is ParMETIS_* w. slack; ParMETIS_* is ParMETIS modified to use wildcards

• Now POE_{MSE} will replay forward from this stack frame, initially giving no buffering at all to subsequent sends. This will cause a head-to-head deadlock between $S_{1,0}(2)$ trying to send to P_2 and $S_{2,4}(1)$ trying to send to P_1.
• If we were to buffer $S_{1,0}(2)$, this head-to-head deadlock will disappear. This is why we may need to buffer some sends (culprit sends) and not buffer other sends (head-to-head deadlock inducing sends).
In [17], we explain all these details, provides actual pseudo-codes and mathematical proofs. Here are additional facts:
• In general, we may find *multiple potential culprit sends*. More than one might need to be buffered to break an HB-path. However, it is also important to break HB-paths in a minimal fashion –*i.e.*, giving a "flood of buffering" is not a good idea because we can mask later head-to-head deadlocks. Thus POE_{MSE} always allocates buffering for potential culprit sends only.
• It is only for deadlock checking that we need these precautions. All non-deadlock safety violations can be checked with infinite buffering, which is simulated as follows: we `malloc` as much buffer space as the message the MPI send is trying to send, and copy away the data into it, and nullify `MPI_Wait`. Thus, ISP when running POE_{MSE} really converts MPI waits into no-ops.

4 Results and Conclusions

We first study variants of the examples in Figure 1, Figures 1(b) and 2(a) (called `sendbuff`). These examples explore POE_{MSE}'s capabilities to detect the different matchings as well as deadlocks. For each of the sendbuff variants, POE_{MSE} correctly discovers the minimal number of send operations to be buffered in

order to enable other sends to match with wildcard receives. We also reproduced our example in Figure 2(b) as sendbuff-1a.c, where our algorithm indeed caught the deadlock at the second interleaving, where $S_{0,2}(2)$ is matched with $R_{2,0}(*)$.

Next we study large realistic examples that show that POE_{MSE} adds virtually no overheads. We used ParMETIS [14, 15], a hypergraph partition library (14K LOC of MPI/C), as a benchmark for measuring the overhead of POE_{MSE} (shown in Table 1 as ParMETIS (xprocs) where x is the number of processes that we ran the benchmarks with. $ParMETIS_*$ is a modified version where we rewrote a small part of the algorithm using wildcard receives. In most of our benchmarks where no additional interleavings are needed, the overhead is less than 3%, even in the presence of wildcard receives, where the new algorithm has to run extra steps to make sure we have covered all possible matchings in the presence of buffering.

Finally, we study large examples with slack inelastic patterns inserted into them. This is reflected in Table 1 as $ParMETIS\_b$ where we rewrote the algorithm of ParMETIS again, this time not only to introduce wildcard receives, but also to allow the possibility of a different order of matching that can only be discovered by allowing some certain sends to be buffered. Our experiment shows that POE_{MSE} successfully discovered the alternative matching during the second interleaving. POE_{MSE} has handled all the large examples previously handled by POE with only negligible overhead in practice.

The POE_{MSE} algorithm in our actual ISP tool handles over 60 widely used MPI functions, and hence is practical for many large MPI programs.

4.1 Concluding Remarks

In addition to MPI, MCAPI [11], CUDA [1], and OpenCL [13] include the same kind of non-blocking calls and waits discussed here. Programmers worry about the cost (the amount of memory) tied up by buffering. Memory costs and memory power consumption already outweigh those figures for CPUs. Thus one likes to allocate buffer space wisely.

Non-deterministic reception is an essential construct for optimization. Unfortunately non-deterministic reception and buffering immediately leads to slack variant behaviors. The code patterns that cause slack inelastic behaviors are not that complex – meaning, they can be easily introduced.

We propose the POE_{MSE} algorithm that detects such behaviors based on an MPI-specific happens-before relation. It works by first locating all minimal sets of non-blocking message send operations that must be buffered, so as to enable other message send operations to match wildcard receives; and subsequently running the dynamic analysis over all such minimal send sets. The overhead of these steps is negligible in practice. A promising avenue of future research is to detect slack inelastic behaviors during the design of new APIs.

Acknowledgements: We thank Grzegorz Szubzda and Subodh Sharma for their valuable contributions, and Bronis R. de Supinski for his encouraging remarks.

References

1. Compute Unified Device Architecture (CUDA). http://www.nvidia.com/object/cuda_get.html.
2. A. Danalis, L. Pollock, M. Swany, and J. Cavazos. Mpi-aware compiler optimizations for improving communication-computation overlap. In *ICS '09*.
3. J. DeSouza, B. Kuhn, B. R. de Supinski, V. Samofalov, S. Zheltov, and S. Bratanov. Automated, scalable debugging of mpi programs with Intel Message Checker. In *SE-HPCS '05*.
4. http://www.hpcx.ac.uk/support/FAQ/eager.html.
5. T. Hilbrich, B. R. de Supinski, M. Schulz, and M. S. Müller. A graph based approach for MPI deadlock detection. In *ICS 2009*, pages 296–305.
6. http://www.cs.unb.ca/acrl/training/general/ibm_parallel_programming/pgm3.PDF.
7. PE MPI buffer management for eager protocol. http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp?topic=/com.ibm.cluster.pe431.mpiprog.doc/am106_buff.html.
8. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.
9. R. Lusk, S. Pieper, R. Butler, and A. Chan. Asynchronous dynamic load balancing. unedf.org/content/talks/Lusk-ADLB.pdf.
10. R. Manohar and A. J. Martin. Slack elasticity in concurrent computing. In *Proceedings of the Fourth International Conference on the Mathematics of Program Construction*, pages 272–285. Springer-Verlag, 1998. LNCS 1422.
11. http://www.multicore-association.org.
12. http://www.mpiblast.org.
13. OpenCL: Open Computing Language. http://www.khronos.org/opencl.
14. ParMETIS - Parallel graph partitioning and fill-reducing matrix ordering. http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview.
15. K. Schloegel, G. Karypis, and V. Kumar. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience*, 14:219–240, 2002.
16. S. F. Siegel. Efficient verification of halting properties for MPI programs with wildcard receives. In R. Cousot, editor, *VMCAI 2005, Paris, January 17–19, 2005, Proceedings*, volume 3385 of *LNCS*, pages 413–429, 2005.
17. S. Vakkalanka. *Efficient Dynamic Verification Algorithms for MPI Applications*. PhD Dissertation, 2010. http://www.cs.utah.edu/~sarvani/dissertation.html.
18. 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)*, Seattle, WA, July 2008.
19. S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, R. Thakur, and W. Gropp. Implementing efficient dynamic formal verification methods for MPI programs. In *EuroPVM/MPI*, volume 5205 of *LNCS*. Springer, 2008.
20. S. Vakkalanka, G. Gopalakrishnan, and R. M. Kirby. Dynamic Verification of MPI Programs with Reductions in Presence of Split Operations and Relaxed Orderings. In *Computer Aided Verification (CAV 2008)*, pages 66–79, 2008.
21. S. Vakkalanka, A. Vo, G. Gopalakrishnan, and R. M. Kirby. Reduced execution semantics of mpi: From theory to practice. In *FM 2009*, pages 724–740, Nov. 2009.