Received: 9 October 2017 | Revised: 7 June 2018 | Accepted: 13 June 2018
DOI: 10.1002/cpe.4862

WILEY

SPECIAL ISSUE PAPER

Communication-aware message matching in MPI

S. Mahdieh Ghazimirsaeed | Seyed H. Mirsadeghi | Ahmad Afsahi

Department of Electrical and Computer Engineering, Queen's University, Kingston, ON, Canada

Correspondence
S. Mahdieh Ghazimirsaeed, Department of Electrical and Computer Engineering, Queen's University, Kingston, ONK7L 3N6, Canada.
Email: s.ghazimirsaeed@queensu.ca

Seyed H. Mirsadeghi, Department of Electrical and Computer Engineering, Queen's University, Kingston, ONK7L 3N6, Canada.
Email: s.mirsadeghi@queensu.ca

Ahmad Afsahi, Department of Electrical and Computer Engineering, Queen's University, Kingston, ONK7L 3N6, Canada.
Email: ahmad.afsahi@queensu.ca

Funding information
Natural Sciences and Engineering Research Council of Canada, Grant/Award Number: #RGPIN/05389-2016; Canada Foundation for Innovation, Grant/Award Number: #7154

Summary

The Message Passing Interface (MPI) is the de facto standard for parallel programming in High Performance Computing (HPC). Asynchronous communications in MPI involve message matching semantics that must be satisfied by the conforming libraries. The matching performance is in the critical path of communications in MPI. However, the current message matching approaches suffer from scalability issues and/or do not consider the message queue characteristics of the applications. In this paper, we propose a new message matching mechanism for MPI that can speed up the operation by allocating dedicated queues for certain communications of an application. More specifically, we propose a design that categorizes communications into a set of *partners* and *non-partners* based on the communication frequency in the corresponding queues. We propose a static and a dynamic approach for our message matching design. While the static approach works based on the information from a profiling stage, the dynamic approach utilizes the message queue characteristics at runtime. Our experimental evaluations show that the proposed design can provide up to 28x speedup in queue search time for long list traversals without degrading the performance for short list traversals. We can also gain up to 5x speedup for the FDS application, which is highly affected by the message matching performance.

KEYWORDS

message matching, message queue, MPI, partner/non-partner queues

1 | INTRODUCTION

The Message Passing Interface (MPI)[1] is the de facto standard for parallel programming in High Performance Computing (HPC). The processes in MPI require a collaborative mechanism to exchange the data with their peers. In this regard, the efficiency of inter-process communications is one of the most important challenges in MPI and has a significant impact on the performance of parallel applications.

In order to support the asynchronous communications in MPI, the MPI libraries often use two message queues: Posted Receive Queue (PRQ) and Unexpected Message Queue (UMQ). When a message arrives at the receiver, the PRQ must be traversed first to locate a matching receive queue item. If no matching item is found, a Queue Element (QE) is added to the UMQ. On the other hand, when a receive operation is called, the UMQ is searched first to see whether the desired message has already (unexpectedly) arrived. If not, a new QE is added to the PRQ. The overheads associated with traversing these message queues can become a bottleneck in MPI applications that build long message queues. Therefore, designing an efficient message matching mechanism is of high importance for high-performance communications.

The message matching mechanisms currently used in well-known MPI libraries such as MPICH,[2] MVAPICH,[3] and Open MPI[4] or those proposed in literature[5-7] do not consider the message queue characteristics of the applications in devising the message queue data structure and managing its operations. The only exceptions are the works of Bayatpour et al[8] and Ghazimirsaeed and Afsahi.[9] The approach used in the work of Bayatpour et al[8] switches between three different data structures: 1) linked list as in MPICH or MVAPICH; 2) rank based as in Open MPI; and 3) bin based,[6] based on the number of message traversals by a process at runtime or based on a configuration parameter. However, as detailed in Section 3, this work suffers from a number of aspects associated with the aforementioned data structures, including memory scalability of the rank-based approach and not knowing the optimal number of bins/queues as in the bin-based approach. In the work of Ghazimirsaeed and Afsahi,[9] the authors propose an MPI message matching mechanism based on K-means[10] clustering that considers the message queue behavior of the applications statically to categorize the communicating peers into clusters and assign a dedicated queue to each cluster. One of the disadvantages of the K-means clustering approach

is that the optimal number of clusters (K) is not known in advance. The work also has memory scalability issue due to maintaining information about clusters for each peer process.

To deal with aforementioned issues, in this paper, we propose a novel MPI message queue architecture based on *partner/non-partner* message queue traffic that decreases the queue search time and maintains a scalable memory consumption. The main contributions of this paper are as follows:

- Gathering the dynamics of the message queue characteristics of the application with two different approaches: *static* and *dynamic*. In the static approach, the application is executed once to gather the profiling information. This information is used in subsequent executions. On the other hand, the dynamic approach gathers application message queue characteristics at runtime.
- Allocating dedicated queues for messages coming from certain processes with respect to the profiling/runtime information. In other words, we build a collection of message queues that belong to two different classes: *partners* and *non-partners*. While each partner queue is used only for messages from a certain partner, the non-partner queue provides a shared container for messages from all non-partner processes. The static approach allocates all the required queues at once based on the profiling information. The dynamic approach allocates the queues gradually based on the application runtime characteristics.
- Providing the runtime and memory complexity of the proposed design and compare it with message queue data structures in well-known MPI implementations such as MPICH, MVAPICH, and Open MPI. The complexity analysis shows that the proposed message matching mechanism is scalable in terms of speed of operation compared to linked list data structure in MPICH/MVAPICH. Moreover, it provides scalable memory consumption compared to Open MPI queue data structure.
- Demonstrating the benefits of the proposed design through experimental evaluation and comparing it with the linked list data structure used in MVAPICH. The results show that we can improve the queue search time performance by up to 28x and the application runtime as high as 5x for the FDS[11] application.

The rest of the paper is organized as follows. Section 2 provides the background information and discusses the motivation behind this work. Section 3 presents the related work and distinguishes our work from them. Section 4 describes the static and dynamic approaches in the proposed message queue design. Section 5 presents the runtime and memory complexities of our design and compares them with the linked list and Open MPI data structures, respectively. The experimental results are presented in Section 6. Finally, we conclude the paper in Section 7.

2 | BACKGROUND AND MOTIVATION

In MPI, message matching is done based on the tuple context-id, rank, and tag. The context-id is an integer value that specifies the communicator. The rank is the corresponding address for each process in a communicator and it represents the source process rank in a PRQ request, and the receive process rank in an UMQ request. The tag is an integer value used to distinguish among the messages from the same rank. Various queue data structures have been used in MPI libraries. MPICH[2] and MVAPICH[3] use the linked list data structure. One of the advantages of the linked list is in its minimal memory consumption. The other advantage is that the QEs are stored in the order of their arrival, which makes it compatible with the MPI point-to-point ordering semantics. However, linked list becomes very inefficient for long message queues due to its high traversal costs which is of $\mathcal{O}(q)$, where q denotes the number of elements in the queue.

The linear queue structure can be improved since the context-id restricts the rank space and the rank restricts the tag space for a given request. Accordingly, Open MPI[4] uses a hierarchical structure by considering context-id and rank as the first and second levels, respectively. Each context-id associated with a communicator of size n has an array of size n. Each element of the array corresponds to a rank that has a pointer to a linked list dedicated to the messages coming from that rank. Therefore, once a context-id is found, the short list of UMQ or PRQ can be reached in $\mathcal{O}(1)$. Consequently, the queue search time in this approach is significantly faster than the linked list data structure. However, it comes at the expense of a much higher memory consumption required for an array of size n for each communicator of size n. Such a high memory footprint becomes prohibitive at large scales.

2.1 | Motivation

Some applications do not generate long queues. For such applications, a linear linked list search is acceptable. However, for applications with long list traversals, linked list imposes a significant overhead not only due to its computational complexity[12] but also the large number of pointer operations that can cause a large memory access penalty at the cache level. Thus, it is important to have a message queue design that is fast in terms of traversing/matching and also scalable in terms of memory consumption.

