SAND2017-7378C

Characterizing MPI Matching via Trace-based Simulation

Kurt B. Ferreira, Scott Levy, Kevin Pedretti and Ryan Grant*
Center for Computing Research
Sandia National Laboratories
[kbferre,slevy,ktpedre,regrant]@sandia.gov

ABSTRACT

With the increased scale expected on future leadership-class systems, detailed information about the resource usage and performance of MPI message matching provides important insights into how to maintain application performance on next-generation systems. However, obtaining MPI message matching performance data is often not possible without significant effort. A common approach is to instrument an MPI implementation to collect relevant statistics. While this approach can provide important data, collecting matching data at runtime perturbs the application's execution, including its matching performance. In this paper, we introduce a trace-based simulation approach to obtain detailed MPI message matching performance data for MPI applications without perturbing their execution. Using a number of key parallel workloads, we demonstrate that this simulator approach can rapidly and accurately characterize matching matching behavior. Specifically, we use our simulator to collect several important statistics about the operation of the MPI posted and unexpected queues. For example, we present data about search lengths and the duration that messages spend in the queues waiting to be matched. Data gathered using this simulation-based approach have significant potential to aid hardware designers in determining resource allocation for MPI matching functions and provide application and middleware developers with insight into the scalability issues associated with MPI message matching.

ACM Reference format:
Kurt B. Ferreira, Scott Levy, Kevin Pedretti and Ryan Grant. 2017. Characterizing MPI Matching via Trace-based Simulation. In *Proceedings of EuroMPI/USA, Chicago, Illinois USA, September 2017 (EuroMPI'17),* 10 pages.
DOI: 10.475/123_4

1 INTRODUCTION

With the increased scale expected on future leadership-class systems, understanding the resource usage and performance of MPI message matching is critical to to the development of MPI implementations. A detailed understanding of current match queue performance informs the design of algorithms and data structures that will be used to implement matching internally on future systems (e.g., whether to implement match queues as a linear queue or a hash table [11]). MPI hardware designers require match queue data to determine how to allocate hardware resources to the message matching task in order to balance performance benefit and cost. Finally, application developers may use this data to identify scalability bottlenecks due to MPI message matching overhead.

MPI [12] message matching is performed on a tuple containing: (1.) *communicator* , a communication context; (2.) message *source ID*, the MPI rank ID of the sending process within the communicator's context; and (3.) message *tag*, a message ID created by the application. When the application performs a receive operation, the source and/or the tag of the request can be wildcarded. A wildcard source (`MPI_ANY_SOURCE()`) indicates that the receive operation will match to incoming messages from any source ID. Similarly, a receive operation with a wildcard tag value (`MPI_ANY_TAG()`) will match to any tag value in the incoming message. When a message is received, the MPI library searches the posted receive queue (PQ), a list of receives the application has posted (through for example, `MPI_Irecv()`) that is yet to be received. If no matching receive operation is found, this message is appended to the unexpected message queue (UQ). When a receive operation is posted by the application, the MPI library must first search the UQ to see if a matching message was already received. If no entry is found on the UQ, the receive descriptor is appended to the PQ, awaiting a future match. MPI matching semantics guarantees that messages sent between process-pairs are matched in program order.

While matching information is useful, it is often difficult to obtain. MPI matching behavior is typically not tracked or exposed by current MPI implementations. To obtain this information, developers typically must instrument the MPI implementation and export the statistics of interest. While the required instrumentation is straightforward, it is time consuming and adds overhead which can potentially impact application execution, possibly affecting the quality of the information obtained.

In this work, we introduce a trace-based simulation approach to obtain MPI matching information without the pitfalls of this implementation-based approach. This validated simulation toolkit utilizes application traces to characterize the MPI matching behavior. This approach has a number of advantages. First, it is possible to gather more detailed information in the simulator since there is no impact on application performance (i.e., the matching instrumentation is performed outside of *simulated time*). Second, results can typically be gathered more rapidly and at larger scales than with the empirical approach. There is no need to instrument an MPI implementation and gathering a trace for this simulator is a well-known process. Many of these traces are already freely available to the community as a whole. Finally, the simulator provides the capability to change coarse-grained system behavior, such as

*Sandia National Laboratories is a multi mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

EuroMPI'17, Chicago, Illinois USA

DOI: 10.475/123_4

slow network links, slow compute nodes, and different hardware configurations than of that was used when the trace was collected. These scenarios have the potential to impact MPI matching behavior and are critical to predicting performance as they typically occur frequently on large-scale HPC systems.

The specific contributions of this paper are:

- We introduce a lightweight and efficient trace-based simulation approach to obtain MPI match queue information (§2).
- Using this validated simulator, we characterize the match length performance for the posted and unexpected queues for a number of key high-performance computing workloads. This characterization includes the cummulative distribution function, the mean values, and the maximimum search traversals and queue sizes (§§3.1–3.4)
- We characterize the time-in-queue for each of these queues (§3.5).
- Finally, we analyze the match queue scaling performance for each of these workloads (§§3.3,3.6).

2 APPROACH

This section describes the experimental framework we used to measure match queue characteristics. We also describe the set of applications used for this study.

2.1 Simulating Application Performance

We use trace-based simulation to study how MPI match queues are affected by application behavior. As stated previously, the principal benefit of using simulation in this context is it allows us to carefully characterize match queue performance without perturbing the performance of the application under study. An alternative approach to characterizing match queue performance is to instrument the MPI library implementation. However, modifying the match engine to capture statistical information has the potential to perturb the observed results (e.g., delays introduced by collecting performance data may change how match queue depths change over time). Simulation allows us to delay the simulation and record detailed information about how MPI messages are matched without affecting the timing of any operations that occurs within the simulation. In addition, our simulation approach can accurately predict application and match queue lengths and times, while using significantly fewer resources and in less time [23] than running on a actual large-scale system. For example, we can simulate a 10 hour LAMMPS SNAP Problem at 1024 nodes in less than 10 minutes. Our simulation-based approach also allows for greater flexibility in perturbing application performance and investigating how these perturbation potentially impact match queue performance.

