Not All Applications Have Boring Communication Patterns: Profiling Message Matching with BMM

Taylor Groves*, Naveen Ravichandrasekaran†, Brandon Cook*, Brian Friesen*,
Noel Keen*, David Trebotich*, Nicholas J. Wright*, Bob Alverson†, Duncan Roweth†, Keith Underwood†
* *Lawrence Berkeley National Lab*
† *Cray, an HPE Company*

***Abstract*—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 significantly long match list lengths which motivates a Binned Message Matching approach.**

***Index Terms*—MPI, Message Matching, Tag Matching, Offload NIC**

I. Introduction

Two-sided MPI operations are the well used form of send and receive operations, such that 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. In tag matching, there are two lists which are searched to establish this criteria. When a message arrives, a queue of expected messages (posted receive queue) 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). The speed of these operations can have a significant impact on communication performance [2]. Typically these queues are implemented as singly linked lists, which performs well for small message counts, but for complex and irregular applications that may have a larger number of messages, alternative designs have been proposed [3]–[6].

HPE has been investigating techniques that allow users to analyze MPI Message matching queue usage for DOE 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.

MPI queue usage is an important consideration in the design and development of future NIC hardware. Usage is thought to vary widely between applications and 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. These workloads span a range of idioms in MPI usage and expose interesting behavior that informs hardware and software design.

II. Background

A. MPI Message Matching

MPI supports two classes of communication known as one-sided and two-sided communications. In this work we are focused on two-sided (e.g. MPI_Send, MPI_Recv), which is the most common form of communication within MPI, and was the first style of communication specified within the MPI standard.

Within an MPI communicator two-sided messaging guarantees deterministic processing with respect to the order the message is received. This guarantee is enabled through the use of several match queues, that keep track of the order in which messages arrive and store identifier fields to distinguish between messages by source rank, communicator, and message tag. Though the communicator field must be specified, wildcards may be used for the source and tag fields by any receiving process.

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 Posted Receive Queue (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 Unexpected Message Queue (UMQ). To elaborate, as a target process prepares for the receipt of a

Correspondence: Taylor Groves, tgroves@lbl.gov.
This manuscript has been authored by an author at Lawrence Berkeley National Laboratory under Contract No. DE-AC02-05CH11231 with the U.S. Department of Energy. The U.S. Government retains, and the publisher, by accepting the article for publication, acknowledges, that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for U.S. Government purposes.

