Using simulation to examine the effect of MPI message matching costs on application performance

Scott Levy*, Kurt B. Ferreira, Whit Schonbein, Ryan E. Grant, Matthew G.F. Dosanjh

Center for Computing Research, Sandia National Laboratories, Albuquerque, NM United States

ARTICLE INFO

Article history:
Received 23 October 2018
Revised 4 January 2019
Accepted 26 February 2019
Available online 27 February 2019

Keywords:
Message passing interface
Message processing
Performance
Multithreading

ABSTRACT

Attaining high performance with MPI applications requires efficient message matching to minimize message processing overheads and the latency these overheads introduce into application communication. In this paper, we use a validated simulation-based approach to examine the relationship between MPI message matching performance and application time-to-solution. Specifically, we examine how the performance of several important HPC workloads is affected by the time required for matching. Our analysis yields several important contributions: (i) the performance of current workloads is unlikely to be significantly affected by MPI matching unless match queue operations get much slower or match queues get much longer; (ii) match queue designs that provide sublinear performance as a function of queue length are unlikely to yield much benefit unless match queue lengths increase dramatically; and (iii) we provide guidance on how long the mean time per match attempt may be without significantly affecting application performance. The results and analysis in this paper provide valuable guidance on the design and development of MPI message match queues.

© 2019 Published by Elsevier B.V.

1. Introduction

MPI [1] is the dominant programming model on leadership-class HPC systems. MPI has flourished in part because its semantics make it easy for application programmers to reason about communication on large and complex HPC systems. MPI message matching semantics ensure that, except as directed by the application programmer, messages are processed in a deterministic order. The mechanisms for introducing non-determinism include wildcards (e.g., MPI_ANY_SOURCE, MPI_ANY_TAG) and specific operations that allow for non-determinism (e.g., MPI_Waitany()). Enforcing deterministic message processing requires that receive requests are carefully compared to messages that have arrived and are awaiting processing. Similarly, incoming messages must be compared to pending receive requests. These rules are typically enforced using two MPI-internal match queues: one for pending receive requests and one for incoming messages that do not match any existing request.

MPI’s prominence is also due to the fact that it provides high-performance data exchange on a wide-range of HPC platforms. Attaining high performance with MPI requires very efficient message matching to minimize overheads and the latency these overheads introduce into application communication. As a result, significant research efforts have investigated mechanisms for accelerating MPI message matching. These approaches include: specialized hardware [2,3], exploiting application communication patterns [4], and software hash tables [5]. Others have proposed to improve matching performance by relaxing MPI match semantics [6].

In this paper, we use a validated simulation-based approach to examine how workload time-to-solution is affected by match queue performance. Specifically, we examine how several important HPC workloads respond to changes in MPI matching speed. We also consider the circumstances under which MPI message matching performance may significantly degrade application performance.

We provide a detailed analysis of the relationship between the MPI message matching performance and workload time-to-solution. Specifically, we make the following contributions:

• Current workloads are unlikely to be significantly affected by the cost of MPI message matching unless matching gets much slower than current mechanisms or match queues get much longer than observed in current applications.
• Advances in match queue design that provide sublinear performance as a function of queue length are unlikely to provide significant performance benefit unless match queue lengths increase dramatically.

* Corresponding author.
E-mail addresses: slevy@sandia.gov (S. Levy), kferreira@sandia.gov (K.B. Ferreira), wschon@sandia.gov (W. Schonbein), regrant@sandia.gov (R.E. Grant), mdsanj@sandia.gov (M.G.F. Dosanjh).

https://doi.org/10.1016/j.parco.2019.02.008
0167-8191/© 2019 Published by Elsevier B.V.
• Although multithreaded MPI has the potential to significantly increase message matching costs, current forecasts suggest that if locking costs can be managed, the impact of message matching on execution time is likely to remain modest.
• In the event that match queue performance degrades in the future, we provide guidance on how long the mean time per match attempt may be without significantly affecting application performance.

Simulation allows us to carefully examine the relationship between MPI message matching performance and application time-to-solution. The data and analysis in this paper provide valuable guidance on the design and development of MPI message match queues.

2. Background

Most MPI point-to-point communication operations fall into one of two categories: one-sided (e.g., MPI_Put() and MPI_Get()) and two-sided operations (e.g., MPI_Send() and MPI_Recv()). Message matching enforces determinism and in-order delivery of messages for two-sided operations. Message matching is unnecessary for one-sided operations.

Message matching is the process of matching incoming messages with local receive requests. Messages are matched to requests (and vice-versa) based on their characteristics, including the source of the message, the communicator on which the message is sent, and the tag value associated with the message. To result in a successful match, the characteristics of the message must satisfy the characteristics specified in the receive request. However, exact matches are not always required; receive requests may include wildcards that will match against messages from any source (MPI_ANY_SOURCE) or with any tag value (MPI_ANY_TAG). Additionally, MPI’s no overruling rule requires that messages and receive requests must be matched in the order in which they were posted between any two communicating hosts.