Our simulation framework is based on `LogGOPSim` [16]. `LogGOPSim` is a trace-based simulator of MPI applications. The extensive validation of `LogGOPSim` has been documented elsewhere [15, 16, 23]. To simulate the execution of an application, a trace is collected during the application's execution. The trace records details about the sequence of MPI operations for each MPI process, including when each operation began and when it completed. The simulator uses the information contained in the trace to reproduce all communication dependencies, including indirect dependencies between processes which do not communicate directly. The temporal cost of communication events are modeled with the the LogGOPS communication model, an extension of the well-known LogP model [6]. `LogGOPSim` is also capable of extrapolating traces from small application runs; a trace collected by running the application with p processes can be extrapolated to simulate performance of the application running with $k \cdot p$ processes. However, none of the data that is presented in this paper was collected using extrapolation.

To accurately simulate MPI application execution, `LogGOPSim` must perform the same set of matching functions (and follow the same rules of matching semantics) as an actual MPI implementation (e.g., MPICH, OpenMPI). To examine how match queue characteristics are affected by application behavior, we instrumented the `LogGOPSim` matching engine to record details about the length of the match queues and about when messages arrive at, and depart from, the match queues. Because MPI message matching is typically very fast, `LogGOPSim` does not currently model the temporal cost of message matching. Instead, it assumes that the application's execution is effectively unaffected by the cost of matching operations. As a result, we can instrument the `LogGOPSim` message matching engine without perturbing the application's execution in any way. In the future, a more realistic model of match engine performance would not likely depend on the performance of the simulator, so we anticipate that this advantage will persist.

As mentioned previously, our simulation framework allows for the potential of investigating match queue performance under performance perturbation scenarios. These perturbations can come from a number of sources: OS noise or *jitter* [9, 15], resilience activities [10], and in situ data analytics [22]. To simulate these perturbations, the simulator has the ability to take away CPU cycles from the application during execution. For these performance variations, `LogGOPSim` accepts an *execution trace* of the perturbation: an ordered list of an execution, expressed as the start time and duration of each of these interrupts.

2.2 Workload Descriptions

In this paper, we examine the results from eight HPC workloads. These workloads, described in Table 1, include two important DOE production applications (LAMMPS and CTH), a proxy application (LULESH) from the Department of Energy's Exascale Co-Design Center for Materials in Extreme Environments (ExMatEx), a proxy application (MCCK) from the Department of Energy's Center for Exascale Simulation of Advanced Reactors (CESAR), a proxy application (AMG2013) used for co-design at Lawrence Livermore National Laboratories, and two mini-applications (HPCCG and miniFE) from Sandia's Mantevo suite. This set of workloads captures a wide range of computation techniques and application behaviors and therefore represents a significant cross-section of highly scalable high-performance applicators which are run on extreme-scale systems.

2.3 Limitations of this study

While we believe this work makes an important contribution in the study of matching for scalable applications, there are a number of limitations that we outline here and must be aware of while drawing conclusions. First, while the application included in this study

Application	Description
LAMMPS	Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS). A classical molecular dynamics simulator from Sandia National Laboratories [19, 25]. The data presented in this paper are from experiments that use the Lennard-Jones (LAMMPS-lj) potential that is included with the LAMMPS distribution, and the SNAP (LAMMPS-snap) potential.
CTH	A multi-material, large deformation, strong shock wave, solid mechanics code [8, 24] developed at Sandia National Laboratories. The data presented in this paper are from experiments that use an input that describes the simulation of the detonation of a conical explosive charge (CTH-st).
LULESH	Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH). A proxy application from the Department of Energy Exascale Co-Design Center for Materials in Extreme Environments (ExMatEx). LULESH approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh [20].
MCCK	Monte Carlo Communication Kernel (MCCK) [5] is a proxy application that approximates the communication of the domain decomposed particle algorithm.
miniFE	A proxy application that captures the key behaviors of unstructured implicit finite element codes [14].
HPCCG	A Mantevo mini-application [21] designed to mimic finite element generation, assembly and solution for an unstructured grid problem.
AMG2013	A parallel algebraic multigrid solver for linear systems arising from problems on unstructured grids [13].

Table 1: Descriptions of the workloads used for evaluating memory compression.

represent a number of key representative HPC workloads, this list is by no means exhaustive. For example, in this work we exclude from the analysis a number of applications with extremely long match queue entries (*c.f.* the Fire Dynamics Study (FDS) in [11]). Also, while the simulator is proven to be highly accurate [15, 16, 23], the simulator may not accurately reflect all possible scenarios. For example, the current version of the simulator does not accurately model network contention. Therefore, in scenarios of high network contention, the simulator may not accurately predict matching performance. Lastly, in this current study we only consider the match queue performance of single-threaded applications. Use of multiple threads per MPI process may increase queue lengths. Analysis of these multi-threaded workloads is beyond the scope of the current work and is slated for future work

