Received: 18 December 2020 | Accepted: 26 April 2021

DOI: 10.1002/cpe.6380

SPECIAL ISSUE PAPER

WILEY

Not all applications have boring communication patterns: Profiling message matching with BMM

Taylor Groves[1] | Naveen Ravichandrasekaran[2] | Brandon Cook[1] | Noel Keen[1] | David Trebotich[1] | Nicholas J. Wright[1] | Bob Alverson[2] | Duncan Roweth[2] | Keith Underwood[2]

[1]Lawrence Berkeley National Laboratory, Berkeley, California, USA

[2]Cray, an HPE Company, Seattle, Washington, USA

Correspondence
Taylor Groves, Lawrence Berkeley National Laboratory, Building 59, 1 Cyclotron Rd, Berkeley, CA 94720, USA.
Email: tgroves@lbl.gov

Summary

Message matching within MPI is an important performance consideration for applications that utilize two-sided semantics. In this work, we present an instrumentation of the CrayMPI library that allows the collection of detailed message-matching statistics as well as an implementation of hashed matching in software. We use this functionality to profile key DOE applications with complex communication patterns to determine under what circumstances an application might benefit from hardware offload capabilities within the NIC to accelerate message matching. We find that there are several applications and libraries that exhibit sufficiently long match list lengths to motivate a Binned Message Matching approach.

KEYWORDS

message matching, MPI, offload NIC, tag matching

1 | INTRODUCTION

Two-sided MPI operations are the well-used form of send and receive operations, where both sending and receiving processes are engaged in message processing and forwarding (in contrast to one-sided operations such as *put* and *get*). More than 90% of HPC applications rely on two-sided (i.e., point-to-point) communication, according to recent survey.[1] Two-sided MPI provides deterministic in-order processing of messages between sender and receiver through a procedure called tag matching. Because MPI does not guarantee the message to arrive after the receive is posted, tag matching involves searching two lists to deterministically match a message to its target buffer. When a message arrives, a queue of expected messages (posted receive queue—PRQ) is searched for a match of metadata (source, MPI communicator, and tag). Otherwise the metadata is appended to a second queue called the unexpected message queue (UMQ). When a receive is posted, the UMQ is searched for a match using the same criteria before the operation can be added to the PRQ. The speed of these operations can have a significant impact on communication performance.[2] Typically these queues are implemented as singly linked lists, which perform well for small message counts, but for complex and irregular applications that may have a larger number of messages. While alternative designs have been proposed,[3-6] their usage is not well motivated by studies of real applications.

By default, most MPI implementations maintain message queues as a single linked list. HPE MPI also offers an alternate "Binning Message Matching" (BMM) algorithm to implement message matching. This feature is exposed within a non-default Cray MPI implementation on current generation HPE hardware. By breaking a single list into several bins, it is possible to have more efficient operation by reducing search lengths. Binned matching algorithms are made more complex by the usage of wildcard receives, because ordering must be reconciled between the bins and any wildcard receives.[4]

HPE has been investigating techniques that allow users to analyze MPI message matching queue usage for DOE applications. MPI queue usage is an important consideration in the design and development of future NIC hardware. Usage is thought to vary widely between applications based on their communication characteristics. In this work, we examine a range of mini applications and full applications that are important to the DOE community. This study is meant to motivate the need for maintaining multiple bins for message queues, and to identify the potential impact from wildcard operations. We also consider the applicability of binning to both the PRQ and UMQ. These workloads span a range of idioms in MPI usage and expose interesting behavior that informs hardware and software design.

2 | BACKGROUND

2.1 | MPI message matching

MPI supports two classes of communication known as one-sided and two-sided communications. Two-sided communications (e.g., MPI_Send, MPI_Recv) and non-blocking versions of those (e.g., MPI_Isend, MPI_Irecv) were introduced in MPI-1 with one-sided operations (e.g., MPI_Put) arriving in MPI-2. Because of its earlier availability and the long life cycle of scientific applications, two-sided communications became the most common form of communication within MPI based applications;[1] thus, our studies here focus on the usage characteristics of two-sided operations and their implication for MPI implementations.

Two-sided messaging within MPI matches a message to a receive buffer based on three criteria: the MPI communicator, the source rank, and the message tag. Though the communicator field must be specified, wildcards may be used for the source and tag fields by any receiving process. A receiving process may provide many receive buffers with the same matching criteria. To provide determinism, two-sided messaging guarantees deterministic processing with respect to the order the message is sent and the receive is called within a communicator. Specifically, the default ordering in MPI says that messages between a pair of ranks will be processed in the order that MPI_Send (or its variants) is called. It also states that incoming messages on a communicator will choose a receive buffer based on matching criteria and the order the MPI_Recv (or its variants) is called. Because of these requirements, traditional MPI implementations have used simple linked list structures that keep track of the order in which messages arrive and receives are posted. These linked lists store identifier fields to distinguish between messages by source rank, communicator, and message tag.