The array-based data structure in Open MPI highly improves the performance. However, the problem is that it allocates one queue for each peer process. Such an allocation scheme may not incur high memory overheads at small scales. However, the once-for-all allocation scheme causes linear degradation of memory consumption at large scale. Moreover, allocating arrays equal to the size of the communicators will waste high amounts of memory at large scales as many elements of the array might not be even used at all. The reason is that most well-crafted MPI applications avoid the fully connected communication pattern in which all processes communicate with all the other processes. Consider AMG[13] and LAMMPS[14] application as an example. Figure 1 shows the total number of elements sent to the queues from different processes in AMG2006 and LAMMPS. In

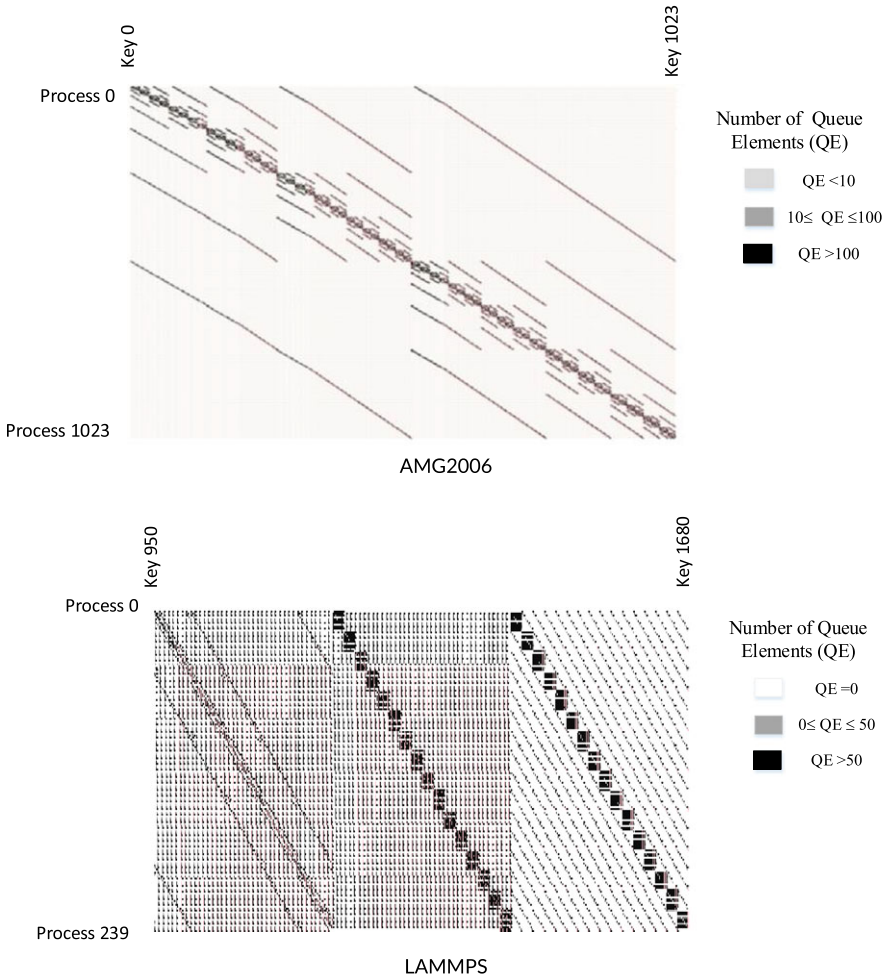

FIGURE 1 Number of elements sent to UMQ/PRQ from different processes in the AMG2006 and LAMMPS applications

the figure, each row shows the queue profiling information for one process. Each column corresponds to a key value that represents a process with a specific rank and context-id derived from Equation (1). In this equation, P is the number of processes and *Mapped_context_id* is an integer value derived from mapping the context-id to a small integer range between 0 and the total number of communicators minus one

$$Key = Rank + (P \times Mapped\_Context\_id). \tag{1}$$

In Figure 1, we only show the queue profiling information for a fraction of the keys. The black and dark gray data points represent high communicating processes, whereas the white and light gray data points show low communicating processes. It can be seen that many processes send a few or no messages to the queue of the other processes (98% and 91% of the data points for AMG2006 and LAMMPS applications are white, respectively). This shows the inefficiency of the Open MPI data structure in terms of allocating unnecessary memory for such processes. In other words, Open MPI wastes up to 98% and 91% of the memory used for the queues allocated to peer processes for AMG2006 and LAMMPS applications. This observation, along with the unscalable performance of the linked list, motivates us to design message matching mechanism that is scalable in terms of both speed of operation and memory consumption. In the proposed message queue data structure, we avoid wasting memory by considering the communication pattern and queue behavior of the applications so that we can allocate dedicated message queues only to the processes with large number of messages in the queues. Moreover, this approach speeds up the search operation and reduces the number of queue traversals. In the following, we contrast our proposed message matching design with other research proposed in literature.

3 | RELATED WORK

Several works have been proposed to reduce the number of queue traversals in order to improve the queue search time.[5-8] The 4-dimensional data structure[5] decomposes ranks to multiple dimensions. This way, it skips searching a large portion of the queue for which the search is guaranteed to yield no result. The problem with 4-dimensional data structure is that it has a small fixed overhead for searching any queue item. If the queue length is large enough, this overhead is negligible. However, in the case of short list traversals, this data structure performs worse than the linked list data structure. In order to mitigate this problem, the authors used the communicator size as a metric to decide whether to use the 4-dimensional data

structure. However, the communicator size is not always a good indicator of the average search length. For example, even with a communicator size of two, the two processes can have a lot of communications with each other, which results in long list traversals.

In the work of Flajslik et al,[6] the authors take advantage of a hash function to speed up the search operation. The proposed data structure consists of multiple bins (or queues) with a linked list for each bin. There are two problems with the bin-based approach in the work of Flajslik et al.[6] First, it does not determine the required number of bins for each process in a given application in order to have a good distribution of messages across the bins. While increasing the number of bins would accelerate the search operation for long list traversal, the optimal number of bins is totally application dependent and it might be different for each process in the application. Therefore, allocating a large number of bins in this approach for each and every process would result in unnecessary, large memory footprint at scale. The second issue with the bin-based approach is that it suffers from some overhead for searching short queues. Our work differs from this work in that it avoids the unnecessary memory overhead by determining the suitable number of queues for each process based on application message queue trace. Moreover, our approach avoids sharing the bins among the processes with high frequency of communication and instead allocates a dedicated queue to each of such processes.

The authors in the work of Bayatpour et al[8] try to address the overhead issue for short list traversals in the bin-based approach by proposing a message matching design that dynamically switches to one of the existing designs: a linked list, a bin-based design,[6] and a rank-based design. This work always starts with the default linked list. When the number of traversals crosses a threshold, it switches to the bin-based design. It can also switch to the rank-based design if it is requested by the user at configuration time. This work performs better than the linked list data structure for long list traversals and better than the bin-based/rank-based design for short list traversals. However, it suffers from the issues associated with these data structures. For example, it does not find the number of bins/queues for a given application adaptively and dynamically. Switching to the rank-based design requires a user-configurable parameter and also allocates a dedicated queue for each source process, resulting in low memory scalability. In the work of Ghazimirsaeed and Afsahi,[9] the authors propose a message matching strategy based on the K-means clustering algorithm that profiles the message queue behavior of the applications statically to categorize the processes into some clusters. It will then assign a dedicated message queue to each cluster. The problem with this approach is that it groups the processes who send similarly large number of queue elements into the same cluster/queue, highly increasing the queue search time for such processes. In addition, there is no definitive way to determine the optimal number of clusters. The proposed work is also not scalable in terms of memory as it needs to store the cluster information for each communicating process. The proposed partner/non-partner message queue design in this paper differs from the works of Bayatpour et al[8] and Ghazimirsaeed and Afsahi[9] in that it profiles the message queue communication traffic between each communicating peers and allocates a dedicated message queue only to the processes with high frequency of communication. Moreover, our approach determines the appropriate number of queues dynamically during the application runtime and avoids sharing the queues between high communicating processes.