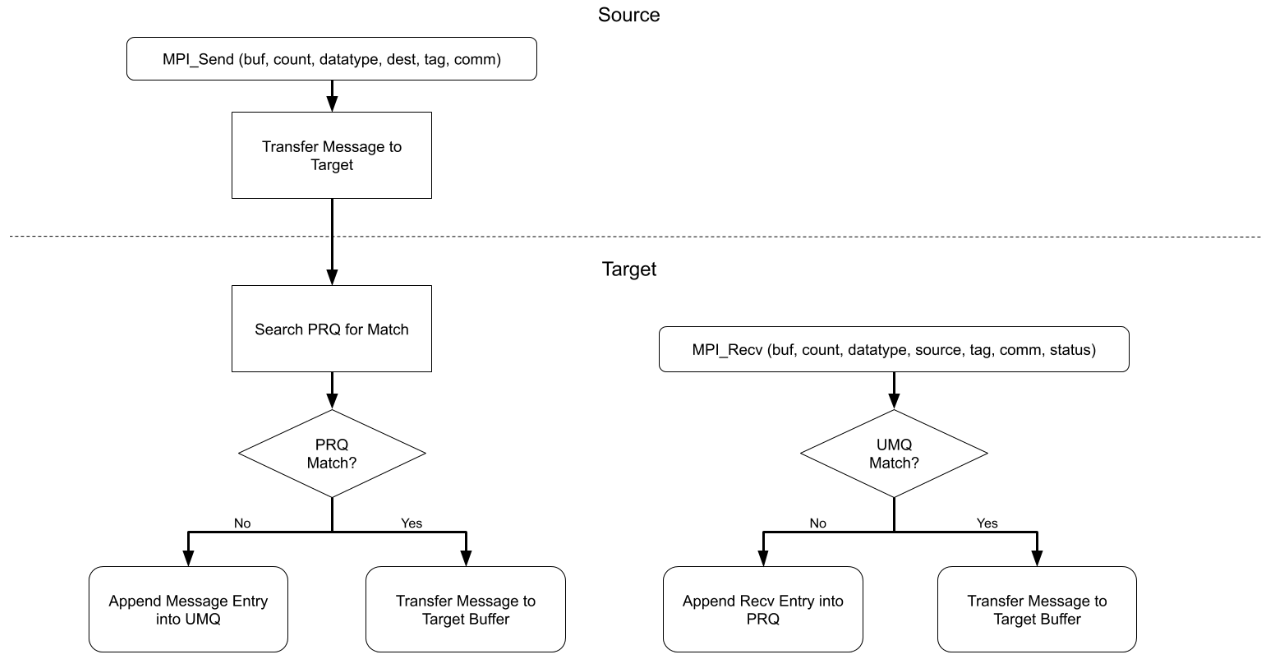

Fig. 1: Flowchart highlighting key transitions in MPI message matching for source and target process. Original figure from work by Ferreira et al. [7].

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.

The ability to match wildcard fields further complicates this process, making it so that even if more complex data structures are used for message matching (e.g. a hashmap based off of source, tag and communicator), performance is still limited by the number of outstanding messages with wildcard fields.

Thus, every two-sided communication is limited by the time to search the UMQ and the PRQ. This abstract process of message matching is illustrated in Figure 1.

B. 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 and store a certain number of messages per second. This number varies 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.

C. 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. The set of applications are:

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

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

E3SM [12]: E3SM 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 [13]: 4D mesh simulates quantum chromodynamics (QCD), 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.

III. Methodology and Evaluation

For our experiments we take advantage of the Binned Message Matching (BMM) implementation that we can enable/disable within Cray MPICH.

a) Singly linked list vs Binned Message Matching: By default, the 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 user-defined file using the _MPICH_GET_BMM_INFO_FILE environment variable.

b) 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).

A. AMReX

a) Run environment: Runs of AMReX were done on the NERSC Cori KNL supercomputer across 2176 processes. Two levels of adaptive mesh refinement were used.

b) PRQ: The data for the PRQ shows a split in the message matching behavior as shown in Figure 2. 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.

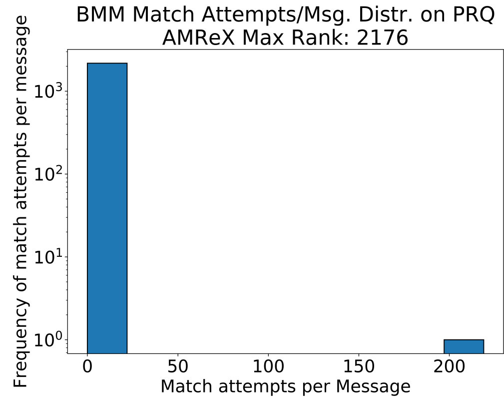

Fig. 2: Histogram of average match attempts per message for the Posted Receive Queue while running AMReX.

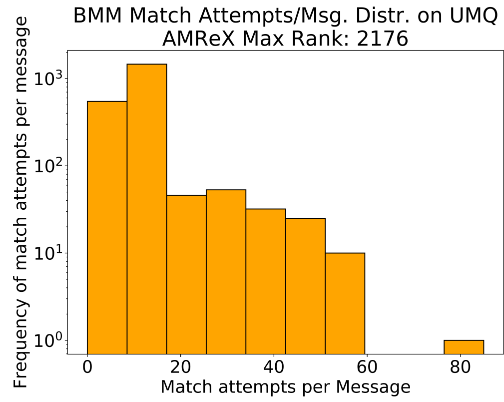

Fig. 3: Histogram of average match attempts per message for the Unexpected Message Queue while running AMReX.

c) UMQ: The data for the UMQ (Figure 3) shows 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 was 7 across all ranks.

d) Wildcard Usage: There was no reported wildcard usage within AMReX.