When any two-sided message request arrives at the target, one of two scenarios unfolds. In the first scenario, the target process has already anticipated the arrival of the message and has prepared an entry in a PRQ that allows it to determine where to deliver the message payload. In the second scenario, the target process has not yet prepared for the source's message and the message matching fields must be pushed onto an UMQ. To elaborate, as a target process prepares for the receipt of a message, it must first make sure that the message has not already been delivered. This is accomplished by searching the UMQ. Similarly, as the target process receives a message, it must first search the PRQ to determine if the message was expected or not. If it was expected, it has the necessary information to deliver the message to the appropriate target buffer. Otherwise the message is pushed into the UMQ. This abstract process of message matching is illustrated in Figure 1.

The ability to match wildcard fields further complicates the matching process. Even if more complex data structures are used for message matching (e.g., a hashmap based off of source, tag, and communicator), those structures must account for wildcard entries. Entries with wildcards in the fields being hashed must be handled in a separate linked list. Thus, in the limit, this degrades to where performance is limited by the number of outstanding messages with wildcard fields. This work attempts to answer questions related to the actual usage patterns of applications to enable better choices about data structures and hashing in the future.

2.2 | Accelerated message matching

In determining the requirements for future hardware, it is important to understand the characteristics of posted receive lists and unexpected message lists. Is their use sufficiently widespread to justify acceleration via dedicated matching logic or would hardware design time be better used elsewhere? Would it have a significant impact on the cost of NIC parts? If use of long receive/message lists is widespread, then parameters of interest are the number of bins that can be used efficiently and the prevalence of wildcard matching. The statistics collected in this study have been designed to help answer these questions.

Message matching performance is such an important aspect of MPI performance that it is now common for NICs to offload some of the cost associated with matching from the host CPU—such as Mellanox ConnectX-5[8] and the Bull-Atos's BXI.[9] This offload enables asynchronous progress that can improve application performance; however, one of the challenges of designing such hardware is designing it with the necessary capabilities to process a high number of messages per second—which impacts both processing and storage capacity. These details vary by application communication pattern and workload. Though there are existing studies for a selection of HPC workloads, evaluations of a wide variety of workloads continue to provide valuable information to the MPI community.

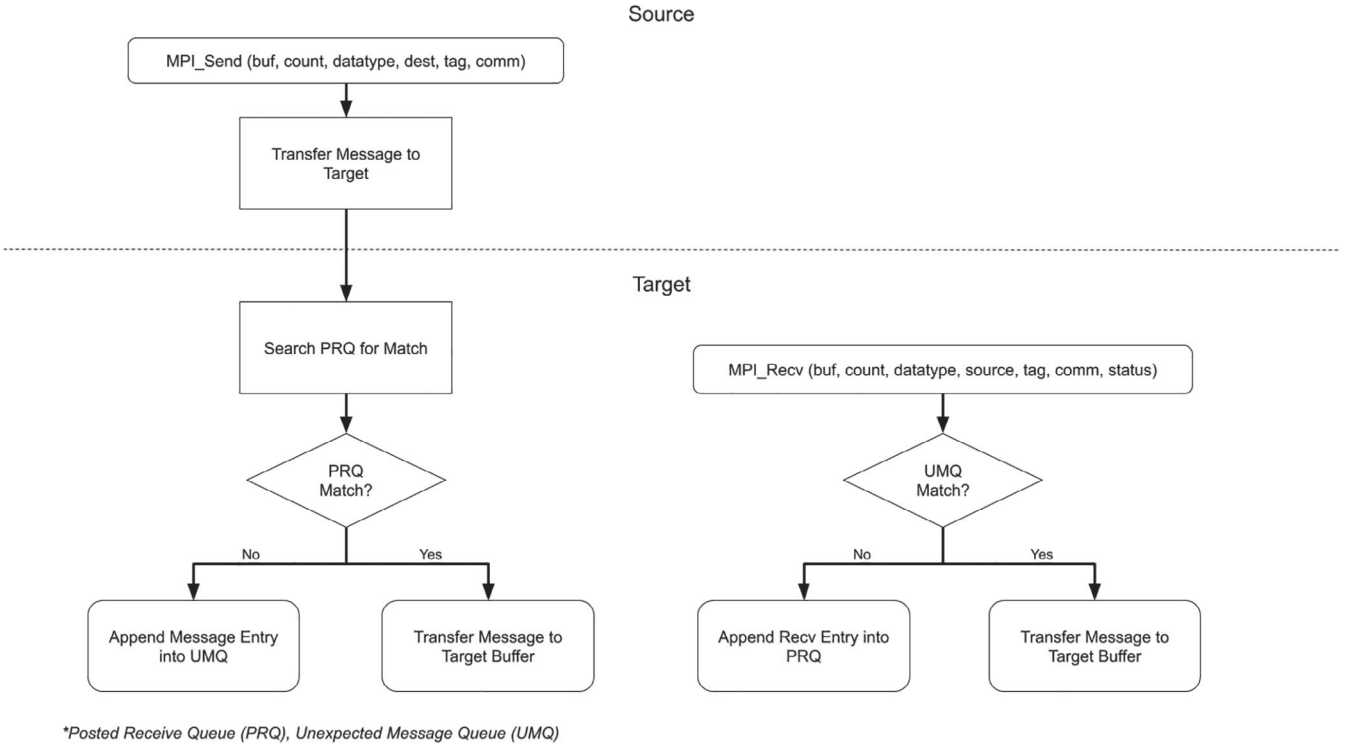

FIGURE 1 Flowchart highlighting key transitions in MPI message matching for source and target process. Original figure from work by Ferreira et al.[7]

2.3 | Workload selection

Many existing workloads of HPC centers exhibit predictable and well-structured communication patterns with a fixed number of neighbors. This includes things like a nearest neighbor or halo exchange, where the number of neighbors corresponds with a geometry found in the three-dimensional world being simulated.