MPI Message matching is commonly implemented using two queues: a Posted Receive Queue (RQ) for unsatisfied receive requests, and an Unexpected Messages Queue (UQ) for messages that have not yet been matched to a receive request. When an application posts a receive request, it is first compared to all of the messages that are currently waiting in the UQ to determine whether an existing message will satisfy the request. If no message in the UQ satisfies the request, then the receive request is appended to the RQ. Similarly, when an incoming message arrives, it is compared to all of the receive requests that are currently in the RQ. If the message satisfies none of the requests in the RQ, then the messages are appended to the UQ.

3. Methodology

3.1. Matching cost models

The cost of matching MPI messages will depend on the underlying data structures that an MPI implementation uses. In this paper, we examine four models for matching cost: linear, superlinear, logarithmic, and fixed-cost. These models are discussed in detail below and their characteristics are summarized in Table 1.

All of our models are built on the cost of performing a queue operation. A queue operation is the process by which an MPI runtime manipulates match queue entries. For three of our models (linear, superlinear, and logarithmic) a queue operation is the act of comparing a single entry in a match queue with the metadata associated with an incoming message or a receive request. For the fixed-cost model, a queue operation is the act of inserting a new entry after an unsuccessful match attempt.

![Table 1](https://example.com/table1.png)

<table>
<thead>
<tr>
<th>Match cost model</th>
<th>Successful match (Hit Cost)</th>
<th>Unsuccessful match (Miss Cost)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>(\ell \times t)</td>
<td>(n \times t + C)</td>
</tr>
<tr>
<td>Superlinear</td>
<td>((0.1t + (1 - 0.1))) (\times t)</td>
<td>((n \times (0.1n + (1 - 0.1)) + C)</td>
</tr>
<tr>
<td>Logarithmic</td>
<td>([\ell \log_2(i) + 1] \times t)</td>
<td>([([\ell \log_2(n) + 1] + ([\log_2(i) + 1]) \times t)</td>
</tr>
<tr>
<td>Fixed-cost</td>
<td>(\ell + C)</td>
<td>(\ell + C)</td>
</tr>
</tbody>
</table>

3.1.1. Linear

The simplest approach to matching is to use a linked list for each of the queues. The simplicity and determinism of this approach means that it is straightforward to enforce MPI’s no overruling rule. This is the approach used by MPICH [5]. In this case, searching all of the entries in a queue is a \(O(n)\) operation, where \(n\) is the length of the queue. A successful match is a \(O(\ell)\) operation, where \(\ell\) is position in the queue where the successful match occurred. In our model, the time to search a queue is equal to the product of the number entries searched and the per-entry search time. Inserting an element into a linked list can typically be accomplished by updating two pointers. Therefore, the time required to insert an element into a queue is a \(O(1)\) operation. For the purposes of our model, we assume that the time required to insert an element is 10 ns.

3.1.2. Superlinear

On some processors, the cost of traversing a linked-list may not be a linear function of the number of entries that have been accessed. For example, Barrett et al. [7] showed that for processors built from simpler cores (e.g., ARM Cortex-A9) the per-entry cost of traversing a linked-list is itself a linear function of the number of entries traversed. Our superlinear model is based on the data presented in this paper, but it is slightly more pessimistic (i.e., the per-entry cost in our model grows faster than the data presented by Barrett et al.).

3.1.3. Logarithmic

The search cost of a match queue can be reduced by implementing the queues as trees. Because of wildcards, care is required to ensure that MPI’s no overruling rule is not broken. This is the approach used by Open MPI [8]. Open MPI uses a set of data structures that is essentially a three-level tree of linked lists. The leaves of the tree are linked lists of all of the messages from the same communicator and source rank. As a result, the cost per match attempt is unlikely to be logarithmic when the total number of match entries in a given queue is large. Additional code is necessary to ensure that messages are properly matched to receive requests that contain wildcards. If messages are evenly distributed between multiple communicators and arrive from many different source ranks, this data structure can significantly reduce the number of elements that must be searched. However, in the worst case (all messages arrive on a single communicator from a single source

---

1 The slope of the ARM Cortex-A9 per-entry cost curve presented by Barrett et al. is approximately 0.078; the slope of the per-entry cost curve in our model is 0.1.
2 This assumes that a given workload uses a small number of communicators, which is true of most current workloads. In principle, a workload could potentially obtain constant matching costs if it used a large number of communicators, although that approach may have other, undesirable consequences.
rank), traversing the additional layers of the tree provides no benefit (in fact it results in a small increase in the search cost).

For our logarithmic-cost model, we assume the match queues are balanced binary trees and we model the time required to search a match queue as the base-2 logarithm of the number of elements that would have to be searched in the linear model. In practice, it may not always be possible to implement a match queue in this way, but this model allows us to approximate the match cost of a binary tree without needing to resolve all of the implementation details. Moreover, our objective is to understand how time spent matching messages may impact application performance, not to faithfully simulate one current MPI implementation. Additionally, a balanced binary tree has more predictable performance characteristics and is easier to reason about. In this case, a complete search of a match queue is a $O(\log_2 n)$ operation, where $n$ is the length of the queue. The time required for a successful match is a $O(\log_2 \ell)$ operation, where $\ell$ is the position where the match would have occurred in the equivalent linked-list implementation. Inserting an element into such a match queue is also a $O(\log_2 n)$ operation.