B. Chombo

a) Run environment: Chombo was run on the NERSC Cori KNL partition, weak scaling from 8 to 16,384 ranks. For brevity we examine runs between 512 ranks 1,024 ranks and 16,384 ranks.

b) PRQ: Irrespective of whether the run was across 512, 4,096 or 16,384 ranks we see the same split in the histogram of PRQ, 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.

c) UMQ: For Chombo the UMQ sees a smaller number of average comparisons for each match with most ranks having

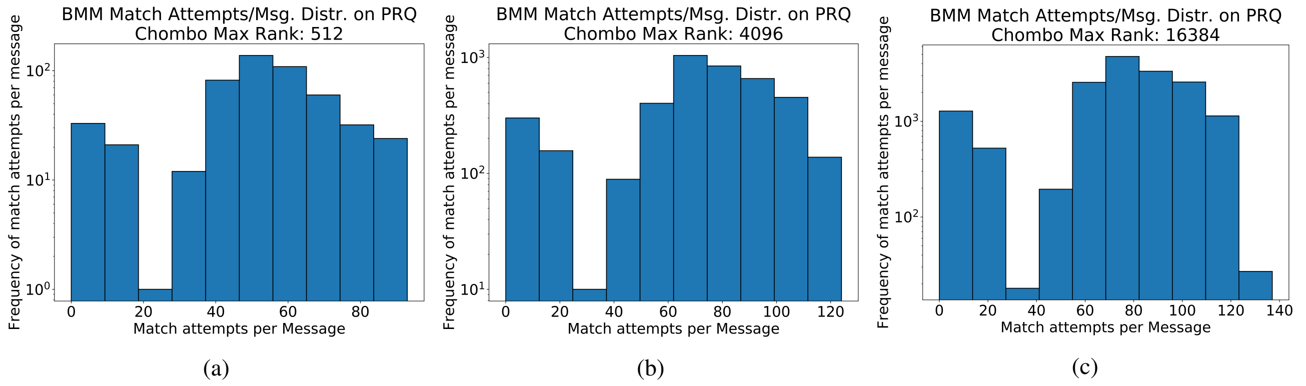

Fig. 4: Three histograms showing average comparisons made per match for the PRQ for Chombo weak-scaled at 512, 4096 and 16384 processes. The average comparisons ranges between 20 and 140.

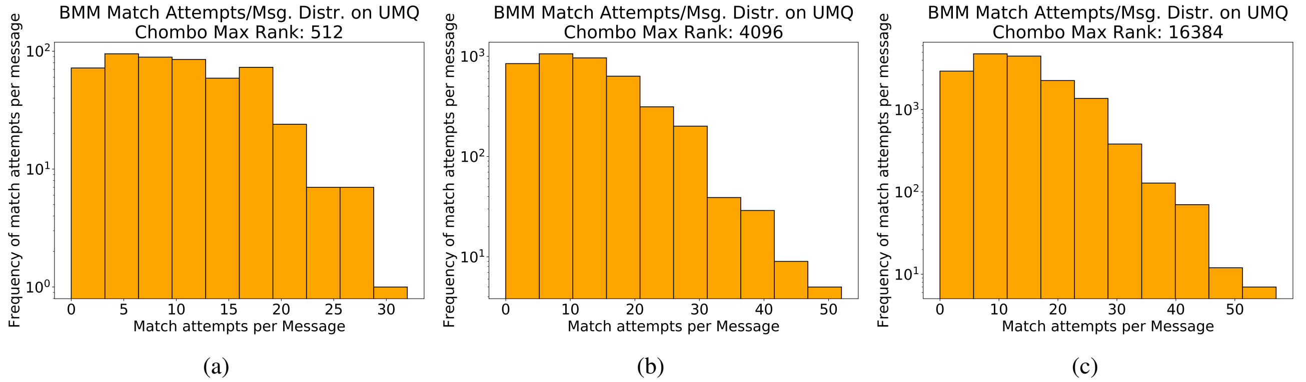

Fig. 5: Three histograms showing average comparisons made per match for the UMQ for Chombo weak-scaled at 512, 4096 and 16384 processes. The average comparisons ranges between 20 and 50.