In this work, we expand the set of applications that have been analyzed previously by selecting representative workloads that exhibit complex communication and I/O patterns, such as an adaptive mesh refinement (AMR). The set of applications are:

AMReX: AMReX[10] is designed to support parallel, block-structured AMR applications. AMReX creates a hierarchy of MPI_Comm objects that can be used to split work.

Chombo: Chombo[11] provides tools for high resolution applied PDE simulators in arbitrarily complex geometry. Chombo provides AMR with embedded boundaries to represent heterogeneous materials (e.g., shale).

E3SM: E3SM[12] is a fully coupled model of the Earth's climate including biogeochemical and cryospheric processes. Because of the complexity of the processes being modeled there are a wide variety of ways E3SM may be configured and utilized. Our runs are atmosphere-only (F case) simulating 5 days and include the writing of a large IO file.

MILC: MILC[13] is a 4D mesh simulation of quantum chromodynamics, the theory of the strong interactions of subatomic physics. Communication is characterized by non-blocking point to point operations followed by small message MPI_Allreduce.

3 | METHODOLOGY AND EVALUATION

3.1 | System specification

Our evaluation was performed on NERSC Cori. Cori is a Cray XC40 system with Aries interconnect. The Aries NIC does not provide message matching offload, requiring the message matching to be performed on the host CPU. Cori consists of 2388, dual socket Intel Xeon Haswell (32 cores) nodes and 9688, Intel Xeon Phi Knight's Landing (KNL 68 cores) nodes. Each Haswell node has 128 GB DDR4 2133 MHz memory (four 16 GB DIMMs per socket). Each KNL node has 96 GB DDR4 2400 MHz memory (six 16 GB DIMMs) as well as 16 GB of MCDRAM.

3.2 | BMM configuration and output

For our experiments, we take advantage of the Binned Message Matching (BMM) implementation that we can enable/disable within Cray MPICH. BMM allows us to generate detailed statistics of MPI matching performance for the configured number of bins.

3.2.1 | Singly linked list versus BMM

By default, Cray MPICH uses a singly linked list for maintaining the message queues. To use the BMM algorithm, the following environment variable needs to be set to 1: MPICH_USE_BINNING_MSG_MATCH. Similarly, the following environment variables control the number of bins in receive and unexpected queues, respectively: MPICH_NUM_POST_RECV_BINS and MPICH_NUM_UNEXPECTED_BINS. By default, the BMM implementation in Cray MPI uses four bins for receive queue and one bin for unexpected queue. To enable instrumentation of the BMM implementation, users need to set the _MPICH_GET_BMM_INFO environment variable to 1. An instrumentation report can be generated in a user-defined file using the _MPICH_GET_BMM_INFO_FILE environment variable.

3.2.2 | Data generated

The following statistics are collected for each rank in the user-defined file in CSV format. (1) High-water mark length of each bin for receive and unexpected queues, (2) Average number of match attempts for each bin, (3) Maximum number of match attempts for each bin, (4) Total number of messages matched for each bin, (5) Number of transitions from wild-card to non-wild card bins performed in the receive queues (though these transitions are not explored in this study).

3.3 | AMReX

Runs of AMReX were done on the NERSC Cori KNL supercomputer across 2176 processes. Two levels of AMR were used. While AMReX did not use wildcard operations, it had interesting PRQ and UMQ characteristics.

The data for the PRQ show a split in the message matching behavior as shown in Figure 2(A). Typical match lengths searched were seven while a single rank (rank 0) had an average search length of over 200 with a maximum approaching 500. This shows significant load imbalance with regards to the number of messages received by rank 0 compared to other ranks.

The data for the UMQ (Figure 2(B)) show a wider spread distribution compared to the PRQ; however, rank 0 is still an outlier with around 80 comparisons per match on average and a peak of nearly 500 (not shown) for the UMQ. The average comparisons per match were 7 across all ranks.