3.1.4. Fixed-cost

Message matching can also be implemented with hash tables [5] or with specialized hardware (e.g., a ternary content addressable memory (TCAM) [3]). One of the key advantages of these approaches is that they enable very fast ($O(1)$) searches. While the expected value of the cost of entry insertion is also constant, it may be much larger than the search cost. For example, a dynamically-sized hash table requires all of the elements to be rehashed (an $O(n)$ operation) when the size of the hash table grows. Similarly, in the worst case, inserting an element into a TCAM may also be a $O(n)$ operation [9].

In our fixed-cost cost model, we model the time required to perform a search as a small constant value (100ns). We model the time required to insert an element into a match queue as the cost of a queue operation.

3.2. Queue length multiplier

To explore the impact of match queue length on application performance, we introduced a queue length multiplier into our simulation framework. Applying a queue length multiplier of $m$ has the effect of increasing the overall length of a match queue by a factor of $m$. This is accomplished in a way that is functionally equivalent to inserting $m-1$ entries before each element of the match queue. As a result, in addition to the queue being $m$ times longer, every search of the queue is also $m$ times longer. Varying the value of the queue length multiplier allows us to approximate the impact of growth in match queue length on application performance.

3.3. Multithreading

Multithreaded MPI has the potential to increase matching costs. If individual threads send messages independently, the total number of messages that need to be matched may increase. For example, Schonbein et al. [10] demonstrated that simple multithreading has the potential to significantly increase match queue lengths. Moreover, because threads make progress independently, programmers may no longer be able to guarantee that receive requests are posted and messages are sent in a predictable way. As a result, the techniques that programmers use to ensure short match queue searches may no longer be effective in multithreaded MPI programs.

Fig. 1 compares the communication of a single-threaded (Fig. 1a) and a (Fig. 1b) 5-point, two-dimensional stencil using simple multithreading. In Fig. 1a, each square represents a single node running a single MPI process and the arrows represent stencil communication. In the example shown in Fig. 1b, each node is running a single process with 16 threads, each thread is represented by a smaller square (labelled $p_x$, $r_x$, $e_x$, $s_x$, and $w_x$). Threads that communicate with threads on other nodes are shaded gray. Threads that only communicate with threads on the same node are shaded white. For the purposes of this example, we assume that on-node communication takes place via shared memory and does not affect message matching queues. In this simple example, the threads are arranged in a grid and the domain owned by the containing process is divided equally among each of the threads.

To understand the impact that multithreading may have on application performance, we extended the microbenchmark used by Schonbein et al. [10]. The original benchmark used two MPI processes to emulate a bulk-synchronous parallel (BSP) stencil computation. For the experiments in this paper, we have extended the benchmark to allow an arbitrary number of processes to emulate in bidirectional multithreaded communication. The resulting mi-

---

3 Schonbein et al. consider naive multithreading, in which every thread posts MPI messages regardless of destination, and a more nuanced approach, in which threads on a single node communicate without involving the MPI matching engine. In this paper, we examine the performance impact of the latter approach, which we refer to as simple multithreading.
crobenchmark allows us to examine how matching costs may affect the performance of several common stencil communication patterns across multiple processes concurrently. The benchmark we used for the data in this paper conducts 1,000 iterations, where each such iteration consists of stencil communication followed by a MPI_Barrier() and a configurable sleep. For all of the data in this paper, we used a sleep of 100 ms.\textsuperscript{4}

3.4. Default values

For the purposes of the data presented in Section 4, we used a set of default values, summarized in Table 2. Unless otherwise specified, these are the values that we used for all of the experiments that form the basis of the data presented in this paper. We used a linear cost model as the default because it is easy to reason about and it most closely matches the approaches that are commonly used in MPI implementations. We selected 100 ns as the default per-entry search time. Our selection is pessimistic but consistent with existing empirical evidence, see [4,5,7]. For example, Barrett et al. found for that for modern multicore processors (e.g., Intel Core i7 (Sandy Bridge) and Xeon (Haswell), IBM POWER7) searching an entry of the RQ requires less than 10 ns even when the queue is quite long (>1000 entries). Even for processors built on simpler cores (e.g., Intel Xeon Phi, ARM Cortex-A9), the maximum observed search time per entry was approximately 90 ns.

3.5. Simulation framework

We use trace-based simulation to understand how the cost of message matching affects workload performance. Simulation gives us fine-grained control over the cost of matching messages and allows for straightforward experimentation with different match queue models. It also allows us to maintain detailed statistics without perturbing the workload under test.

Our simulation framework is based on LogGOPSim,\textsuperscript{21} a trace-based simulator of MPI applications. The extensive validation of LogGOPSim has been documented elsewhere\textsuperscript{21–24}. To simulate an application, a trace is collected during its 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 information contained in the trace allows LogGOPSim to account for all inter-process communication dependencies, including indirect dependencies. The temporal cost of sending messages between processes is modeled with the LogGOPS communication model, an extension of the well-known LogP model\textsuperscript{25}. The LogGOPS parameters we used for our experiments are based on measurements taken on a Cray XC40 system. The exact values are available in our earlier work\textsuperscript{24}.