less than 20 comparisons per match but a minority enduring more than 50 comparisons per match on average.

d) Wildcard usage: When we received results from Chombo we were surprised to see wildcard usage that the developers had not previously expected, with a high watermark of up to 25 for the wild card bin.

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 swapped out 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 6 and 7.

Furthermore, swapping the libraries resulted in a shift to zero wildcard usage. This highlights the importance of running a wide variety of applications but also, studying different inputs, libraries and parameters.

C. E3SM

a) Run environment: Our runs of E3SM were done on Cori KNL with strong scaling of 169, 323 and 1,350 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.

b) PRQ: The average number of comparisons done per match was 17, 15 and 20 for runs of 169, 323 and 1,350 nodes, respectively. For each run, there was a small number of ranks (approximately 16 to 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 8.

c) UMQ: 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 to 32 processes) that incurred substantially higher maximum match lengths. Because the UMQ data is so similar to the PRQ we omit the figure.

d) Wildcard usage: At each scale evaluated we observed wildcard usage only by MPI rank 0. No other ranks utilized the wildcard matching bin.

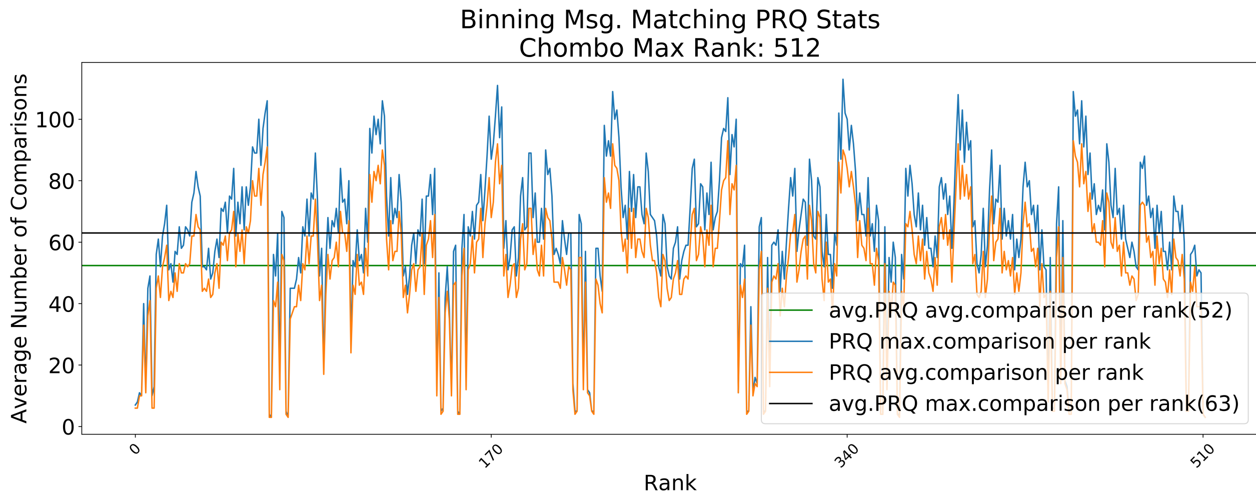

Fig. 6: Average number of comparisons per message match (Y-axis) for the PRQ, with 512 process runs of Chombo before swapping out math libraries. X-axis is the MPI rank identifier.