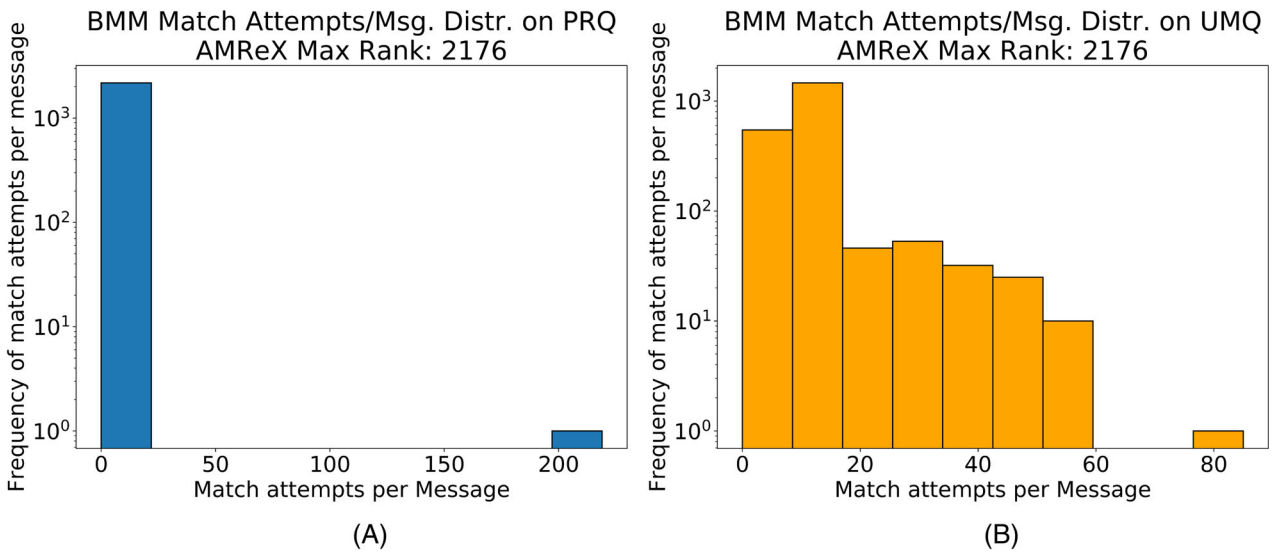

FIGURE 2 Histograms of the average match attempts per message while running AMReX for (A) the posted receive queue, and (B) the unexpected message queue

The AMReX data highlight a couple of important application characteristics. Foremost, unexpected messages are common. Optimizing only for the expected message case will miss important use cases. In addition, there is significant load-imbalance in the match attempts per rank. Optimizations should focus on reducing the impact of this imbalance. Finally, dynamic allocation of resources is clearly important.

3.4 | Chombo

Chombo was run on the NERSC Cori KNL partition ranging in sizes from 8 to 16,384 ranks. The problem size was increased to match the number of ranks (i.e., weak scaling). For brevity, we examine runs of 512 ranks 4096 ranks and 16,384 ranks to explore how queue usage scales with the number of ranks.

Irrespective of whether the run was across 512, 4096, or 16,384 ranks we see the same split in the histograms of PRQ (Figure 3), where a subset of processes have less than 20 comparisons per match and the remaining processes see between 40 and 140 average comparisons per match. Even going as far out as 16,384 processes, we only see a mild increase in average comparisons per match.

For Chombo, the UMQ (Figure 4) sees a smaller number of average comparisons for each match with most ranks having less than 20 comparisons per match. Nonetheless, a minority of ranks encountered more than 50 comparisons per match on average in the UMQ. This is likely a result of minor load imbalance that causes some receivers to fall slightly behind their peers.

When we received results from Chombo we were surprised to see wildcard usage that the developers had not previously expected. The high watermark was up to 25 for the wild card bin. Heavy usage of the wild card bin will quickly negate the advantages of a hashed matching implementation, so we decided to explore this result further.

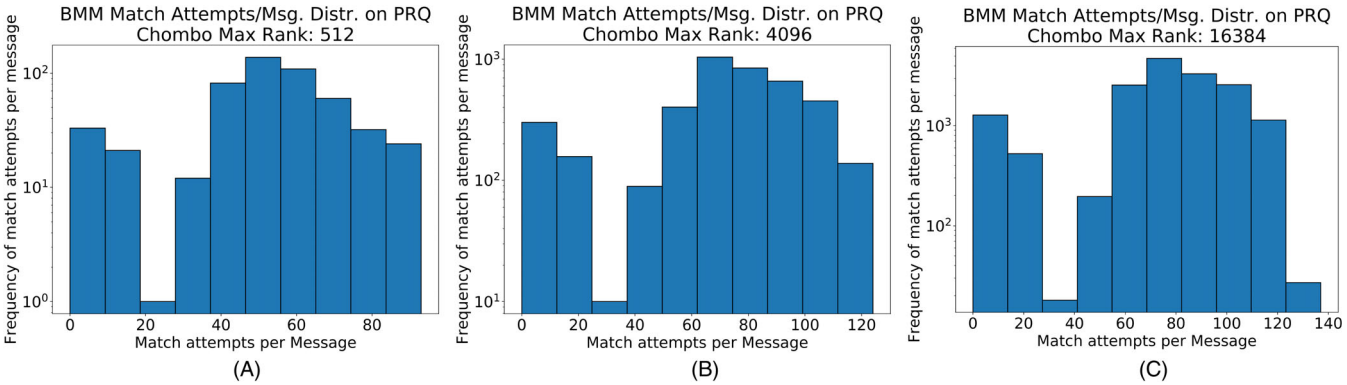

FIGURE 3 Three histograms showing average comparisons made per match for the PRQ for Chombo weak-scaled at 512, 4096, and 16,384 processes. The average comparisons range between 20 and 140