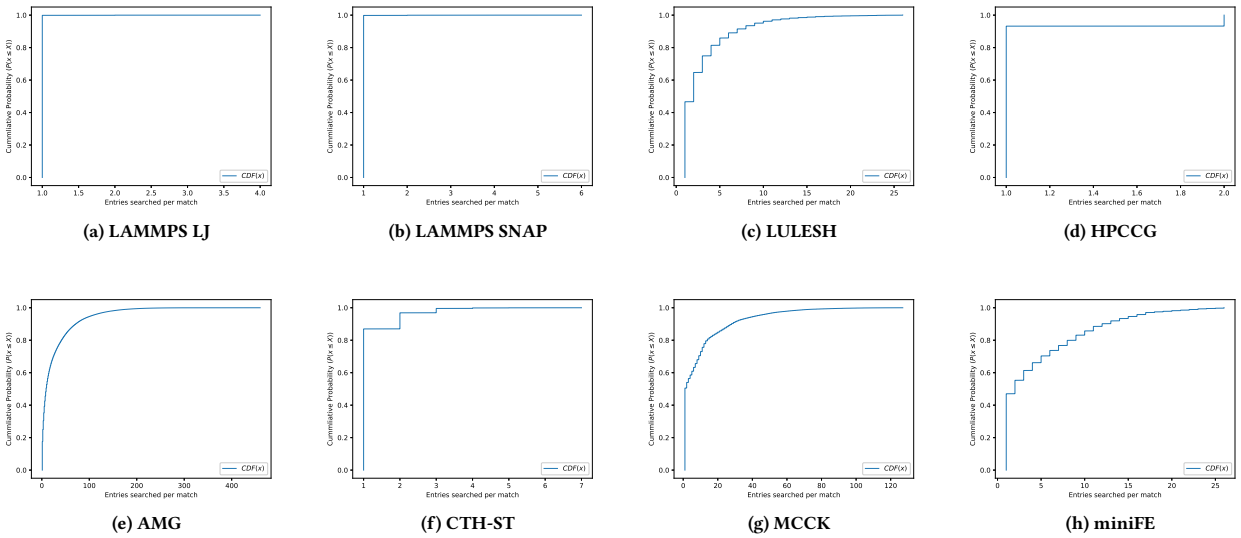
3 RESULTS

In this section, we analyze the match queue results from our simulation framework. Again, we display statistics from the two MPI match queues the posted receive queue (PQ) and the unexpected receive queue (UQ), described previously. These results will focus on two properties of each match operation, the number of entries searched in the queue before a match is found and the amount of time each entry spent in the queue before being matched. To display the queue behavior we will use the cumulative distribution function (CDF) of the population considered. For a value x, the $CDF(x)$ is the probability that a value X will take a value less than or equal to x. We will also use a quartiles plot to represent the distribution of data considered. This can be viewed as compact representation of the cumulative distribution function. The range of the whiskers of the plot show the minimum and maximum values, the extent of the box shows the 25% and 75% quartiles, and a line (i this case red) within the box representing the arithmetic mean. We will also display maximum queue lengths and other summary properties of the matching process. This summary statistics are over all processes within the application. These summary statistics are likely most useful to the MPI system designers as they specify, for example, how many bins might needed for an efficient hash-based match queue implementation [11], while the CDFs are likely most useful to application designers as they specify the efficiency of the match queues in terms of search depths and queue time durations.

3.1 Search length CDFs

Figure 1 shows the CDF for the PQ over all processes in the measured application (128 nodes for all expect LULESH which is 125 processes). The first thing we notice from this figure is that match search lengths can vary dramatically across applications, from less than 5 entries for applications like LAMMPS to well over 400 entries for AMG. Also we see from this figure that for many of the applications the majority of searches only traverse a small portion of the queue. In most cases, half of the searches match in less than 5 entries. This demonstrates that many applications pre-post their receives in an effort to increase performance.

Figure 2 shows the search length CDF for the UQ. Similar to the PQ data, UQ search length sizes can vary dramatically from application to application. When compared to the PQ data, we also see that for those application with longer posted receive queues match lengths, the unexpected receive queue match lengths are also typically longer. The lengths of the UQ in comparison to the



(a) LAMMPS LJ (b) LAMMPS SNAP (c) LULESH (d) HPCCG

(e) AMG (f) CTH-ST (g) MCCK (h) miniFE

Figure 1: Cumulative distribution functions for the search length of the posted receive queue (PQ) over all 128 (125 for lulesh) processes

PQ, however, are not consistent across applications. For some applications (i.e. LAMMPS, miniFE, LULESH, etc) the UQ search length is less that or equal to PQ search length, but for other applications the UQ length can be significantly larger. For example, for AMG the UQ search length is over a factor 2 longer in some cases. These longer UQs can lead to significant slowdowns in the application due to additional memory consumption or increased latency due to the rendezvous protocol typically used to handle long unexpected messages.

3.2 Per-node mean search lengths CDFs

In this section we examine the mean search lengths for each node of the application. For each node the mean match search length across the entire run of the application is calculated. The CDF of those means is displayed. This metric gives us an idea of the average search length for each of the queues. It also will point out if some nodes have significantly longer match lengths that other nodes.

Figures 3 and 4 show the CDF for the PQ and UQ, respectively. From these figures we see that for the majority of the applications, the average match traversal is close to 1, the exception being AMG which has very large traversals in comparison and a significant degree of variation in those traversals across processes.

3.3 Search length scaling

Next, we look at how the traversals change with increase application scale. In these figures we display the quartiles of the distribution. The red line in the figure represents the arithmetic mean of the values. Figures 5 and 6 displays this data for the PQ and UQ, respectively. From the figures we see that for nearly all of the applications the search traversals do increase with scale for both the PQ and UQ. This has significant implications as system sizes continue to increase. While typically not a bottleneck currently, this demonstrates that matching time may limit application scalability going forward and application designers may want to take steps to ensure queues are kept small.

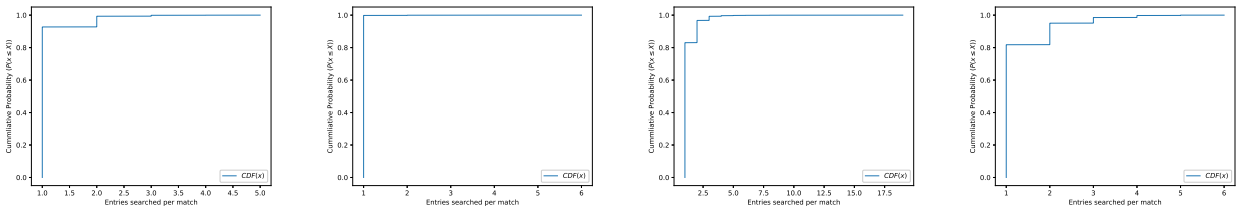
3.4 Match queue statistics

Lastly, we look at some summary statistics of the PQ and UQ traversals. In Table 2, we show the maximum total size of each of the queues through the run of the application ($_{max}||PQ||$ and $_{max}||UQ||$), the longest match in each queue ($_{max}|match(PQ)|$, $_{max}|match(UQ)|$), and the total number of entries matched for each queue ($PQ_{entries\ matched}$ and $UQ_{entries\ matched}$).