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). We have extended LogGOPSim to generate match queue statistics and validated that its match queue exhibits correct behavior\textsuperscript{24}. LogGOPSim uses one set of match queues (a RQ and a PQ) per MPI process. Each queue contains entries that correspond to point-to-point communication as well as the point-to-point messages that result from decomposing the process’s collective operations.\textsuperscript{5}

LogGOPSim also provides functionality for extrapolating (weak-scaling) application traces such that the execution of a trace collected from \( n \) processes can be simulated on \( k \times n \) processes, for integer values of \( k \). Collective operations are scaled exactly; the underlying collective algorithm is executed by \( k \times n \) processes instead of \( n \) processes. Point-to-point communication is approximated. The point-to-point operations in the original trace are replicated for each group of \( n \) processes. For example, if the trace contains an MPI_Send() from rank \( 0 \) to rank \( 1 \) in the original trace, the extrapolated trace will include an MPI_Send() from rank \( (0 + j \times n) \) to rank \( (1 + j \times n) \) for all \( 1 \leq j \leq k \).

The cost of each MPI match attempt (searching one queue and, in the case of an unsuccessful match, inserting into the other) is calculated based on the match cost model that is being used. Our modifications to LogGOPSim assume that all matching operations occur in software: the time required for message matching is assessed against the simulated process’s CPU. We do not currently account for low-level hardware behavior that may, in some cases, affect MPI message matching performance (e.g., the effect of cache performance, cf.\textsuperscript{[26]}).

LogGOPSim does not currently support simulation of multithreaded execution. However, for the purposes of the experiments in this paper, we emulate the impact of multithreaded execution on MPI match queue behavior by assigning unique tags to each thread and by randomizing the message exchange in the execution trace, see Section 4.6. We do not currently model the impact of resource contention, including locking internal data structures to prevent concurrent access by multiple threads. We believe that this approach is reasonable because the efficient use of multithreaded MPI requires that the costs of resource contention be kept low. Moreover, significant research has been devoted to minimizing resource contention in the MPI library, see e.g.,\textsuperscript{[27]}.

3.6. Workloads

In this paper, we examine results collected from six HPC workloads. These workloads, described in Table 3, include important DOE production applications (LAMMPS and CTH-st), a proxy application (LULESH) from the Department of Energy’s Exascale Co-Design Center for Materials in Extreme Environments (ExMatEx), an application for studying quantum chromodynamics (MILC), a mini-application (minife) from Sandia’s Mantevo suite, and a benchmark (HPCG) that captures the computational and data access patterns of a range of important applications. This is a diverse set of workloads that captures a wide range of computational methods and behaviors. It also represents an important cross-section of the scalable, high-performance applications that are run on current extreme-scale systems and are expected to be run on future systems.

4. Experimental results

In this section, we examine the results of our simulation experiments. Throughout this section, normalized execution time is defined as the ratio of a given experiment’s execution time to the execution time that we observed when queue operations are infinitely fast (i.e., require 0 seconds to complete). Except as otherwise stated in this section, all of these experiments used the parameters specified in Table 2.

4.1. Impact of match cost

Scientific workloads use MPI in many different ways, cf.\textsuperscript{[11,28]}. As a result, different workloads respond differently to changes in the time required for message matching. In this subsection, we examine the relationship between the time required to search a match queue entry. The results are shown in Fig. 2. For each of our workloads, this figure shows a sequence of stacked bar charts. Each stacked bar corresponds to a different value of the cost of a queue

---

\textsuperscript{4} This value is consistent with inter-collective time of several important HPC applications, see\textsuperscript{[11]}.

\textsuperscript{5} Unique MPI_TAG values that are outside of the application’s tag space are used to ensure collective messages do not interfere with point-to-point operations.
2. Impact were Table of the Number Attribute Queue normalized run 2 Large-scale approximates (LAMMPS). volumetric Hydrodynamics experiments sparse (CTH-st). simulation A Materials unstructured description interactions Lennard-Jones Energy National Laboratories [12,13]. Data presented uses the Lennard-Jones (LAMMPS-lj) potential. 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 [14]. A benchmark that generates and solves a synthetic 3D sparse linear system using a local symmetric Gauss-Seidel preconditioned conjugate gradient method [15,16]. A multi-material, large deformation, strong shock wave, solid mechanics code [17,18] 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). A proxy application that captures the key behaviors of unstructured implicit finite element codes [19]. Large-scale numerical simulation of quantum chromodynamics (QCD), the theory of the strong interactions of subatomic physics [20].

---

6 In the context of the linear match cost model used for these data, a queue operation is the search a single entry of the match queue.