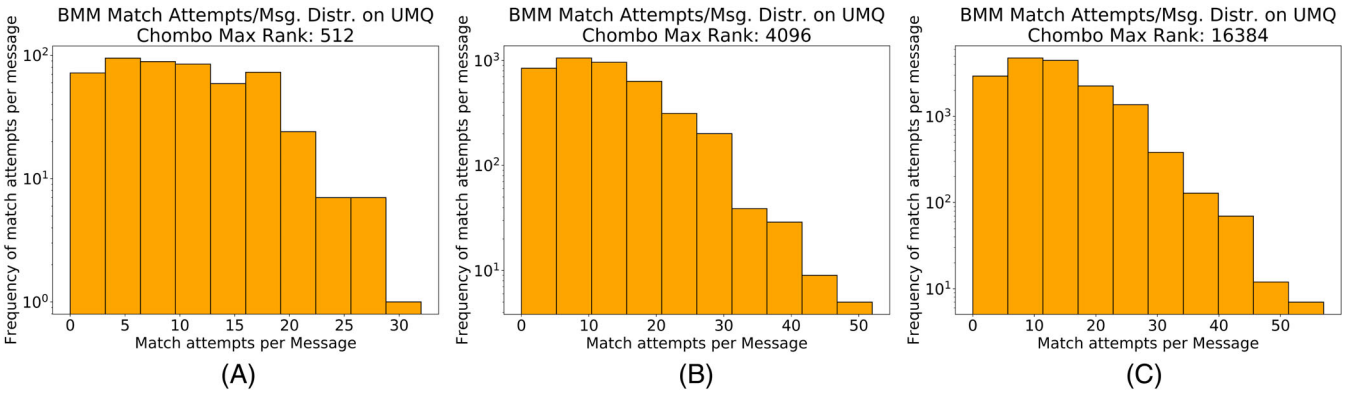

FIGURE 4 Three histograms showing average comparisons made per match for the UMQ for Chombo weak-scaled at 512, 4096, and 16,384 processes. The average comparisons range between 20 and 50

3.4.1 | Chombo, take-two

The message matching behavior observed for Chombo was surprising to the authors and the developers, who initially thought it would be similar to the other AMR code evaluated (AMReX). To investigate further we evaluated Chombo again, but this time used different math libraries. A simple change of math libraries resulted in dramatically different communication patterns and message matching behavior, which closely matched the results of AMReX. A detailed comparison showing statistics collected for each rank is shown in Figures 5 and 6. Furthermore, swapping the libraries resulted in a shift to zero wildcard usage.

Comparing Figures 5 and 6 shows dramatic differences. In the early runs, the average comparison depth in the PRQ was over 4 times what it was after the math libraries were swapped. An average depth of 13 is a relatively easy search problem when compared to a depth of 52. The more dramatic difference, however, was the wide variation in the per-rank search depths. When some ranks are traversing a list 100 items deep while others are traversing a list 10 items deep, the MPI matching time can lead to substantially different network performance. In contrast, the second set of math libraries led to a much narrower range of match depth.

The original Chombo data highlight one of the fundamental challenges in collecting relevant data: it is easy to create a configuration with poor network characteristics. Notably, though, a user might make the configuration choices in the original run without being aware of the additional pressure it puts on the MPI implementation. Many large applications have literally dozens of libraries—each with its own usage of MPI. It is unlikely for a user to truly understand all of those libraries, or the network characteristics of them. A solver library, for example, is more likely to be chosen for its numerical stability on the problem than its MPI queue usage characteristics. This highlights the importance of running a wide variety of applications but also, studying different inputs, libraries, and parameters.

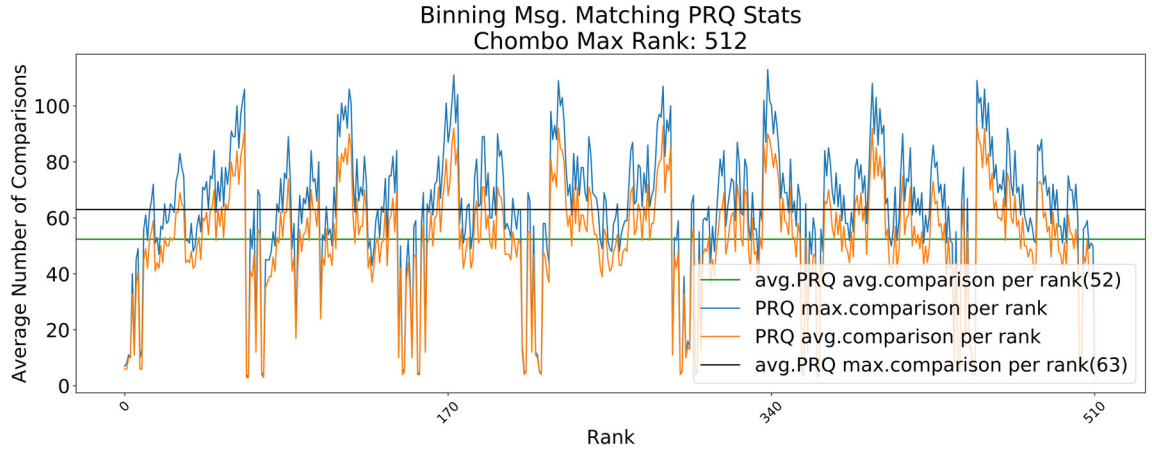

FIGURE 5 Average number of comparisons per message match (*Y*-axis) for the PRQ, with 512 process runs of Chombo before swapping out math libraries. *X*-axis is the MPI rank identifier