The authors in the work of Klenk et al[7] leverage GPU features to improve the queue traversal time. They take advantage of the availability of a large number of GPU threads to perform the search operation in two different phases of scan and reduce. One of the issues is that the proposed design cannot achieve high message rates when the length of queue is larger than 1K. Moreover, the reduce stage is done sequentially and it cannot be overlapped with the scan stage in a fully-compliant MPI semantic scenario.

As for hardware solutions, the work of Underwood et al[15] offloads MPI message matching to specialized hardware to accelerate message queue traversals. Barret et al[16] evaluate many-core matching rates. Portals[17] enable offloading MPI message matching to reduce MPI messaging latency. Several low-level networking APIs leverage specialized hardware and software techniques to provide interfaces that support message matching.[18,19] New approaches such as CH4 in MPICH consider the scalability issues in MPI and propose hardware supported message matching.[2,20] Some earlier works studied the impact of the message queue characteristics on the application performance[21-24] or they evaluate MPI message matching performance.[25-27]

4 | PARTNER/NON-PARTNER MESSAGE QUEUE

The core idea of our proposed design is to allocate a dedicated queue for each process that sends/posts a large-enough number of messages/receives to the UMQ/PRQ. We refer to such processes/queues as partner processes/queues. We design the partner/non-partner message queue data structure using two different approaches: a static approach and a dynamic approach. In the static approach, we run the application once to the end and use the gathered message queue information to build all the partner/non-partner queues at once in the future runs. In the dynamic approach, we use the characteristics of the queues at runtime to build partner/non-partner queues dynamically. The dynamic approach is certainly more promising and relevant in practice. However, the static approach is practical in cases where an application is expected to be run many times and each run is considerably affected by message queue traversal overheads.

4.1 | Metrics for selecting partner processes

For extracting the partner processes, we first count the number of elements each process sends to the queue. Then, we use this information to calculate an edge point. The edge point is a threshold parameter for selecting the partner processes. Any process whose number of elements in

the queue is more than the edge point is chosen as the partner. Three different metrics, ie, average, median, and upper fence outliers, are used to determine the edge point. The advantage of the average metric in selecting the partners is that it considers every value in the data. Moreover, its calculation is less intensive compared to the median and upper fence outliers metrics, since it does not require sorting the elements. However, the problem with the average metric is that it is sensitive to the extreme values. On the other hand, median is more robust to the extreme values and is a better indicator of the dataset.

Equation (2) shows the upper fence formulation, where Q_3 and Q_1 are the upper and lower quartiles, respectively, and α is a constant coefficient. Using the upper fence outliers as the edge point, only the processes who have considerably larger number of elements in the queue are chosen as the partners. This would potentially result in extracting fewer partner processes compared to the average and median metrics, leading to fewer partner queues and hence improved memory consumption at the expense of less performance improvement

$$\text{Upper\_Fence} = Q_3 - \alpha \times (Q_3 - Q_1). \tag{2}$$

4.2 | The static approach

Figure 2 illustrates the static partner/non-partner message queue design. In this approach, the application is profiled once to gather the total number of QEs each process sends to the UMQ/PRQ during the entire application runtime. This information is used to identify the partner processes based on one of the metrics discussed in Section 4.1. Information about the partner processes is then saved in a hash table to be used in the future runs, where a dedicated message queue is allocated for each partner process. In addition, a single non-partner queue is allocated for the QEs from all non-partner processes. A hash table is used for its $O(1)$ search complexity and less memory overhead over other data structures, as discussed in Section 5.2.

Algorithm 1 shows the steps involved in extracting the partner processes in the profiling stage. The metrics discussed in Section 4.1 are used to determine the edge point for selecting the partners (Line 1). Any process whose number of elements in the queue is greater than the edge point is considered as a partner and its corresponding < *hash_key, hash_value* > pair is added to the hash table (Line 3 to 10). Equation (1) is used to generate the hash keys; this ensures that the hash key is unique for all the ranks in different communicators. As most applications usually use a small number of communicators, the time for mapping context-id to small integers is negligible. The hash value is a unique integer for each process, between 0 and $n_p - 1$, where n_p is the number of partners for each process. Each hash value indicates one partner queue. We also count the number of selected partners for each process in Line 8.

We should note that any hash function could be used in Algorithm 1. However, considering that in the static approach the partners are selected once in the profiling stage and that they do not change during the future runs of the application, there will be no insertion/deletion to/from the hash table. Therefore, we use a perfect hash function[28,29] to eliminate hash collisions. For the sake of completeness, we present the implementation of

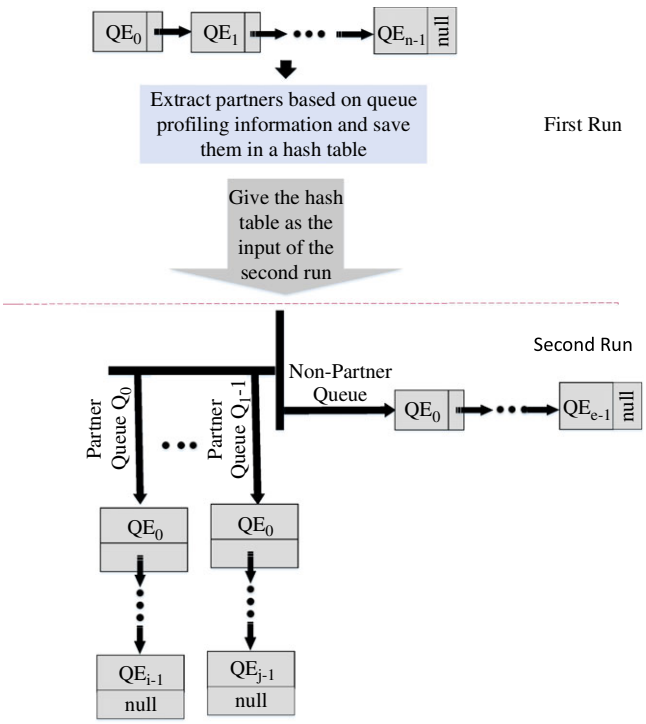

FIGURE 2 Static partner/non-partner message queue design

Algorithm 1 Partner extraction in partner/non-partner message queue design

Input: Total number of elements sent to UMQ/PRQ from each process Num-of-$QE_{0...P}$, Number of processes P

Output: A hash table containing all partner processes HT, The number of partners for process p n_p

```
1:  E = average/median/upper fence outliers of Num-of-QE_{0...P}
2:  n_p = 0
3:  for i ∈ 1...P do
4:      if Num-of-QE_i > E then
5:          Generate hash_key for process i
6:          Determine hash_value(s)
7:          Insert < hash_key, hash_value(s) > to the hash table
8:          n_p++
9:      end if
10: end for
```

the perfect hash function below. The interested reader is referred to the work of Gettys[29] for a detailed description of the perfect hash function. Assuming that the keys are saved in array S, the algorithm to generate perfect hash function is as follows:

1. Find the parameter m such that $(m \times m) \geq max(S)$. Each key is to be mapped into a matrix of size $m \times m$.
2. Place each key k in the matrix at location (x, y), where $x = k \div m, y = k \bmod m$.
3. Slide each row of the matrix to the right some amount so that no column has more than one entry.
4. Collapse the matrix down into a linear array.
5. The hash function uses m and the displacements from step 3 to locate k.