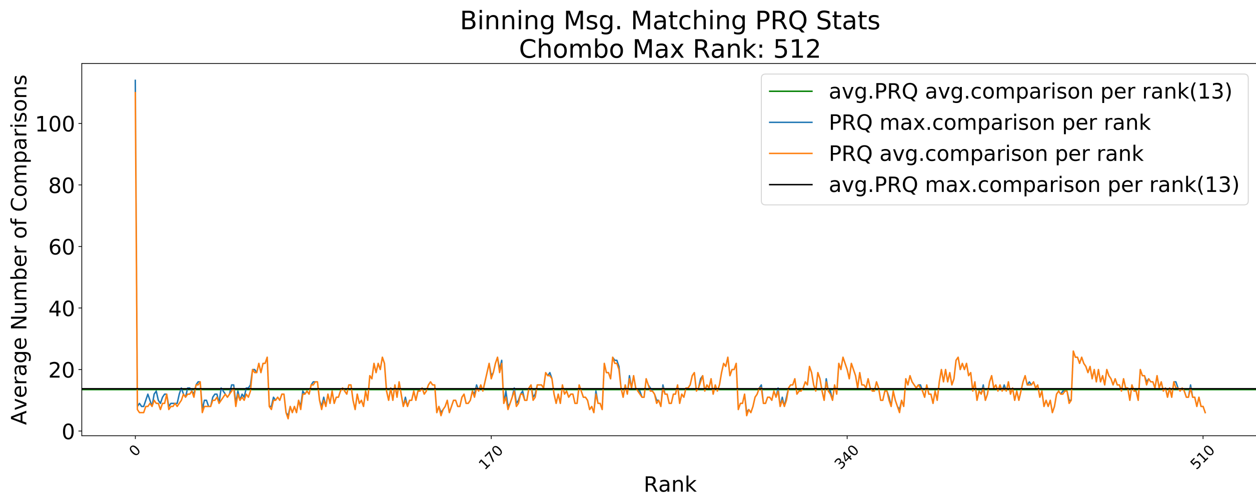

Fig. 7: Average number of comparisons per message match (Y-axis) for the PRQ, with 512 process runs of Chombo after swapping out math libraries. X-axis is the MPI rank identifier.

D. 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.

a) Run environment: We ran MILC at scales of 32 to 256 processes on the NERSC Cori KNL partition.

b) PRQ: 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.

c) UMQ: Average UMQ comparisons per match were similar to the PRQ statistics, and are displayed in Figure 9.

d) Wildcard Usage: MILC utilizes wildcards and the average match attempts per message in the wild-bin ranged between 9 and 14.