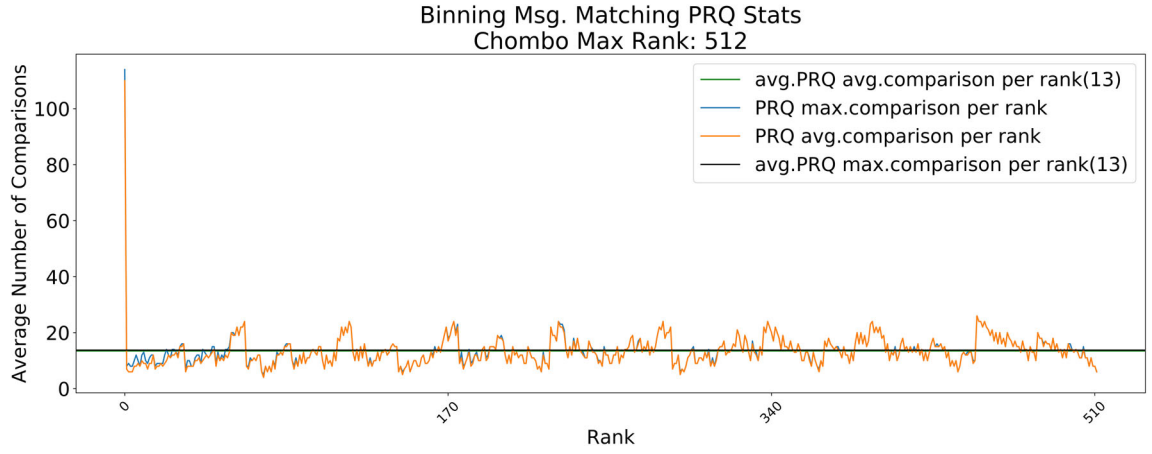

FIGURE 6 Average number of comparisons per message match (*Y*-axis) for the PRQ, with 512 process runs of Chombo after swapping out math libraries. *X*-axis is the MPI rank identifier

3.5 | E3SM

Runs of E3SM were executed on Cori KNL with strong scaling of 169, 323, and 1350 nodes with up to 86,400 MPI ranks for the largest runs. A significant amount of communication takes place as part of I/O. There is no standard way that I/O is done within E3SM, and methods depend on what variables are requested for a given simulation and how frequent the data is written (i.e., once per hour, once per day, once per month). One common scenario is that the most expensive file is written at the end of every month. However, for the purposes of our analysis we run 5 days at quarter degree resolution before writing the large output file. We utilize PIO version 2.

The average number of comparisons done per match was 17, 15, and 20 for runs of 169, 323, and 1350 nodes, respectively. For each run, there was a small number of ranks (approximately 16–32 processes) that incurred substantially higher maximum match lengths (peaking at approximately 120 comparisons). For brevity, we only show the results for the largest run in Figure 7.

The UMQ had the same average comparisons per match as the PRQ at the respective scales. Again, for each run, there was a small number of ranks (approximately 16–32 processes) that incurred substantially higher maximum match lengths. Because the UMQ data are so similar to the PRQ we omit the figure. In Figure 8, we show the impact of scaling E3SM process counts on UMQ match attempts. As the process and node count increase by 8-fold (approximately 10,000–80,000 processes), we see some increases to the tail of the distribution as a minority of matches encounter greater list lengths and require a greater number of attempts for a successful match.

At each scale evaluated we observed wildcard usage only by MPI rank 0. No other ranks utilized the wildcard matching bin. For rank 0, the average attempts per match in the wild-bin were 38. This same statistic was observed at each of the three scales evaluated.

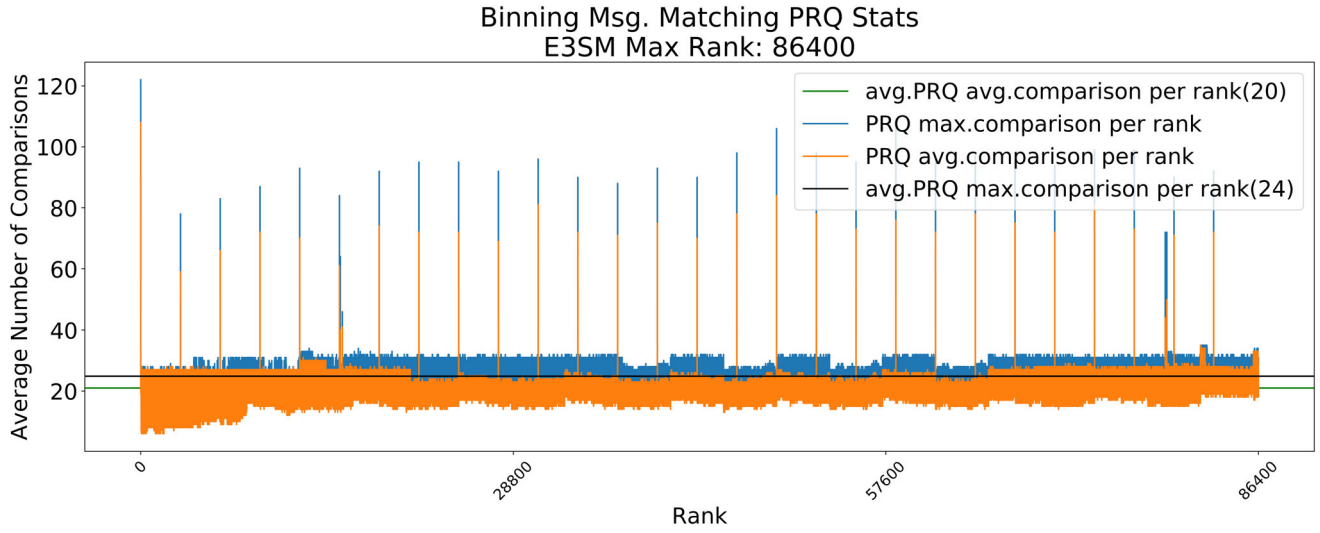

FIGURE 7 Average match attempts per message across all ranks (*x*-axis) for the posted receive queue while running E3SM