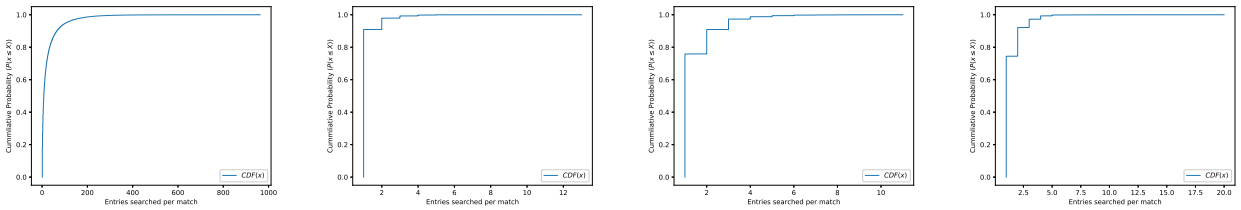
From this table, we see that for all of the applications the vast majority of messages match on the PQ, which typically leads to better performance. Again, we also see from this table that application can vary dramatically in terms of the maximum traversal in the Q and the maximum sizes. Finally, we see that for both the PQ and UQ the maximum queue traversal does not go the end of the list.

3.5 Time in queue CDFs

In the remainder of this section we will focus our attention on the amount of time an entry spends in each of the queues, from the time it is added to a successful match. Figures 7 and 8 show the CDFs for each of the queues. Minimizing time spent in a queue can also help minimize the length of a queue. From this figure we note that for the majority of the applications, the amount of time in a queue is low and long periods of times in the queue are outliers, with the exceptions being LAMMPS and LULESH. For LULESH, the PQ times are bi-model with half of the entries remaining in the queue for around 5 microseconds and the other half for about 15 microseconds. Overall, the lifetime of each entry in the queue has a short lifetime, even for queues that can be quite long (e.g. AMG).

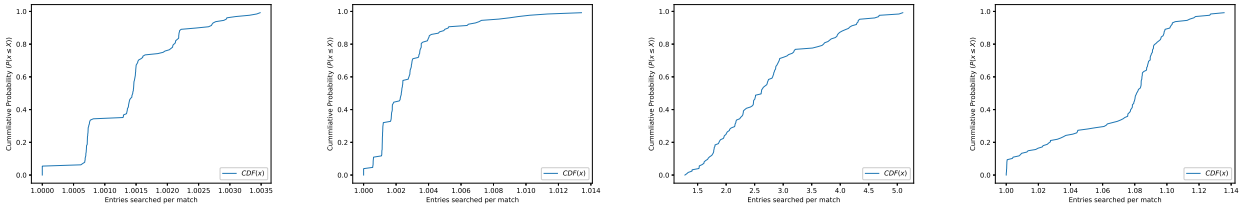


(a) LAMMPS LJ (b) LAMMPS SNAP (c) LULESH (d) HPCCG

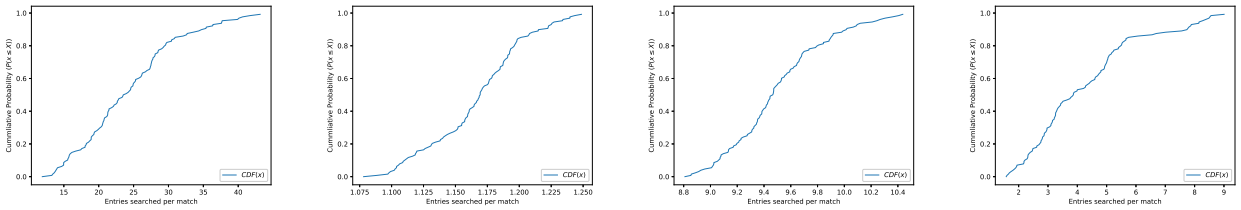


(e) AMG (f) CTH-ST (g) MCCK (h) miniFE

Figure 2: Cumulative distribution functions for the search length of the unexpected receive queue (UQ) over all 128 processes



(a) LAMMPS LJ (b) LAMMPS SNAP (c) LULESH (d) HPCCG



(e) AMG (f) CTH-ST (g) MCCK (h) miniFE

Figure 3: Cumulative distribution function for the per-node mean search length for the posted receive queue (PQ).