When the application runs a second time, the output of the profiling stage (the hash table and the number of partners for each process n_p) is used to improve the search operation. For that, n_p partner queues and one non-partner queue are created at initialization for each process. Algorithm 2 shows the message matching mechanism in the static approach that is used in the second run. To search for an element in the queue, we first use Equation 1 to generate the search key based on the rank and context-id of the corresponding element (Line 1). Then, the search key is given as the input of the hash function to derive the hash table index (Line 2). If the hash key corresponding to the derived hash table index was equal to the search key, it means that the corresponding process is a partner and hence, we should search the corresponding (dedicated) partner queue specified by the hash value. Otherwise, we should search for the desired element in the non-partner queue (Line 4 to 7).

Algorithm 2 The searching mechanism in the static approach (Second run)

Input: The hash table HT, The partner and non-partner queues, The searching element (Context-id, rank, tag)

```
1: Generate the search_key from rank and context_id
2: hash_table_index = hash_function(search_key)
3: < hash_key, hash_value > ⇐ HT_{Hash_table_index}
4: if hash_key == search_key then
5:     Search the partner queue specified by hash_value
6: else
7:     Search the non-partner queue
8: end if
```

4.3 | The dynamic approach

Unlike the static approach, the dynamic approach identifies the partner processes dynamically at runtime, without a need to a profiling stage. As soon as the size of the queue reaches a specific threshold, t, we identify the partner processes at that level and allocate a dedicated message queue to each of them. From this point on, the incoming QEs from the partner processes are added to their own dedicated queues, whereas the messages from all the other processes (non-partners) are added to a single, shared non-partner queue. This procedure is repeated in a multi-level fashion during the application execution time. The motivation behind this approach is to capture the dynamics of the applications at runtime and allocate the partner queues based on the discovery of the new partners in each phase of the application.

As shown in Figure 3, the dynamic approach consists of multiple levels. Each level includes a number of partner queues and a single non-partner queue. If the length of the non-partner queue reaches the threshold, t, some new partners are identified and the non-partner queue itself is divided into a number of new partner queues and a new non-partner queue.

When searching the queues for non-partner processes, the initial (non-partner) queue at the Base Level is searched first. Then, all the non-partner queues from Level 0 to Level $L - 1$ are searched in sequence. The green arrows in Figure 3 show the order in which a queue element from a

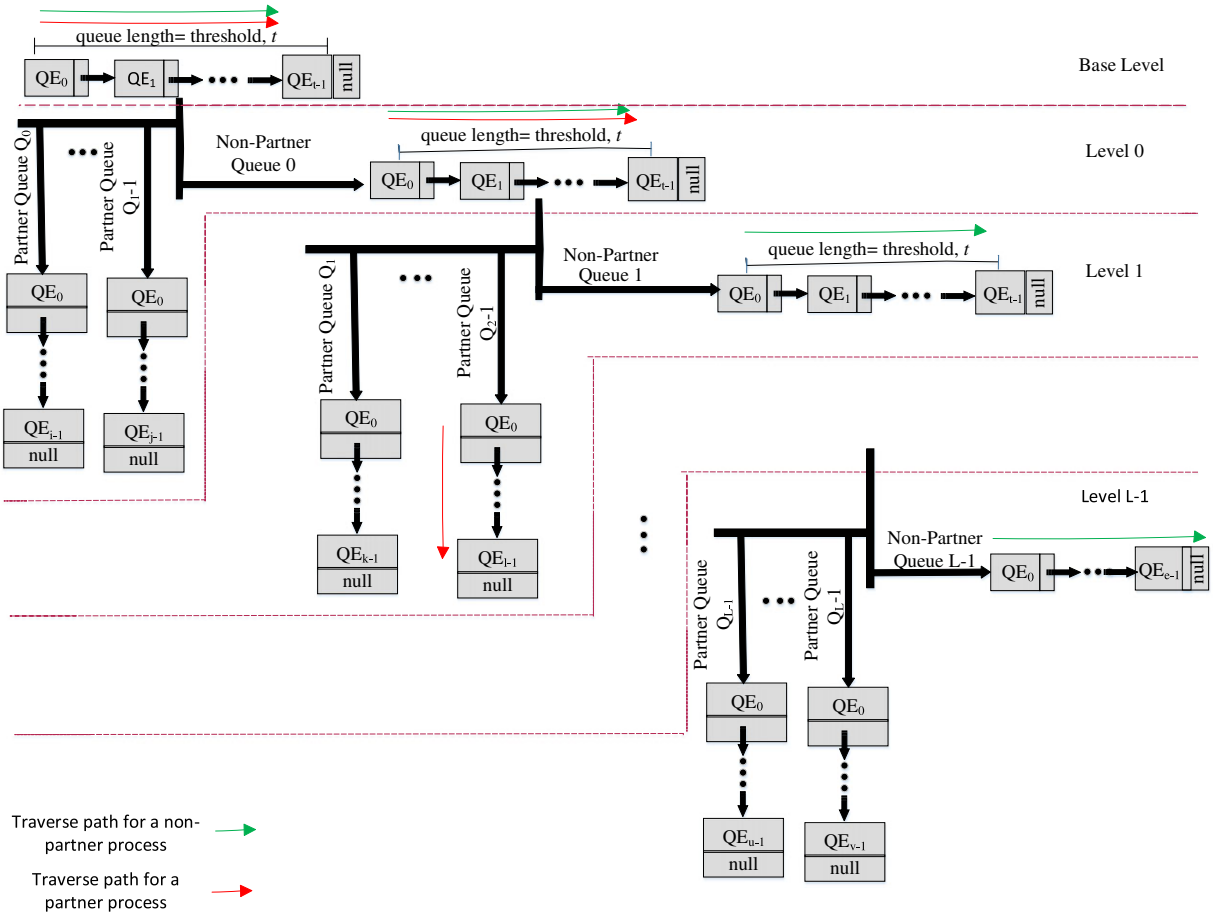

FIGURE 3 Dynamic partner/non-partner message queue design

non-partner process is searched. The order in which the queues are searched for partner processes depends on its level of partnership (ie, the level in which a process becomes a partner*). For example, if a process becomes a partner at Level 0, then the initial queue and its dedicated queue are searched in that order. However, if the process becomes a partner at Level 1, it might still have some elements in the non-partner queue at Level 0. Therefore, the order in which the queues are searched for this partner process will be: 1) the initial queue, 2) the non-partner queue at Level 0, and 3) its own dedicated queue (red arrows in Figure 3).

Similar to the static approach, we use a hash table to store the partner processes and to distinguish them from the non-partner processes at search time. To search the partner queues, the hash table should also contain the queue number and the level of partnership for the corresponding partner process. The hash key is derived from Equation (1) and is unique for each partner process. Each key has two hash values. The first value shows the queue number of the corresponding partner process, whereas the second hash value shows the level of partnership for that process. Any hash function can be used for this purpose. However, we do not use the perfect hash function that was used in the static approach, as it is designed to search a static list with no insertions or deletions. The dynamic approach though requires some new partners to be inserted into the hash table at each level. Therefore, we use a round-robin hash table for our purpose. To maintain memory scalability in the dynamic approach, an important advantage of our design is that we bound the size of the hash table to a sublinear order of the total number of processes. Therefore, new partners are extracted only if the bound on hash table size is not violated. We restrict the total number of partner processes to $c\sqrt{N}$, where N denotes the number of processes and c is an environment variable. The hash table, thus, has $\sqrt{N}$ number of entries, where each entry can have c elements, each for a partner process.