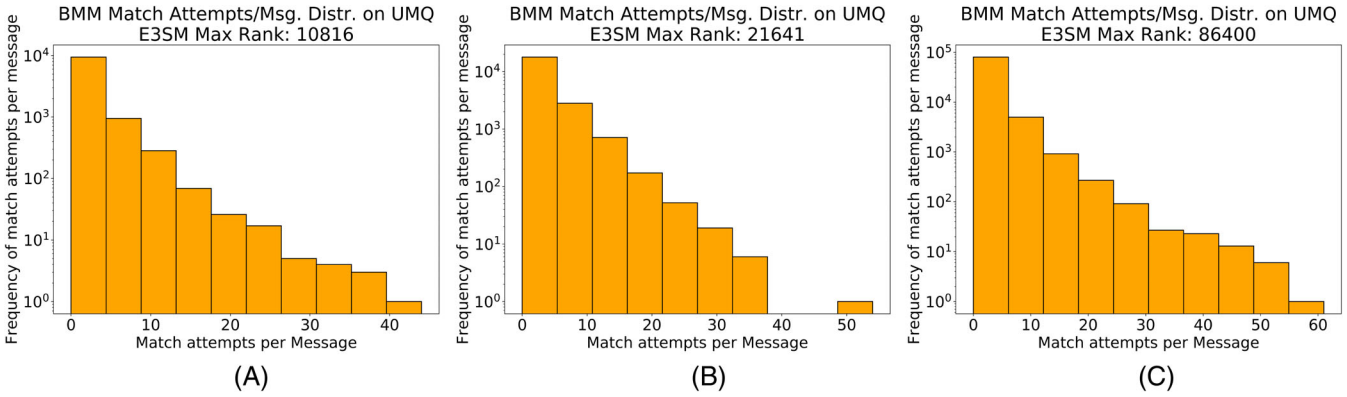

FIGURE 8 Three histograms showing average comparisons made per match for the UMQ for E3SM scaled at 10k, 21k, and 86k processes

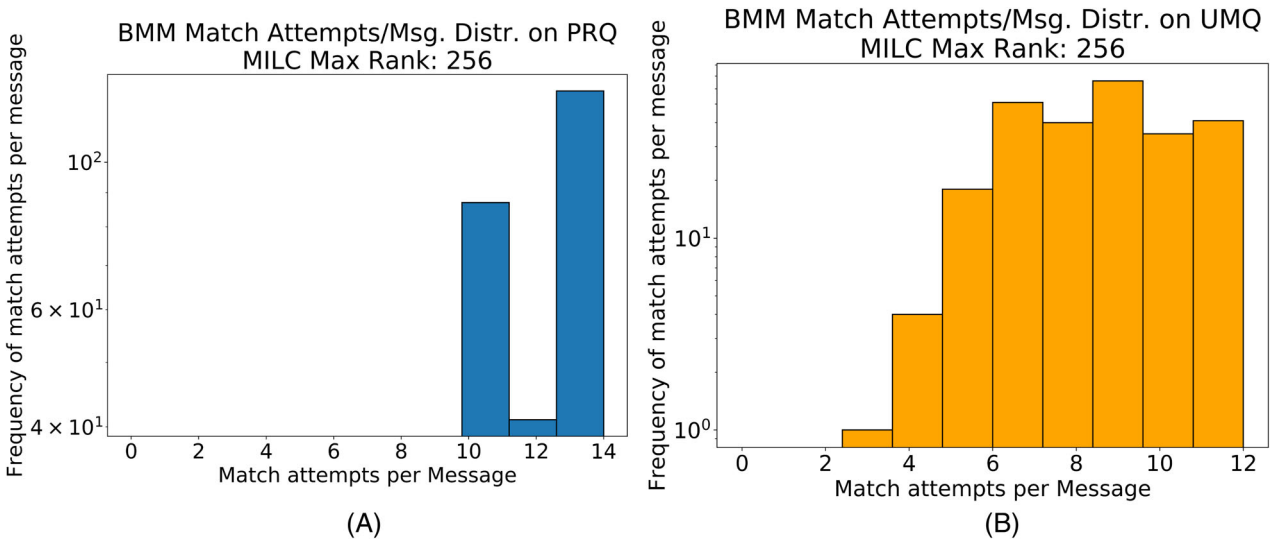

FIGURE 9 Histogram of average attempts per message match for the PRQ and UQM while running MILC on 256 nodes

3.6 | MILC

MILC exhibits a regular communication pattern with non-blocking point to point communication in four dimensions. These exchanges are followed by small MPI_Allreduce operations. This type of communication is common to many applications performing simulations on large scale HPC systems. We ran MILC at scales of 32–256 processes on the NERSC Cori KNL partition.