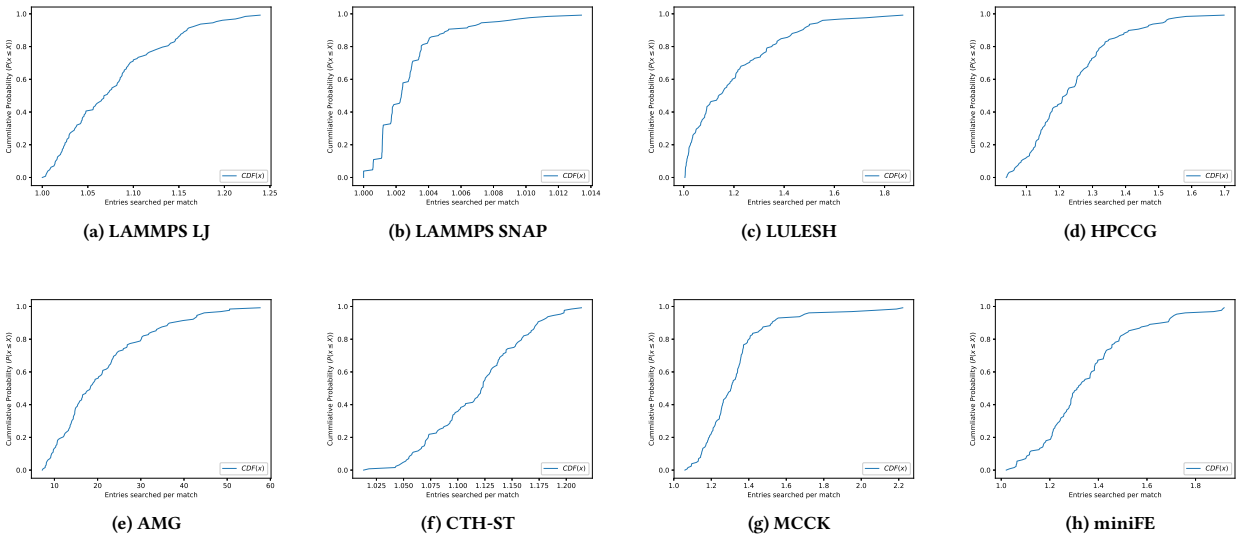
3.6 Queue time scaling

Finally, we look at how the amount of time an entry spends in queue varies as we increase process count for the application. Again, we display quartile plots for each of the tested applications, with the whiskers displaying the minimum and maximum values, the box edges denoting the 25% and 75% quartiles, and the red line denoting the arithmetic mean.

Figures 9 and 10 displays the quartiles data for each of our applications over a number of process counts. From this figure we see that, unlike the search length scaling numbers in Section 3.3, the time spent in the queue decreases as we increase scale for most of the applications. The exception to this decreasing trend being the LAMMPS-snap and AMG workloads, whose intervals of time spent on queue increase with scale.

4 RELATED WORK

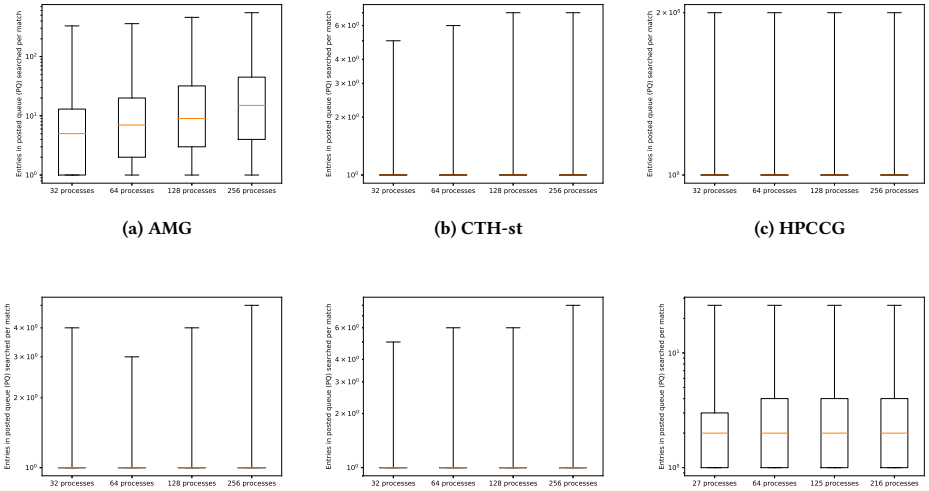
A number of published studies have examined MPI matching performance in HPC systems. Unexpected and posted receive queue lengths have been studied previously [3, 4, 17, 26]. These works demonstrate that unexpected message queue lengths can grow with



(a) LAMMPS LJ (b) LAMMPS SNAP (c) LULESH (d) HPCCG

(e) AMG (f) CTH-ST (g) MCCK (h) miniFE

Figure 4: Cumulative distribution function of the per-node mean search length for the unexpected receive queue (UQ)



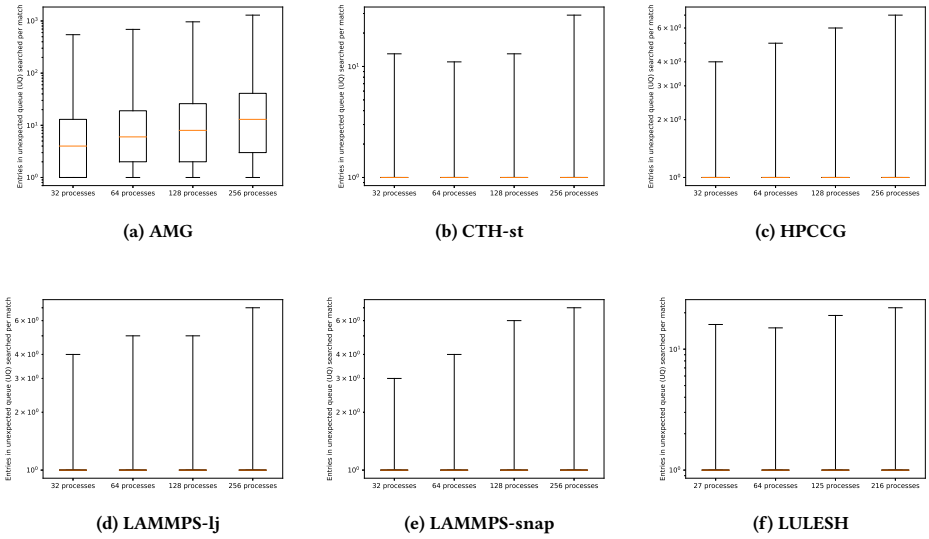
(a) AMG (b) CTH-st (c) HPCCG

(d) LAMMPS-lj (e) LAMMPS-snap (f) LULESH

Figure 5: Quartiles of search length of the posted queue (PQ) for each process count. The red line represents the arithmetic mean

process counts, in one case reaching over 200 entries at 140 processes, and that there can be dramatic variation in queue lengths across nodes, often showing increased queue lengths on an application's rank 0. In addition, these works demonstrate that the posted queue lengths are typically smaller than the unexpected queue for real applications, demonstrating the necessity of analyzing real applications rather than small benchmarks.

Dang et al. [7], examined a number of DOE exascale proxy applications and showed similar matching results. This work demonstrates how MPI tag matching can potentially impact scalability. This work proposes to address this scalability problem by modifying MPI semantics and not allowing wildcards or alternatively having the sender specify when a message will be used in a wildcard match. Similarly, Klenk et al. [18] propose relaxing MPI semantics to allow for efficient message matching on SIMT processors (GPUs).