---

**Table 2**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of processes</td>
<td>128 (125 for LULESH)</td>
</tr>
<tr>
<td>Queue depth multiplier</td>
<td>1</td>
</tr>
<tr>
<td>Match cost model</td>
<td>Linear</td>
</tr>
<tr>
<td>Time per searched entry</td>
<td>100 ns</td>
</tr>
</tbody>
</table>

**Table 3**

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAMMPS</td>
<td>Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS). A classical molecular dynamics simulator from Sandia National Laboratories [12,13]. Data presented uses the Lennard-Jones (LAMMPS-lj) potential.</td>
</tr>
<tr>
<td>LULESH</td>
<td>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 [14].</td>
</tr>
<tr>
<td>HPCG</td>
<td>A benchmark that generates and solves a synthetic 3D sparse linear system using a local symmetric Gauss-Seidel preconditioned conjugate gradient method [15,16].</td>
</tr>
<tr>
<td>CTH</td>
<td>A multi-material, large deformation, strong shock wave, solid mechanics code [17,18] 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).</td>
</tr>
<tr>
<td>miniFE</td>
<td>A proxy application that captures the key behaviors of unstructured implicit finite element codes [19].</td>
</tr>
<tr>
<td>MILC</td>
<td>Large-scale numerical simulation of quantum chromodynamics (QCD), the theory of the strong interactions of subatomic physics [20].</td>
</tr>
</tbody>
</table>
Specifically, these data indicate the point at which the matching cost begins to have a significant impact on workload time-to-solution. For example, Fig. 2a shows that the message matching cost begins to significantly impact the performance of HPCG when the per-entry search time is between 10 and 100 μs. Fig. 3a shows that this corresponds to a rise in the median RQ cost per match attempt from approximately 28 μs to approximately 254 μs. This interval provides a target per-match-attempt cost to minimize the impact of message matching on the time-to-solution for HPCG. Overall, the data in these two figures provide valuable guidance on how fast match attempts need to be in order to minimize the impact on workload performance.

### 4.2. Matching cost models

The impact of matching costs will depend on the data structure that an MPI implementation uses for its match queues. As discussed in more detail in Section 3.1, we consider the matching cost models associated with four potential implementations of match queues: linear/superlinear, logarithmic, and fixed-cost. The linear and superlinear models approximate a linked-list implementation (for different processor architectures); searches are exponential (O(1)) but queue insertion is fast (O(1)). The logarithmic model approximates a binary tree: searches are fast (O(log2 n)) but insertion is slow (O(log2 n)). The fixed-cost model approximates a hash table or specialized hardware devices (e.g., TCAMs): searches are very fast (O(1)) as are insertions (O(1)).

In this subsection, we examine how the choice of model impacts application performance as a function of the queue operation cost. Fig. 4 shows the results of these experiments. For each workload and each queue operation cost, this figure shows three colored bars. The blue bars represent the normalized execution times for the fixed-cost model. The aqua bars represent the normalized execution times for the logarithmic model. The orange bars represent the normalized execution times for the linear model. The orange bars are the same values that appeared in Fig. 2 (without the execution time breakdown). The purple bars are the normalized execution times for the superlinear model. Unsurprisingly, these data show that the fixed-cost model outperforms the other three models. However, the differences in performance of the other three models are relatively modest. For LAMMPS-lj and miniFE, the application slowdown for the fixed-cost model is less than 6% even when a queue operation requires 1 ms. Although the fixed-cost model significantly improves application performance for LULESH, HPCG, and MILC for 1 ms queue operations, these applications still experience significant slowdowns: 3.9 × slower for LULESH, 1.3 × slower for HPCG, and 2.6 × slower for MILC. The performance of the superlinear model is very close to the performance

---

7 This result is consistent with our expectations; the delays introduced by increases in the time required for message matching may perturb the match queues, but the number of entries that each application searches should be largely independent of the cost of a queue operation.

8 This result was also consistent with our expectations. The number of entries that a workload must search per match attempt should be largely independent of the time required to complete a queue operation.

9 Although both are constant-time operations, insertions are typically much slower than searches, see [9,30].

---

Table 4 shows the total time each of our workloads spent on message matching as a function of the time required to complete a queue operation. These data show that total time spent matching is nearly a linear function of the queue operation time. The non-linear behavior associated with very fast queue operations (e.g., 1 ns and 10 ns) is due to the fact that the fixed cost (10 ns) of inserting an entry after a miss is significant when other queue operations are so fast. The slope of the function is total number of entries that are searched by the workload. The number of elements searched is a function of the number of communication operations that the workload performs and the number of entries that have to be searched on each match attempt, including entries that have to be searched when inserting an entry into a queue following an unsuccessful match attempt (miss). Part of the reason that the performance of LAMMPS-lj and miniFE is relatively insensitive to message matching is that they search a relatively small number of queue entries. For 100 ns queue operations, LAMMPS-lj and miniFE spend less than 3 ms matching messages. In contrast, HPCG (14.0 ms), MILC (34.2 ms), and LULESH (35.0 ms) spend significantly more time on message matching and are thus more sensitive to increases in the cost of performing queue operations. Time spent on message does not entirely explain performance variability; CTH-st spends approximately 15 ms on matching, but its performance is similar to that of miniFE.