Algorithm 3 shows the message matching design in the dynamic approach for the UMQ. The same scheme is used when searching the PRQ. When searching the UMQ, the initial queue is searched first in Line 1. If the element is found, we are done with the search. Otherwise, we check the level number. If the level number is less than zero, it means that there are no more queues to search and, thus, the element is added to the initial PRQ (Line 37). If the level number is more than 0, it means that there exist some partner and non-partner queues so the search is continued. In order to determine the queues that should be searched at this point, we should find out whether the searching element belongs to a partner process or not. To do so, we simply generate the search key corresponding to the searching element using Equation (1) (Line 6). Then, we look into the hash table for the search key (Line 7 to 9). The same procedure used in the static approach is used here to determine if the search key exists in the hash table. If the search key was in the hash table, it means that the searching element belongs to a partner process (for example, process p). Therefore, all of

*From this point on, we refer to level of partnership for process p as level-of-partnership$_p$

the non-partner queues from Level 0 to the level-of-partnership$_p$ are searched first, followed by the partner queue dedicated to process p (Line 10 to 15). Otherwise, the process is not a partner, so all the non-partner queues from Level 0 to $L - 1$ are searched, respectively (Line 23 to 24). If the element is not found in any of these queues, it is added to the partner queue q_p (if it is a partner), or to the non-partner queue $L - 1$ (if it is a non-partner process), as shown in Lines 19 and 29, respectively. Each time an element is added to the non-partner queue $L - 1$, its queue length is compared to the threshold value, t. If the queue length is greater than the threshold, some new partners are then identified. We use the same procedure presented in Algorithm 1 for selecting the partners (Line 31). The procedure of identifying the new partners is continued until we are limited by the size of the hash table.

Algorithm 3 Message matching design in the dynamic approach (Search UMQ, if not found add a new QE to PRQ)

Input: The hash table HT, Number of levels L, The partner queues (pq_{Q_0}...pq_{Q_L-1}) and non-partner queues (npq_0...npq_{L-1}), The searching element (Context-id, rank, tag), The UMQ length QL, the threshold t

Output: The element found or added to the queue QE

```
1:  Search the initial queue
2:  if element found then
3:      Return QE
4:  else
5:      if L ≥ 0 then
6:          Generate the search_key from rank and context_id
7:          hash_table_index = hash_function(search_key)
8:          < hash_key, hash_value_1, hash_value_2 > ⇐ HT_Hash_table_index
9:          if hash_key == search_key then
10:             pq_P = hash_value_1
11:             level-of-partnership_P = hash_value_2
12:             for i ∈ 0...level_of_partnership_P do
13:                 Search the non-partner queue i
14:             end for
15:             Search the partner queue pq_P
16:             if element found then
17:                 Return QE
18:             else
19:                 Generate QE and add it to partner queue pq_P
20:                 Return QE
21:             end if
22:         else
23:             for i ∈ 0...L − 1 do
24:                 Search the non partner queue i
25:             end for
26:             if Element found then
27:                 Return QE
28:             else
29:                 Generate QE and add it to npq_L−1
30:                 if QL > t then
31:                     Select new partners and add them to the hash table using Algorithm 1
32:                 end if
33:                 Return QE
34:             end if
35:         end if
36:     else
37:         Generate QE and add it to initial PRQ
38:         if QL > t then
39:             Select partners and add them to the hash table using Algorithm 1
40:         end if
41:         Return QE
42:     end if
43: end if
```

It should be mentioned that there is a small, fixed cost associated with searching the hash table in the proposed partner/non-partner message matching architecture. To compensate for this cost, the queue length threshold parameter, t, should be selected wisely so as to improve the performance for long list traversals while delivering the same performance as the linked list data structure for short list queues. In the experimental results, we show the impact of different queue length thresholds.

In regard to wildcard communication, we note that the MPI_ANY_TAG wildcard is automatically supported in both the static and dynamic approaches, as all the elements from each given source are stored in the same linked list and in the order of their arrival. To deal with MPI_ANY_SOURCE wildcard communication, a sequence number is added to each queue element. Requests bearing MPI_ANY_SOURCE are allocated to a separate PRQ called PRQ_ANY. When searching an element in the posted receive queue is required, both the posted receive queue derived from partner/non-partner message matching mechanism and the PRQ_ANY queue are searched and the element whose sequence number is smaller is chosen as the matching element. Similarly, when searching UMQ with MPI_ANY_SOURCE as the source is required, all the partner and non-partner queues will be searched, and the element with the smallest sequence number will be selected as the matching element.

5 | COMPLEXITY ANALYSIS

5.1 | Runtime complexity

Message matching consists of three main operations: insertion, deletion, and search. Insertion has an $\mathcal{O}(1)$ complexity in all the three designs (linked list, Open MPI queue data structure, and the propose partner/non-partner design). For the linked list, this is achieved by storing the tail of the linked list and inserting the elements at the end of the list. The same is true for insertion into a specific linked list in the array-based and partner/non-partner designs. Finding the target linked list in these designs though involves certain computations, but such computations are actually incurred in a search operation that precedes each insertion. More specifically, insertion into the UMQ precedes by a search in the PRQ, and vice versa. The deletion operation will also have an $\mathcal{O}(1)$ complexity in all the three designs because it always happens after the search operation at the position where a queue item is found.

5.1.1 | Linked list message queue

In order to discuss the search complexity, we will use the parameters defined in Table 1. As discussed in Section 2, the search complexity for the linked list data structure can be given by Equation (3) as we need to search through all the elements in the queue

$$\mathcal{O}(q). \tag{3}$$

In order to compare the search complexity of the linked list data structure with Open MPI and the proposed partner/non-partner message queue design, we will present the search complexity based on the number of elements from each rank and context-id. Equation (4) shows that the total number of elements in the queue is the sum of the number of elements from all context-ids

$$q = \sum_{k=0}^{K-1} q_k. \tag{4}$$

The number of elements in the queue from context-id k can be derived from Equation (5). This equation shows that the number of elements in the queue with context-id k is the sum of the number of elements from all its ranks

$$q_k = \sum_{r=0}^{r_k-1} q_{kr}. \tag{5}$$

TABLE 1 List of parameters and their definition

q	Total number of elements in the queue
q_k	Number of queue elements with context-id k
q_{kr}	Number of queue elements with context-id k and rank r
K	Number of active context-ids at a given process
r_k	Number of ranks in context-id k
L	Total number of levels
n_{lkr}	Number of non-partner queue elements with context-id k and rank r at level l
p_{lkr}	Number of partner queue elements with context-id k and rank r at level l
l_{kr}	Level at which the process with rank r in context-id k becomes a partner
a_k	The number of any_source queue items associated with the communicator k
R	Maximum size of the communicators
N	Total number of processes
P	Total number of partner processes

By substituting Equation (5) into Equation (4), we have

$$q = \sum_{k=0}^{K-1} \sum_{r=0}^{r_k-1} q_{kr}. \tag{6}$$

By substituting Equation (6) into Equation (3), the search complexity based on the number of elements from each rank and context-id will be

$$\mathcal{O}\left(\sum_{k=0}^{K-1} \sum_{r=0}^{r_k-1} q_{kr}\right). \tag{7}$$

In the case of MPI_ANY_SOURCE wildcard communication, such elements in the queue should be traversed as well. Therefore, the time complexity is given by Equation (8)

$$\mathcal{O}\left(\sum_{k=0}^{K-1} \sum_{r=0}^{r_k-1} (q_{kr} + a_k)\right). \tag{8}$$