(a) AMG (b) CTH-st (c) HPCCG

(d) LAMMPS-lj (e) LAMMPS-snap (f) LULESH

Figure 6: Quartiles of search length of the unexpected queue (UQ) for each process count. The red line represents the arithmetic mean

Workload	$max\|\|PQ\|\|$	$max\|\|UQ\|\|$	$max\|match(PQ)\|$	$max\|match(UQ)\|$	$PQ_{entries\ matched}$	$UQ_{entries\ matched}$
LAMMPS LJ	7	9	4	5	179283 (73%)	65647 (27%)
LAMMPS SNAP	8	8	6	6	220008 (74%)	76493 (26%)
CTH-ST	7	22	7	13	5457665 (72%)	2130256 (28%)
LULESH	26	23	26	19	13805773 (90%)	1462396 (10%)
HPCCG	2	6	2	6	1541498 (67%)	776874 (33%)
AMG	505	1357	460	963	2341262 (60%)	1529439 (40%)
MCCK	127	19	127	11	1636809 (94%)	103909 (6%)
miniFE	26	25	26	20	780040 (79%)	203462 (21%)

Table 2: Maximum high-water marks, maximum search depth and total number of successful matches for posted (PQ) and unexpected (UQ) Q's at 128 nodes (125 nodes for LULESH).

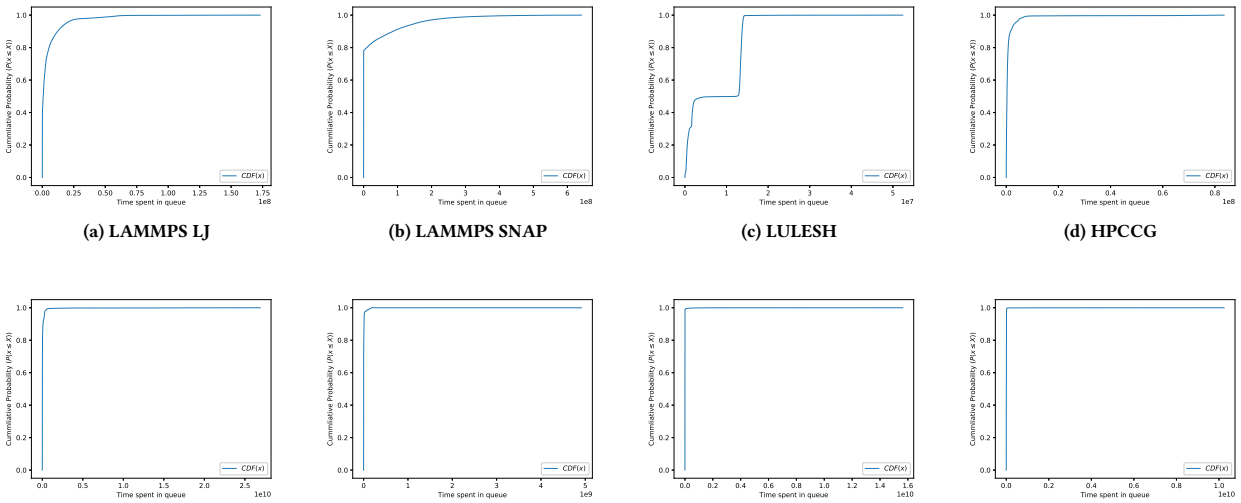
With the expected increase in match queue lengths, a number of works have examined modifications to how matching is done, while keeping current MPI semantics. Flajslik et al. [11] proposed using hashing rater than list traversal to speed up MPI matching. Bayatour et al. [2] proposed a method to dynamically switch between a linked list and a hash table at runtime, The goal of this work is to get the best of both methods: low space overhead and fast first match time for linked lists, constant match time with hashing for long lists.

Finally, performing match operations directly in hardware has also been examined. Brightwell et al. [3] instrumented a hardware offload engine's match list implementation to track MPI tag data, queued times and match list search depths and sizes. Similarly, Underwood et al. [27] examine the requirements of a hardware platform to accelerate MPI matching on these systems.

Our work is different from these existing studies in several important ways. First, our simulation-based approach does not perturb the matching performance of the simulated application like what might be found in an experimental-based approach. Second, our simulation approach can accurately predict application and match queue lengths and times, while using significantly fewer resources and in less time [23] than running on an actual large-scale system. For example, we can simulate a 10 hour LAMMPS run of the SNAP problem at 1024 nodes in less than 10 minutes. Our simulation-based approach also allows for greater flexibility in perturbing the application to investigate how these perturbations impact match queue performance. Perturbations can come from a number of sources: OS noise or *jitter* [9, 15], resilience activities [10], and in situ data analytics [22]. Finally, this work has the potential to enable those who do not have source access to applications to test research solutions as a small and growing list of application traces become available [1].