Fig. 3 shows the distribution of the mean match attempt time per process. These data show that the mean match attempt time is nearly a linear function of the time required to perform a queue operation. The slope of the function is the mean search depth (i.e., the total number of entries searched per match attempt). For 100 ns queue operations, the median LAMMPS-lj, CTH-st and MILC processes require between 93 and 108 ns per match attempt in the RQ. The median processes for the other workloads require between 250 and 381 ns per match attempt. For the PQ, the median HPCG and MILC processes require approximately 163 ns and 113 ns per match attempt, respectively. The median processes for the other workloads require significantly less time: between 33 and 63 ns.

The mean time per match attempt does not appear to be strongly correlated to application performance. The median miniFE process spends significantly more time per RQ match attempt than HPCG, MILC, or LULESH but its performance is less sensitive to increases in matching time. Similarly, the median LULESH process spends less time per match attempt than miniFE or HPCG, but is more sensitive to increases in the match cost.

Combining the data in Fig. 2, 3 provides additional insight into how the per match attempt cost affects application performance.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Per-entry search cost</th>
<th>UQ total match time</th>
<th>Per-entry search cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTH-st</td>
<td>332 μs</td>
<td>1 ns</td>
<td>9 ns</td>
</tr>
<tr>
<td>LAMMPS-lj</td>
<td>6 μs</td>
<td>0.02 ms</td>
<td>0.2 ms</td>
</tr>
<tr>
<td>HPCG</td>
<td>186 μs</td>
<td>1 ms</td>
<td>9 ms</td>
</tr>
<tr>
<td>MILC</td>
<td>583 μs</td>
<td>2 ms</td>
<td>17 ms</td>
</tr>
<tr>
<td>miniFE</td>
<td>35 μs</td>
<td>0.25 ms</td>
<td>2 ms</td>
</tr>
<tr>
<td>LULESH</td>
<td>462 μs</td>
<td>3 ms</td>
<td>31 ms</td>
</tr>
</tbody>
</table>
of the linear model. For most of our workloads (HPCG, LULESH, and miniFE), the logarithmic model provides benefit over the linear model, but the benefit is modest. The reason for these two phenomena is that successful searches (hits) tend to occur in the first few entries of the match queues and the match queues tend to be short as well, cf. [24]. Additionally, inserting new entries in the logarithmic model is much more expensive than the linear model. As a result, much of the benefit of faster searches is eroded by slower insertions after an unsuccessful search (miss). For LAMMPS-lj and CTH-st, the logarithmic model results in greater performance degradation than the linear model. The reason for this is that for these two workloads matches tend to occur in the first or second entry of the match queue. As a result, in most cases, a tree-based search provides little or no benefit. Moreover, the cost of inserting new entries is significantly higher for the logarithmic model than for the linear model, which has a fixed cost for insertion. Figs. 5
4.3. Increasing match queue length

The performance impact of message matching will depend on how long the match queues are. In this subsection, we examine the impact of increasing the length of the match queues on application performance. Details on how we scale the match queues is available in Section 3.2. The results of our experiments are shown in Fig. 7. For each workload and matching cost model, this figure shows our experimental results as a cluster of bars. The blue bars represent the normalized execution time for the fixed-cost model. The aqua bars represent the normalized execution time for the linear model. These results show that for all of our workloads, if we increase the length of the match queues by \(100 \times \) the workload would run less than 9% slower than the ideal case (i.e., where queue operations are infinitely fast). Moreover, for CTH-st, LAMMPS-lj, and miniFE, the performance degradation would be less than 5% even if the match queues were \(1,000 \times \) longer. Additionally, even if match queues were \(1,000 \times \) longer, the performance degradation for the fixed-cost and logarithmic models would be less than 1% for all of our workloads. However, for the linear and superlinear models, performance degradation of LULESH and MILC (and to a lesser extent HPCG) would be significant if queue lengths grow by a factor of 1,000.

These results also demonstrate that the differences between our four models are quite modest until match queues are very long (e.g., \(>100 \times \) current queue lengths). The benefits of the logarithmic and fixed-cost queue models and the costs of the superlinear model are not evident until the match queues contain more than a few hundred elements. Although these data show that the costs/benefits of these queue models are only noticeable at the extreme, the data in Fig. 4 showed that even when queue operations are very slow (e.g., \(>100 \mu s\)) the benefits of the logarithmic model and the costs of the superlinear model were quite modest.\(^\text{10}\)

Table 5: Computation-to-communication ratio for each of our workloads.