IV. 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

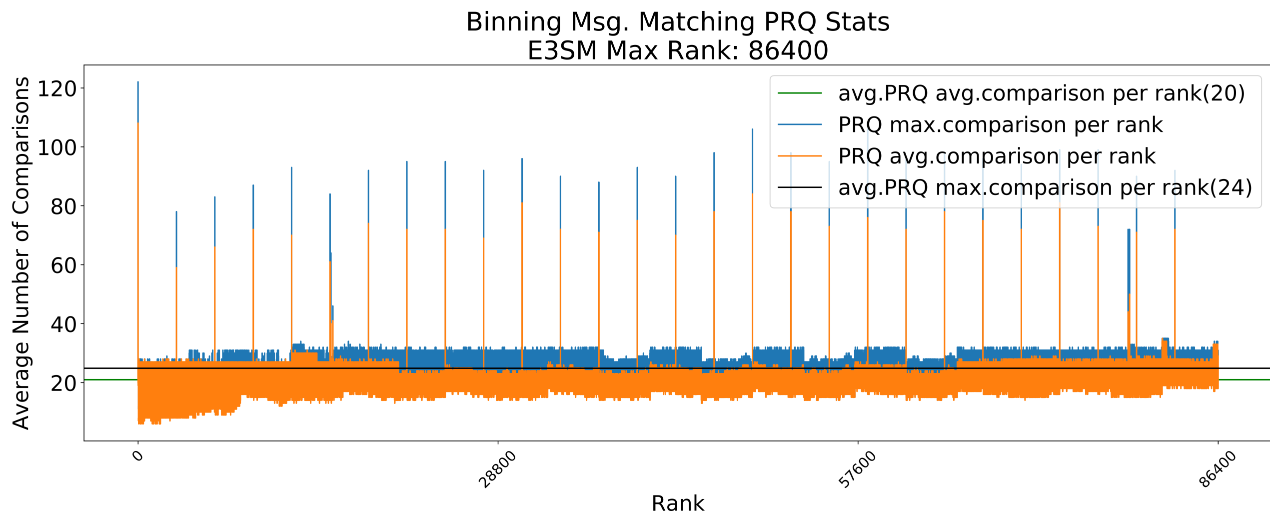

Fig. 8: Average match attempts per message across all ranks (x-axis) for the Posted Receive Queue while running E3SM.

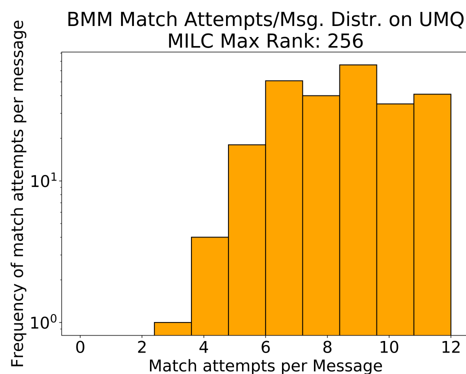

Fig. 9: Histogram of average attempts per message match for the UQM while running MILC on 256 nodes

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.

References

[1] I. Laguna, R. Marshall, K. Mohror, M. Ruefenacht, A. Skjellum, and N. Sultana, "A large-scale study of mpi usage in open-source hpc applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–14.

[2] K. D. Underwood and R. Brightwell, "The impact of MPI queue usage on message latency," *International Conference on Parallel Processing (ICPP)*, pp. 152–160, 2004.

[3] K. D. Underwood, K. S. Hemmert, A. Rodrigues, R. Murphy, and R. Brightwell, "A hardware acceleration unit for mpi queue processing," *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 10–pp, 2005.

[4] M. Flajslik, J. Dinan, and K. D. Underwood, "Mitigating MPI message matching misery," *International Conference on High Performance Computing*, pp. 281–299, 2016.

[5] J. A. Zounmevo and A. Afsahi, "A fast and resource-conscious mpi message queue mechanism for large-scale jobs," *Future Generation Computer Systems*, vol. 30, pp. 265–290, 2014.

[6] K. S. Hemmert, K. D. Underwood, and A. Rodrigues, "An architecture to perform NIC based MPI matching," pp. 211–221, 2007.

[7] K. Ferreira, R. E. Grant, M. J. Levenhagen, S. Levy, and T. Groves, "Hardware mpi message matching: Insights into mpi matching behavior to inform design," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 3, p. e5150, 2020.

[8] "Understanding mpi tag matching and rendezvous offloads (connectx-5)," https://conununity.mellanox.com/docs/DOC-2583, 2020.

[9] S. Derradji, T. Palfer-Sollier, J.-P. Panziera, A. Poudes, and F. W. Atos, "The bxi interconnect architecture," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 2015, pp. 18–25.

[10] W. Zhang, A. Almgren, V. Beckner, J. Bell, J. Blaschke, C. Chan, M. Day, B. Friesen, K. Gott, D. Graves *et al.*, "Amrex: a framework for block-structured adaptive mesh refinement," *Journal of Open Source Software*, vol. 4, no. 37, pp. 1370–1370, 2019.

[11] P. Colella, D. T. Graves, T. Ligocki, D. Martin, D. Modiano, D. Serafini, and B. Van Straalen, "Chombo software package for amr applications design document," *Available at the Chombo website: http://seesar. lbl. gov/ANAG/chombo/(September 2008)*, 2009.

[12] "E3sm," https://github.com/E3SM-Project/E3SM, 2020.

[13] "Milc," https://github.com/milc-qcd, 2020.