5 CONCLUSIONS & FUTURE WORK

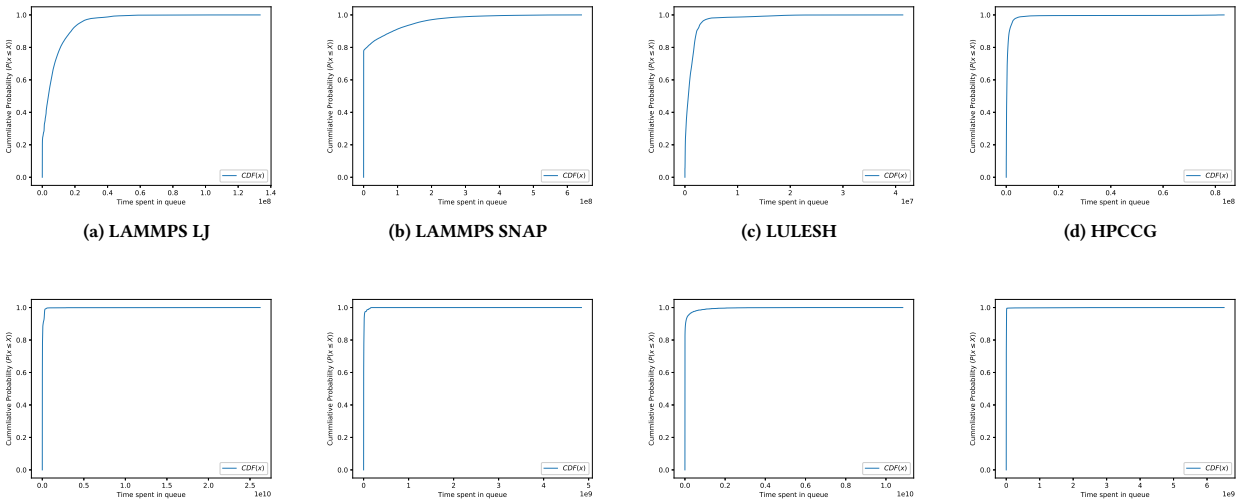
Understanding MPI matching requirements is critical to the scalability of future systems. In this work, we introduced a lightweight,



(a) LAMMPS LJ (b) LAMMPS SNAP (c) LULESH (d) HPCCG

(e) AMG (f) CTH-ST (g) MCCK (h) miniFE

Figure 7: Cumulative distribution functions for the time-in-queue for each entry of the posted receive queue (PQ) over all 128 processes

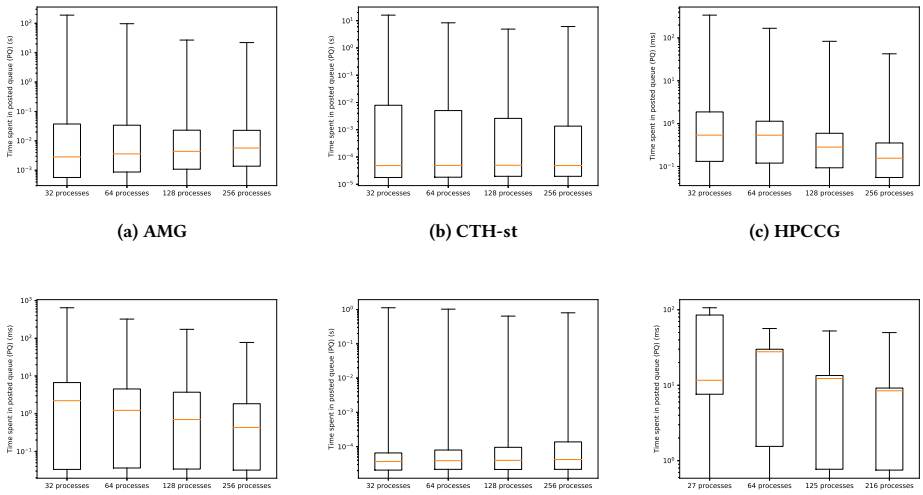


(a) LAMMPS LJ (b) LAMMPS SNAP (c) LULESH (d) HPCCG

(e) AMG (f) CTH-ST (g) MCCK (h) miniFE

Figure 8: Cumulative distribution functions for the time-in-queue for each entry of the unexpected receive queue (UQ) over all 128 processes

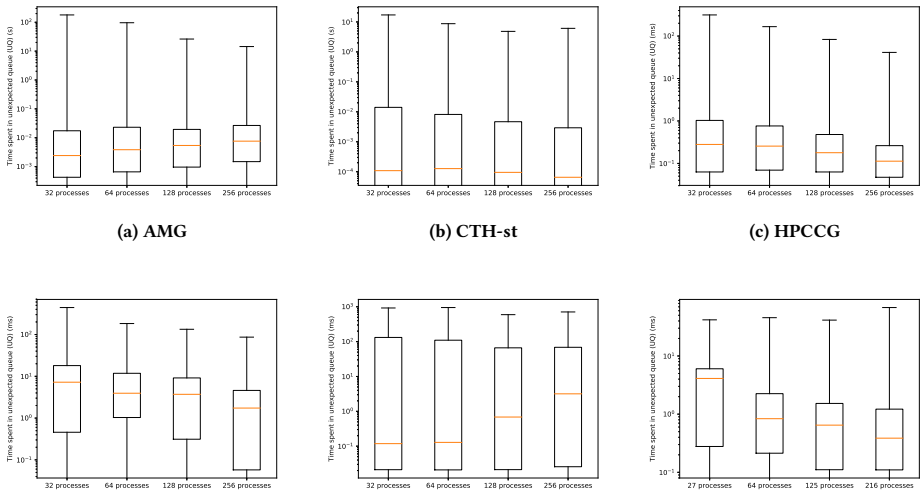
simulation-based approach, based on a previously validated simulation framework to characterize the posted receive and unexpected queues for a number of key HPC workloads. This work made the following significant contributions. First, it showed a number of key characteristics for each of these queues: the cumulative distribution function of both the match traversal and the time spent in queue waiting for a match, the mean and maximum search length, and the maximum list size for both the posted receive and unexpected queues. In addition, we showed how these queues grow in size as the process count for an application increases. Lastly this work showed the duration of time entries spend in each of these queues as well as how that duration changes with increased scale. Data gathered using this simulation-based approach has significant potential in aiding hardware designers in determining resource allocation for MPI matching functions and providing application



(a) AMG (b) CTH-st (c) HPCCG

(d) LAMMPS-lj (e) LAMMPS-snap (f) LULESH

Figure 9: Quartiles of search length of the unexpected queue (PQ) for each process count. The red line represents the arithmetic mean



(a) AMG (b) CTH-st (c) HPCCG

(d) LAMMPS-lj (e) LAMMPS-snap (f) LULESH

Figure 10: Quartiles of search length of the unexpected queue (UQ) for each process count. The red line represents the arithmetic mean

and middleware developers with insight into the scalability issues associated with MPI message matching in their applications.