5.1.2 | Open MPI message queue

In the array-based design, we first search the linked list of context-ids in $\mathcal{O}(K)$, then use the array to access the dedicated linked list for rank r in $\mathcal{O}(1)$ and scan through it in $\mathcal{O}(q_{kr})$. Thus, its total search complexity is $\mathcal{O}(K + q_{kr})$. For long message queues, the number of context-ids is negligible compared to the total number of queue elements and the search complexity can be given by Equation (9)

$$\mathcal{O}(q_{kr}). \tag{9}$$

In the presence of MPI_ANY_SOURCE wildcard communication, the posted receive queue keeps such messages in a separate linked list queue, PRQ_ANY queue, as their rank field cannot be used as an array index. Therefore, we need to traverse the PRQ_ANY queue for any incoming messages on top of the linked list associated with the rank from the message envelope. The PRQ traversal complexity in the presence of MPI_ANY_SOURCE is therefore given by Equation (10)

$$\mathcal{O}(q_{\mathrm{kr}} + a_{\mathrm{k}}). \tag{10}$$

When an MPI_ANY_SOURCE receive call is posted, all the UMQ elements associated with all the ranks must be searched. The UMQ traversal complexity in the presence of MPI_ANY_SOURCE is given by Equation (11)

$$\mathcal{O}\left(\sum_{r=0}^{r_k-1} q_{kr}\right). \tag{11}$$

5.1.3 | Partner/Non-partner message queue

In the proposed partner/non-partner design, the search operation starts with generating a hash key for the target element based on its corresponding context-id and rank. To this end, we first map the context-id to an integer value between 0 and the number of communicators. Then, the perfect hash function in the static approach or the round-robin hash function in the dynamic approach is used to generate the hash key. The context-id map takes $\mathcal{O}(K)$ in our current design as we need to traverse an array of size K. Using the hash key, we query the hash table in $\mathcal{O}(1)$ to find out whether the target element belongs to a partner process or not. If the source process is a non-partner process, we will scan each non-partner queue in increasing order of their level number $\mathcal{O}(K + \sum_{l=0}^{L-1} \sum_{k=0}^{K-1} \sum_{r=0}^{r_k-1} n_{lkr})$. However, if it is a partner, we first scan the non-partner queues at Levels 0 to l_{kr}. After that, we jump to the dedicated partner queue of r to proceed with search $\mathcal{O}(K + (\sum_{l=0}^{l_{kr}} \sum_{k=0}^{K-1} \sum_{r=0}^{r_k-1} n_{lkr}) + p_{lkr})$. For long message queues, the number of communicators K is negligible compared to the number of queue elements and the search complexity for non-partner and partner process can be given by Equations (12) and 13, respectively. We should note that these equations apply to both the static and dynamic approaches. However, there is only one level in the static approach and, hence, the summations corresponding to multiple levels in Equation (12) and Equation (13) will drop

$$\mathcal{O}\left(\sum_{l=0}^{L-1} \sum_{k=0}^{K-1} \sum_{r=0}^{r_k-1} n_{lkr}\right) \tag{12}$$

$$\mathcal{O}\left(\left(\sum_{l=0}^{l_{kr}} \sum_{k=0}^{K-1} \sum_{r=0}^{r_k-1} n_{lkr}\right) + p_{lkr}\right). \tag{13}$$

Note that the total number of elements q in the partner/non-partner message queue design is the sum of the elements for all partner and non-partner processes. In other words,

$$q = \sum_{l=0}^{L-1} \sum_{k=0}^{K-1} \sum_{r=0}^{r_k-1} n_{lkr} + \sum_{l=0}^{L-1} \sum_{k=0}^{K-1} \sum_{r=0}^{r_k-1} p_{lkr}. \tag{14}$$

This shows that the partner/non-partner message queue design can reduce the number of traversals from $\mathcal{O}(q)$ in the linked list data structure (Equation (3) to Equation (12) and Equation (13)) for non-partner and partner processes, respectively.

In the presence of MPI_ANY_SOURCE wildcard communication, the posted receive queue, PRQ_ANY, dedicated to such messages should be searched as well. Therefore, the PRQ traversal complexity for a non-partner and partner process in the presence of MPI_ANY_SOURCE is given by Equation (15) and Equation (16), respectively

$$O\left(\sum_{l=0}^{l=L-1}\sum_{k=0}^{K-1}\sum_{r=0}^{r_k-1} n_{lkr} + \sum_{k=0}^{K-1} a_k\right) \tag{15}$$

$$O\left(\left(\sum_{l=0}^{l_{kr}}\sum_{k=0}^{K-1}\sum_{r=0}^{r_k-1} n_{lkr}\right) + p_{lkr} + \sum_{k=0}^{K-1} a_k\right). \tag{16}$$

When an MPI_ANY_SOURCE receive call is posted, all the UMQ elements associated with all the ranks must be searched. The UMQ traversal complexity in the presence of MPI_ANY_SOURCE is given by Equation (17)

$$O\left(\sum_{l=0}^{L-1}\sum_{k=0}^{K-1}\sum_{r=0}^{r_k-1} n_{lkr} + \sum_{l=0}^{L-1}\sum_{k=0}^{K-1}\sum_{r=0}^{r_k-1} p_{lkr}\right). \tag{17}$$

5.2 | Memory overhead complexity

The total amount of memory required for each message queue data structure consists of two parts: the memory required to store the message queue elements and the additional memory required to maintain the data structure, which we refer to it as the memory overhead of the data structure. The required memory for message queue elements is the same for all three designs (linked list, Open MPI queue data structure, and the proposed partner/non-partner design). Therefore, in this section, we just discuss the memory overhead complexity of the three designs.

5.2.1 | Linked list message queue

The MVAPICH message queue design does not impose any memory overhead as it only stores message queue elements in a linked list. Therefore, the memory overhead complexity can be given by Equation (18)

$$\mathcal{O}(1). \tag{18}$$

5.2.2 | Open MPI message queue

In the Open MPI queue design, the memory overhead consists of two parts: (1) the linked list of communicators/context-ids and (2) one array of size R per communicator. Thus, its memory complexity is given by Equation (19)

$$\mathcal{O}(K \times R). \tag{19}$$

Note that in case of an application having only a single communicator (MPI_COMM_WORLD) $K = 1$, the total number of processes N is equal to number of ranks R in MPI_COMM_WORLD. In this case, the memory overhead complexity of Open MPI queue design is minimized and Equation (19) is equal to $\mathcal{O}(N)$.

5.2.3 | Partner/Non-partner message queue

The partner/non-partner design imposes some memory overhead to store the information about the partner processes in a hash table. The number of entries in the hash table is proportional to the number of extracted partners. Therefore, the memory complexity overhead of the partner/non-partner design will be $\mathcal{O}(P)$. Thus, our approach does not have the fixed and unscalable memory overhead of the Open MPI queue data structure.

The number of partner processes depends on the application characteristics. In the worst case, all the processes will become a partner which results in the same memory consumption as the Open MPI queue design. However, we bound the number of partners (hash table entries) in the partner/non-partner design so as to guarantee a better memory scalability than the Open MPI queue design.

For choosing the cap for the number of queues, we considered the memory consumption in linked list and Open MPI queue data structure. As discussed in Section 5.2.1, the inked list data structure has scalable memory consumption and its memory overhead is $\mathcal{O}(1)$. However, it is not scalable in terms of speed of operation. On the other hand, Open MPI queue data structure is faster than linked list but its memory overhead is at least $\mathcal{O}(N)$ (Section 5.2.2). In partner/non-partner message queue design, we take an in-between approach and bound the number of queues for partner processes to $c \times \sqrt{N}$, where c denotes a constant factor used to evaluate the impact of increasing the memory cap on message matching performance. This will result in a memory complexity overhead of $\mathcal{O}(c \times \sqrt{N})$. Considering c as a constant parameter, the memory overhead complexity can be given by Equation (20) for our partner/non-partner design. Bounding the number of partners will provide a trade-off between memory and performance. Extracting more partners will benefit the execution time due to providing more dedicated queues. However, more partners will consume more memory in the partners hash table

$$\mathcal{O}\left(\sqrt{N}\right). \tag{20}$$

6 | PERFORMANCE RESULTS AND ANALYSIS

We evaluate the efficiency of our proposed approach by comparing it against the linked list data structure used in MVAPICH. The evaluation is done with three MPI applications: AMG2006, version 1.0,[13] LAMMPS, version 14May16,[14] and FDS, version 6.1.2.[11] AMG2006 is a parallel algebraic multigrid solver for linear systems arising from problems on unstructured grids. LAMMPS is a classical molecular dynamics code that can be used for solid-state materials (metals, semiconductors), soft matter (biomolecules, polymers), and coarse-grained or mesoscopic systems. We use the rhodopsin protein benchmark for LAMMPS in our experiments. FDS is a computational fluid dynamics model of fire-driven fluid flow. It numerically solves a form of the Navier-Stokes equations appropriate for low-speed, thermally-driven flow with an emphasis on smoke, and heat transport from fires.