<table>
<thead>
<tr>
<th>Number of processes</th>
<th>LAMMPS-lj</th>
<th>HPCG</th>
<th>LULESH</th>
<th>miniFE</th>
<th>CTH-st</th>
<th>MILC</th>
</tr>
</thead>
<tbody>
<tr>
<td>27/32</td>
<td>33.830</td>
<td>17.039</td>
<td>6.602</td>
<td>2.249</td>
<td>2.767</td>
<td>6.762</td>
</tr>
<tr>
<td>64</td>
<td>27.785</td>
<td>21.711</td>
<td>3.950</td>
<td>2.792</td>
<td>2.126</td>
<td>5.842</td>
</tr>
<tr>
<td>125/128</td>
<td>15.605</td>
<td>20.858</td>
<td>1.045</td>
<td>5.237</td>
<td>1.477</td>
<td>4.492</td>
</tr>
<tr>
<td>216/256</td>
<td>14.127</td>
<td>17.298</td>
<td>1.151</td>
<td>4.605</td>
<td>0.929</td>
<td>2.839</td>
</tr>
</tbody>
</table>

4.4. Strong-scaling workloads

Increasing the number of processes used to run a workload will also impact its performance. In this subsection, we examine the impact of strong-scaling our workloads on their performance. Strongly-scaling a workload divides the same amount of work among a larger number of processes. As a result, each process has a smaller problem to solve but, in many cases, has to perform roughly the same amount of communication. Table 5 shows the computation-to-communication for each of our workloads. Strong-scaling does not universally decrease this ratio. For example, as we strongly scale miniFE the ratio actually increases slightly. Nonetheless, for our other workloads, the computation-to-communication ratio does generally decrease with scale. Although the computation-to-communication ratios do not totally account for the changes in workload performance, scaling a workload so that it spends more time on communication relative to the amount

\(^\text{10}\) The reason for the difference between the performance of the logarithmic model in Figs. 4 and 7 is the way that the match cost is computed. In Fig. 4, if \(n\) entries are searched and each queue operation takes \(t\) seconds, the time required is: \(t \times ([\log_2(n)] + 1)\). For small values of \(n\), the value of this expression is close to \(t \times n\). On the other hand, in Fig. 7 the time required for this set of operations is: \(t \times ([\log_2(n)] + \log_2(m) + 1)\), where \(m\) is the queue length multiplier.
of computation will increase its sensitivity to increases in communication costs. Additionally, because increases in scale also increase the number of participants in each collective operation. Larger collectives generally increase the number of communication dependencies along which delays may propagate.

The results of our strong-scaling experiments are shown in Fig. 8. For each workload and each value of queue operation cost, the results of our experiments are shown as a cluster of four colored bars. From left to right, these bars represent the normalized execution time that corresponds to a factor of two increase in the number of processes used to run the workload. Each result is normalized relative to the case where the same number of processes are used but queue operations are infinitely fast. These data confirm that strongly-scaling our workloads increases their sensitivity to increases in the time required to perform a queue operation.
4.5. Weak-scaling workloads

We can also change the number of processes running our workloads by weak-scaling them. LogGOPSim provides this functionality, however, it only approximates weak-scaled point-to-point operations. Previous work has demonstrated that the performance impact of introducing delays into a workload is largely determined by the frequency with which the workload performs collective operations [11]. LogGOPSim extrapolates collective operations exactly. Therefore, we believe that it is instructive to understand how the time required for MPI message matching might affect the performance of applications running on a much larger scale. In this subsection, we discuss the results of our weak-scaling experiments. Using the extrapolation capability of LogGOPSim we weak-scaled our applications from 2,048 to 16,384 processes in powers of 2 (2,000 to 16,000 for LULESH\footnote{In order to understand how the impact of matching operations impact LULESH (and applications with similar communication patterns), we have included results from extrapolations possible with LogGOPSim that may represent configurations not possible with native LULESH.}). The results are shown in Fig. 9. For each value of per-entry search cost, the cluster of bars represents the normalized execution time for each of four execution scales. These data show that the performance impact of matching costs is relatively constant with increasing scale. Moreover, these results are consistent with the results for 128 processes in Fig. 2.

4.6. Multithreading

In this section, we examine how multithreaded MPI may impact the performance impact of MPI message matching. Using an extension of the benchmark originally developed by Schonbein et al. [10], see Section 3.3 for details. For our analysis, we ran the benchmark over three process counts (8, 16, and 32), six threading configurations (1, 3, 9, 18, 36, and 72), and two three-dimensional stencils (7-point, and 27-point) on an Infiniband cluster. Each node of the cluster has two sockets. Each socket is occupied by an 18-core Intel Broadwell processor (E5-2695v4) with hyper-threading enabled. Thus, each node supports up to 72 hardware threads. For each benchmark configuration, we collected MPI execution traces.

As we discussed in Section 3.3, one of the challenges introduced by multithreading is that the order in which individual threads send and receive messages is difficult to predict. As a result, the workload cannot rely on messages being processed in a particular order and may need search further in the match queue to find a match. An execution trace of our multithreaded benchmark captures a specific ordering of thread execution. To examine the impact of different orderings, we constructed a script that randomized the order in which messages were sent and received (within a single iteration) by manipulating the trace file. For each trace we created 10 variants, each with a different message ordering. We then used LogGOPSim to simulate the benchmark for each trace.