While we believe this work makes significant contributions, additional work is needed to fully understand matching for future systems. First, we would like to investigate the impact of multithreaded MPI applications on MPI matching performance. Also, we would like to examine the influence that process performance variation has on performance. If we slow down certain nodes, how does that impact queue traversals and the sizes of the respective queues? Finally, we would like to characterize the impacts MPI matching has on performance and incorporate this overhead into our current simulation framework as to better characterize matching influence on performance.

REFERENCES

[1] Trace Repository. http://htor.inf.ethz.ch:8888/. (????). Retrieved 16 Jan 2014.

[2] M. Bayatpour, H. Subramoni, S. Chakraborty, and D. K. Panda. 2016. Adaptive and Dynamic Design for MPI Tag Matching. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*.

[3] R. Brightwell, K. Pedretti, and K. Ferreira. 2008. Instrumentation and Analysis of MPI Queue Times on the SeaStar High-Performance Network. In *Proceedings of 17th International Conference on Computer Communications and Networks*. 1–7.

[4] R. Brightwell and K. D. Underwood. 2004. An analysis of NIC resource usage for offloading MPI. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* 183–.

[5] Argonne National Laboratory CESAR. The CESAR Proxy-apps. https://cesar.mcs.anl.gov/content/software. (????). Retrieved 10 June 2013.

[6] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. 1993. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '93)*. ACM, New York, NY, USA, 1–12.

[7] Hoang-Vu Dang, Marc Snir, and William Gropp. 2016. Towards Millions of Communicating Threads. In *Proceedings of the 23rd European MPI Users' Group Meeting (EuroMPI 2016)*. ACM, New York, NY, USA, 1–14.

[8] Jr. E. S. Hertel, R. L. Bell, M. G. Elrick, A. V. Farnsworth, G. I. Kerley, J. M. McGlaun, S. V. PetneY, S. A. Silling, P. A. Taylor, and L. Yarrington. 1993. CTH: A Software Family for Multi-Dimensional Shock Physics Analysis. In *Proceedings of the 19th Intl. Symp. on Shock Waves*. 377–382.

[9] Kurt B. Ferreira, Ron Brightwell, and Patrick G. Bridges. 2008. Characterizing Application Sensitivity to OS Interference Using Kernel-Level Noise Injection. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC'08)*.

[10] Kurt B. Ferreira, Patrick Widener, Scott Levy, Dorian Arnold, and Torsten Hoefler. 2014. Understanding the Effects of Communication and Coordination on Checkpointing at Scale. In *Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage and Analysis (Supercomputing)*.

[11] Mario Flajslik, James Dinan, and Keith D. Underwood. 2016. *Mitigating MPI Message Matching Misery*. Springer International Publishing, Cham, 281–299.

[12] Message Passing Interface Forum. 2012. MPI: A Message-Passing Interface Standard Version 3.0. (09 2012). Chapter author for Collective Communication, Process Topologies, and One Sided Communications.

[13] V.E. Henson and U.M. Yang. 2002. BoomerAMG: A parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics* 41, 1 (2002), 155–177.

[14] Michael A. Heroux, Douglas W. Doerfler, Paul S. Crozier, James M. Willenbring, H. Carter Edwards, Alan Williams, Mahesh Rajan, Eric R. Keiter, Heidi K. Thornquist, and Robert W. Numrich. 2009. *Improving Performance via Mini-applications*. Technical Report SAND2009-5574. Sandia National Laboratory.

[15] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. 2010. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*.

[16] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. 2010. LogGOPSim - Simulating Large-Scale Applications in the LogGOPS Model. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM, 597–604.

[17] Rainer Keller and Richard L. Graham. 2010. Characteristics of the Unexpected Message Queue of MPI Applications. In *Proceedings of the 17th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface (EuroMPI'10)*. Springer-Verlag, Berlin, Heidelberg, 179–188.

[18] Benjamin Klenk, Holger Fröning, Hans Eberle, and Larry Dennison. 2017. Relaxations for High-Performance Message Passing on Massively Parallel SIMT Processors. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS), Orlando, FL*.

[19] Sandia National Laboratories. 2013. LAMMPS Molecular Dynamics Simulator. http://lammps.sandia.gov. (Apr. 10 2013).

[20] Lawrence Livermore National Laboratory. Co-design at Lawrence Livermore National Lab : Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH). http://codesign.llnl.gov/lulesh.php. (????).

[21] Sandia National Laboratory. 2014. Mantevo Project Home Page. http://mantevo.org. (Jan. 10 2014).

[22] Scott Levy, Kurt B. Ferreira, Patrick M. Widener, Patrick G. Bridges, and Oscar H. Mondragon. 2016. How I Learned to Stop Worrying and Love In Situ Analytics: Leveraging Latent Synchronization in MPI Collective Algorithms. In *Proceedings of the 23rd European MPI Users' Group Meeting, EuroMPI 2016, Edinburgh, United Kingdom, September 25-28, 2016*. 140–153.

[23] Scott Levy, Bryan Topp, Kurt B Ferreira, Dorian Arnold, Torsten Hoefler, and Patrick Widener. 2014. Using simulation to evaluate the performance of resilience strategies at scale. In *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation*. Springer, 91–114.

[24] J.M. McGlaun, S.L. Thompson, and M.G. Elrick. 1990. CTH: A three-dimensional shock wave physics code. *International Journal of Impact Engineering* 10, 1 (1990), 351–360.

[25] Steve Plimpton. 1995. Fast Parallel Algorithms For Short-Range Molecular-Dynamics. *Journal of Computational Physics* 117, 1 (1995), 1–19.

[26] Keith Underwood, Sue Goudy, and Ron Brightwell. 2005. A Preliminary Analysis of the MPI Queue Characteristics of Several Applications. *2013 42nd International Conference on Parallel Processing* 00 (2005), 175–183.

[27] K. D. Underwood, K. S. Hemmert, A. Rodrigues, R. Murphy, and R. Brightwell. 2005. A Hardware Acceleration Unit for MPI Queue Processing. In *19th IEEE International Parallel and Distributed Processing Symposium*.