We should note that we consider these three applications due to their different message queue behavior. For example, FDS has long list traversals especially at process 0. On the other hand, AMG2006 and LAMMPS are well-designed applications that have short and medium message queue traversals. We present the results for these three applications to show the efficiency of the proposed approach for well-crafted applications with short and medium traversals and applications with considerably large number of traversals.

The evaluation was conducted on the General Purpose Cluster (GPC) at the SciNet HPC Consortium of Compute Canada. GPC consists of 3780 nodes, for a total of 30240 cores. Each node has two quad-core Intel Xeon sockets operating at 2.53GHz and a 16GB memory. We have used the QDR InfiniBand network of the GPC cluster. The MPI implementation is MVAPICH2-2.2.

6.1 | Selecting the partners

The first step toward the efficiency of the partner/non-partner message queue design is to choose the partners efficiently. Choosing a low communicating process as partner would result in allocating unnecessary queues, which degrades the memory consumption. On the other hand, if a high communicating process is not selected as partner, it would weaken the efficiency of the partner/non-partner approach. In this section, we provide some experimental results to show that the partners are selected appropriately in our design.

Figure 4 shows the selected partners using the average metrics in AMG2006 and LAMMPS applications, respectively. The processes are selected using the dynamic approach with the queue length threshold equal to 100. The black data points are the processes that are selected as partners while the white data points are non-partner processes. Comparing Figure 1 with the partners selected in Figure 4, it is obvious that most of the high communicating processes are correctly selected as partners. Note that we have observed almost the same results with other threshold values and metrics.

We should note that in the FDS application, process 0 is the only process that has a significant number of communications; other processes send only a few number of elements to the queues, so no or a few partners are in fact selected for such processes. Therefore, we avoid presenting a similar figure for the FDS application as it would not convey much information due to containing mostly white data points, except for the first row.

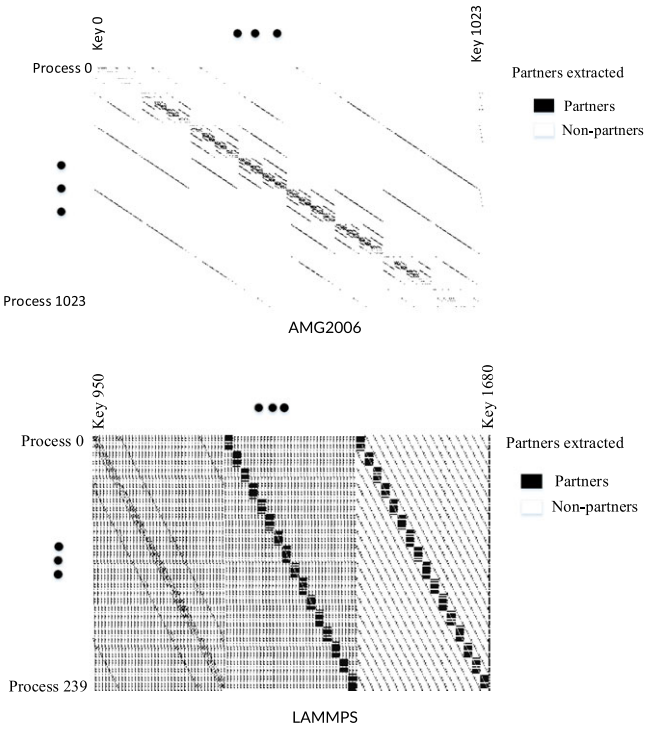

FIGURE 4 Selected partners using the average metric in AMG2006 and LAMMPS with threshold = 100 in the dynamic approach

6.2 | Queue search time

In this section, we compare the queue search time of the linked list data structure with the partner/non-partner message queue design for the FDS, AMG2006, and LAMMPS applications. Figure 5 shows the PRQ and UMQ search time speedup over the linked list for these applications when the average, median, and outliers (with $\alpha = 0$) metrics are used for selecting the partners. The results for the dynamic approach are shown with different threshold values, *t*, from 25 to 1600.

The first observation from Figure 5 is that, in almost all cases, the static approach provides a better or similar performance compared to the dynamic approach. This complies with the general intuition that the static approach should always provide a better performance due to having a global view of application's message queue characteristics. However, as shown in the figure, the dynamic approach can lead to better results in some cases due to a more aggressive partner extraction. This is because the dynamic approach decides about partner extraction with respect to (multiple) short-term queue status. On the other hand, the static approach considers the overall communication pattern based on which fewer processes might appear to be partners in some cases. Another observation is that all the studied metrics (average, median, and outliers) could successfully select the partners and none of them has a significant advantage over the others.

Figure 5A and Figure 5B show the average PRQ and UMQ search time over all processes for AMG2006, respectively. Figure 5A shows that we can reach up to 2.32x speedup in PRQ search time for AMG2006. However, as can be seen in Figure 5B, the UMQ speedup is not considerable, and with