The results of these experiments are shown in Figs. 10 (7-point stencil) and 11 (27-point stencil). The y-axes in these figures show the normalized execution time of each experiment. The execution time of each experiment is averaged over 10 simulations, one for each randomized trace. In each set of experiments, the results are normalized relative to the results for the same configuration but with a queue operation cost of zero.\footnote{As discussed in Section 3.5, a queue operation cost of zero is the default configuration of LogGOPSim.}

Fig. 10 shows that the number of threads has a small impact on benchmark performance until MPI message matching becomes very expensive, e.g., when the queue operation cost exceeds 10 μs. When the queue operation cost is 100 μs or greater, the benchmark does run slower as the number of threads increases. Increasing the number of processes also increases the sensitivity of the benchmark to queue operation cost. When the queue operation is 100 μs or greater, the benchmark does run slower (relative to its baseline) for larger numbers of processes. These results indicate that increasing the number of threads (or processes) does increase the cost of MPI message matching, but the time spent on message matching
still accounts for a relatively small fraction of the benchmark’s execution time for a 7-point stencil in most configurations.

**Figs. 11** shows that the number of threads has a noticeable impact on benchmark performance as the number of processes and threads increases. For example, running the benchmark with 32 processes and 72 threads per process (a total of 2304 threads) shows a significant slowdown (≈57%) even when the per-entry search cost is only 1 μs. As the per-entry search cost exceeds 1 μs, the benchmark’s runtime increases dramatically. Although per-entry search costs are much less than 1 μs (and are likely to remain that way), this 27-point stencil demonstrates much greater sensitivity to message matching costs than any other workload that we considered. When per-entry searches are very expensive (e.g., greater than 10 μs) running the benchmark on 32 processes is more than 8 × slower than the default case (when per-entry searches are infinitely fast). Nonetheless, for values of the per-entry search time that are close to what we currently observe, see e.g., Barrett et al. [7], message matching has a small impact on the benchmark’s total execution time.

For both stencils, the ordering of threads within an iteration has a very small impact on the benchmark’s execution time. The coefficients of variation for all of our 27-point and 7-point stencil experiments are less than 0.005 and 0.003, respectively.

5. Related work

5.1. Measuring MPI message matching performance

Concerns about the future of MPI message matching have prompted many researchers to examine how MPI message matching may affect network performance. Barrett et al. [7] focused on how the cost of matching messages with hybrid-core processors might affect MPI message rate as a function of match queue length. Their measurements also showed that the per entry search time on modern multicore processors is below 10 ns even for very long searches. Processors with simpler cores (e.g., ARM Cortex-A9) exhibited growth in per-entry costs as the match queue lengths increased. Bridges et al. [31] constructed a model of how match queue length may affect message rate.

As part of their evaluation of their matching algorithm, Flajslik et al. [5] showed that the per-entry search time for LAMMPS (using the Rhodospin potential) was about 30 ns. The authors also showed that their new matching algorithm enabled significant speedups of the Fire Dynamics Simulator.

Our analysis is informed by the empirical results in these publications. We build upon these works, using our simulation framework to examine the relationship between the per-entry search time and application time-to-solution.

5.2. Accelerating MPI message matching

Attaining high performance with MPI applications necessitates the ability of MPI runtime to provide communication with very low overhead. Predictions about the potential decline of message matching performance have prompted many researchers to propose methods for accelerating MPI matching. These proposals include specialized hardware [23], relaxation of matching semantics [6], exploiting application communication characteristics [4], and improvements in the design of match queues [5] (software hash tables).

In this paper, we presented four models of match queue performance that capture the high-level characteristics of these approaches. Our work is distinct from this existing research because we have focused on understanding the circumstances under which these approaches meaningfully improve application performance.

6. Conclusion

Significant research has been conducted on developing highly-efficient MPI message matching algorithms. In this paper, we examined the circumstances under which matching activities will
have a significant impact on production, extreme-scale workload performance. The data generated by our experiments yielded several important conclusions, including:

- Accelerating match queue performance will have little effect on the performance on current workloads. Even if the time per match attempt grows superlinearly, the match queues of current workloads would have to be more than 100× longer to significantly affect their performance.
- The benefits of fixed-cost or binary-tree-based match queues are unlikely to be realized on modern multicore processors unless and until we see dramatic growth (e.g., >1,000×) in match queue length. Alternatively, if queue operations become significantly slower (e.g., >100 μs), fixed-cost match queues may provide significant benefit for some workloads. However, binary-tree-based match queues provide little benefit in this circumstances because most match attempts in current workloads traverse a small number of match queue entries.
- Understanding the relationship between the per-match-attempt cost of MPI message matching and workload performance provides important guidance for the design and development for MPI message matching algorithms. Predicting when matching will limit application performance enables developers to carefully target their efforts at those parts of MPI that are most critical for high performance.
- Message matching has the potential to have a larger impact on the performance of workloads that use multithreaded MPI that it does on current single-threaded workloads. Nonetheless, the results from the simple approach to multithreading that we used in this paper suggest that the impact on execution time is likely to be small unless and until per-entry search costs increase substantially above what has been observed on current systems.

Overall, this paper shows that the performance impact of MPI message matching on current workloads is modest and is likely to remain that way. However, future developments still have the potential to change workload behaviors in ways that cause the performance impact of message matching may grow. Therefore, we have also provided valuable guidance on how to maintain high-performance MPI implementations in the future.

Acknowledgments

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & 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. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

References