As we scaled from 32 to 256 processes we saw the average number of comparisons vary from 9 to 12 per match, respectively, with peaks of 14 comparisons. The difference between MILC's regular communication pattern and the other workloads examined (AMReX, Chombo, and E3SM) is notable as MILC has an order of magnitude smaller match list lengths and substantially less variation between ranks.

Average UMQ attempts per match are similar in magnitude to the PRQ. The majority of attempts in the UMQ ranges from 6 to 12. Overall MILC has substantially fewer attempts per match compared to the other applications explored in this work. Results are displayed in Figure 9.

MILC utilizes wildcards and the average match attempts per message in the wild-bin ranged between 9 and 14. This corresponds to use of `MPI_ANY_SOURCE`, within MILC's `do_gather` function. This function is called repeatedly for each node that contains a neighboring site of a given process. Given the fixed number of neighbors within a simulation process (irrespective of scale), we expect the length of wild-bin to remain roughly unchanged if we increased process count beyond 256.

4 | CONCLUSIONS

As we compare results across the four evaluated workloads we see drastically different realities and implications for message matching acceleration. While not all applications leverage MPI's wildcards for point to point communication, many important applications do. Furthermore, libraries these applications depend on may leverage these features in ways that are unknown to users without additional profiling with infrastructure such as BMM.

In the example of AMReX and Chombo, we saw two different behaviors dependent on the underlying libraries included in the application. In one instance, large message matching list lengths were observed across multiple ranks. However, using a different math library was able to isolate the behavior to a single MPI process (rank 0).

As we ran climate simulations in E3SM, we observed how complex I/O behavior resulted in large list lengths for a fixed set of processes.

Lastly, we saw how adaptive communication patterns compared with more static communication patterns as observed in MILC. We noted that many interesting workloads have match lengths an order of magnitude larger than those of traditional stencil applications.

These results inform future hardware design and lay the groundwork for ensuring appropriate resources are dedicated to process messages efficiently without having to fall back to traditional CPU processing of MPI message matching. In many cases, we observe large message matching requirements that necessitate the need for multiple bins to enable efficient matching. As a result, applications will run faster and HPC systems can produce more science.

DATA AVAILABILITY STATEMENT

Data available on request from the authors.

ORCID

Taylor Groves https://orcid.org/0000-0002-7020-8881
Brandon Cook https://orcid.org/0000-0002-4203-4079

REFERENCES

1. Laguna I, Marshall R, Mohror K, Ruefenacht M, Skjellum A, Sultana N. A large-scale study of MPI usage in open-source HPC applications. Paper presented at: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. Denver, CO: ACM/IEEE; 2019:1-14.
2. Underwood KD, Brightwell R. The impact of MPI queue usage on message latency. Paper presented at: Proceedings of the International Conference on Parallel Processing (ICPP), Montreal, Que; 2004:152-160.
3. Underwood KD, Hemmert KS, Rodrigues A, Murphy R, Brightwell R. A hardware acceleration unit for MPI queue processing. Paper presented at: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS), Denver, CO; 2005:10-pp.
4. Flajslik M, Dinan J, Underwood KD. Mitigating MPI message matching misery. Paper presented at: Proceedings of the International Conference on High Performance Computing, Hyderabad, India; 2016:281-299.
5. Zounmevo JA, Afsahi A. A fast and resource-conscious MPI message queue mechanism for large-scale jobs. *Future Generat Comput Syst*. 2014;30:265-290.
6. Hemmert KS, Underwood KD, Rodrigues A. An architecture to perform NIC based MPI matching. Paper presented at: Proceedings of the 2007 IEEE International Conference on Cluster Computing, Austin, TX; 2007:211-221.
7. Ferreira K, Grant RE, Levenhagen MJ, Levy S, Groves T. Hardware MPI message matching: insights into MPI matching behavior to inform design. *Concurr Comput Pract Exper*. 2020;32(3):e5150.
8. Understanding MPI tag matching and rendezvous offloads (Connectx-5); 2020. https://conunurity.mellanox.com/docs/DOC-2583.
9. Derradji S, Palfer-Sollier T, Panziera JP, Poudes A, Atos FW. The BXI interconnect architecture. Paper presented at: Proceedings of the 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects. Santa Clara, CA: IEEE; 2015:18-25.
10. Zhang W, Almgren A, Beckner V, et al. AMReX: a framework for block-structured adaptive mesh refinement. *J Open Source Softw*. 2019;4(37):1370-1370.
11. Colella P, Graves DT, Ligocki T, et al. Chombo software package for AMR applications design document; 2009. http://seesar.lbl.gov/ANAG/chombo/. Accessed September 2008.
12. E3SM; 2020. https://github.com/E3SM-Project/E3SM.
13. MILC; 2020. https://github.com/milc-qcd.

How to cite this article: Groves T, Ravichandrasekaran N, Cook B, et al. Not all applications have boring communication patterns: Profiling message matching with BMM. *Concurrency Computat Pract Exper*. 2021;e6380. https://doi.org/10.1002/cpe.6380