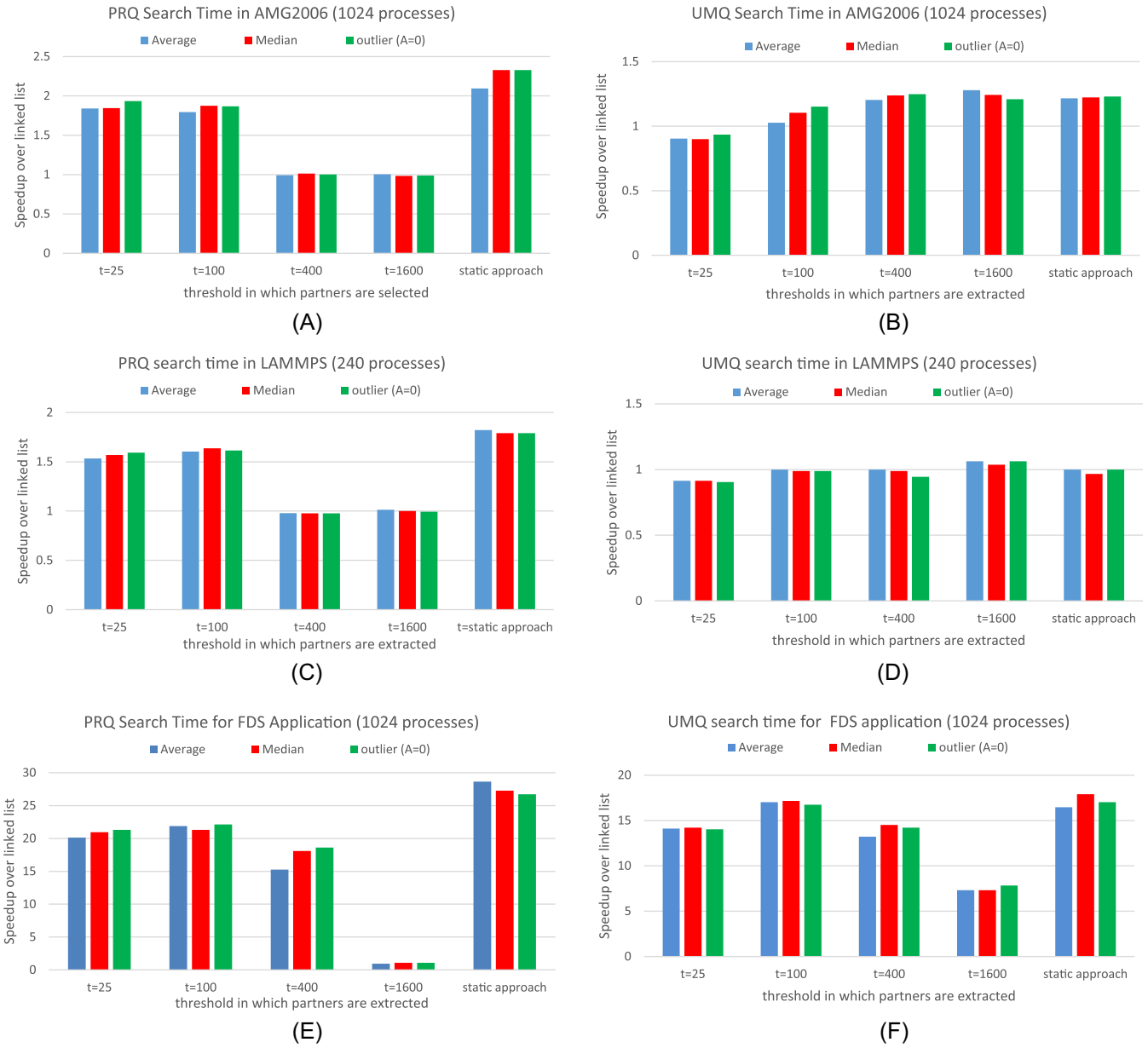

FIGURE 5 PRQ and UMQ search time speedup over linked list in AMG2006, LAMMPS, and FDS applications. A, PRQ search time speedup in AMG2006; B, UMQ search time speedup in AMG2006; C, PRQ search time speedup in LAMMPS; D, UMQ search time speedup in LAMMPS; E, PRQ search time speedup in FDS; F, UMQ search time speedup in FDS

inappropriate t ($t = 25$), the speedup would be even less than 1. The reason for this lies in the short list traversal of the UMQ for many processes in AMG2006, which does not compensate for the overhead of the hash table lookup time when t is small. On the other hand, if the queue length, t, is considered very large, some processes do not build up such a lengthy queue, and so they would not extract any partners, which results in queue search time degradation. For example, as can be seen in Figure 5A with $t \geq 400$, none of the processes selects any partner and the speedup is around 1. Figures 5C and 5D show almost the same results for LAMMPS. We should note that LAMMPS generates longer queues with 240 processes than 1024 processes. Therefore, we show the results with 240 processes.

Figure 5E and Figure 5F show the average PRQ/UMQ search time for the FDS application. In this application, the majority of communications is done with process 0. Therefore, we show the PRQ and UMQ search time for process 0. As can be seen in these figures, we can gain up to 28x queue search time speedup for this application. Increasing t from 25 to 100 would result in higher improvement in the PRQ/UMQ search time. The reason is that larger t values would provide more profiling information for selecting the partners. However, if t is too large (eg, $t = 1600$), only a few partners are selected, which results in less performance improvement.

We should note that FDS does not use MPI communication calls efficiently. In this application, each process sends a number of messages to process 0 through MPI_Send operation, and process 0 receives these messages with blocking MPI_Recv calls. In addition, MPI_Gather(v) is also used inefficiently. This causes building up large message queues at process 0. Therefore, FDS is not an optimized code and that this level of improvement should not be expected for highly-optimized codes. However, we include the FDS application in our study to present the maximum performance that could be potentially gained through our proposed message matching technique. Moreover, the FDS results provide this opportunity to indirectly compare the performance gain of the proposed approach with other message matching mechanisms that use this application.[6,9]

Figure 5 shows that $t = 100$ can provide reasonably good performance in almost all cases. We might get a better performance with $t > 100$, but some applications do not build up such a lengthy queue (Figure 5C). On the other hand, smaller t values might induce overhead for short list traversals (Figure 5D). Therefore, the value of t should simply represent the threshold at which a linked list traversal becomes too costly in terms of message progression. This can be experimentally measured for each application on a given target system.

6.3 | Application execution time

Figure 6 compares the FDS execution time using the partner/non-partner message queue against the linked list for 1024 and 2048 processes and under different metrics and queue length thresholds. As can be seen, the proposed design can improve the application runtime by up to 2.4x for 1024 processes and 5x for 2048 processes, respectively.

Figure 6 shows the execution time speedup when the size of the hash table is unbounded, so-called the *unbounded memory* case. This way, the application can add as many partners as it can find to the hash table. The problem with having too many partners is that it increases the memory consumption. In worst case, the number of partners for each process would be equal to the total number of processes in all communicators and the memory consumption would converge to the Open MPI data structure. In order to achieve a sublinear memory complexity as discussed in Section 4.3 and Section 5.2.3, we restrict the number of elements in each entry of the hash table to c in the dynamic approach. We call this the *bounded memory* case. Figure 7 shows the results for 1024 and 2048 processes, respectively. Because the different partner extraction metrics provide almost the same performance improvement in Figure 5 and Figure 6, we only show the results for the average metric in Figure 7. The results in this figure suggest that the speedup increases with the increasing values for c, except with a much lesser extent for the case of t equal to 1600. The reason is that, as we use larger values for c, the size of the hash table increases and that would allow having more partners/queues.

We do not present the runtime results for AMG2006 and LAMMPS because their queue search time improvements do not translate to a significant improvement in their application runtime. To investigate the reason behind this, we looked into the number of times the UMQ and PRQ are searched in the applications studied in this paper. Table 2 confirms that there is a sharp contrast between the number of queue searches for AMG2006 and LAMMPS and that of FDS, which could translate to application performance only for FDS.

FIGURE 6 FDS application runtime speedup over linked list with unbounded memory

FIGURE 7 FDS application runtime speedup over linked list with bounded memory and using average as the partner extraction metric

TABLE 2 Maximum number of (UMQ/PRQ) queue searches in the applications

Application	Number of UMQ Searches (k)	Number of PRQ Searches (k)
AMMPS	46,673	103,613
AMG2006	99,929	136,665
FDS	6,825,587	9,706,370

6.4 | Number of the levels and partners

As mentioned earlier, the dynamic approach finds the partner processes in multiple levels and that allows capturing the dynamics of the applications. Figure 8 presents the number of levels in the FDS application with 2048 processes. The results show that, as we decrease the queue length threshold, t, the queue length reaches the threshold more frequently, and consequently the number of levels is increased. Another observation is that reducing c fills out the hash table sooner, which results in reducing the number of levels.

Figure 9 presents the total number of selected partners for all processes in the FDS application for 1024 and 2048 processes when $t = 100$. The number of partners depends on the size of the hash table and the parameter c. As long as the hash table is not full, the partners are extracted independent of the threshold, t. We have observed similar results with other t values.

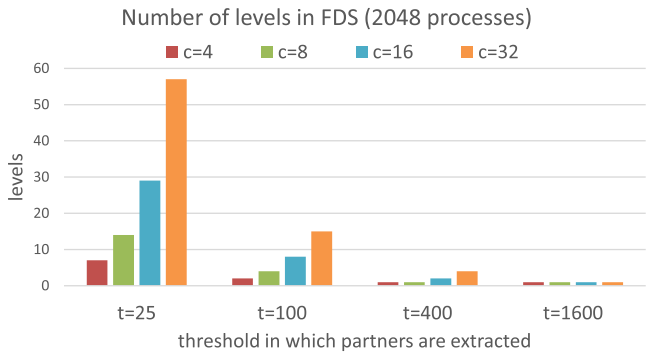

FIGURE 8 Number of levels in FDS application for 2048 processes

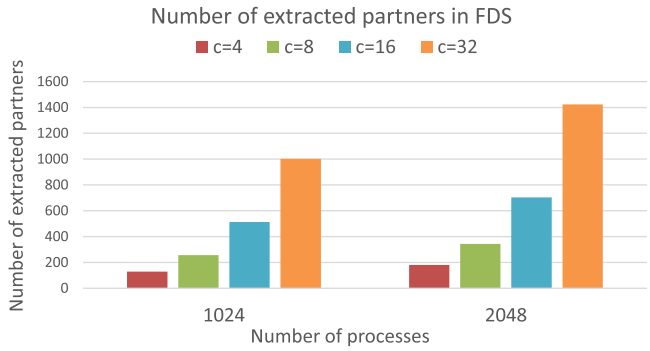

FIGURE 9 Number of extracted partners in FDS application for different number of processes and $t = 100$

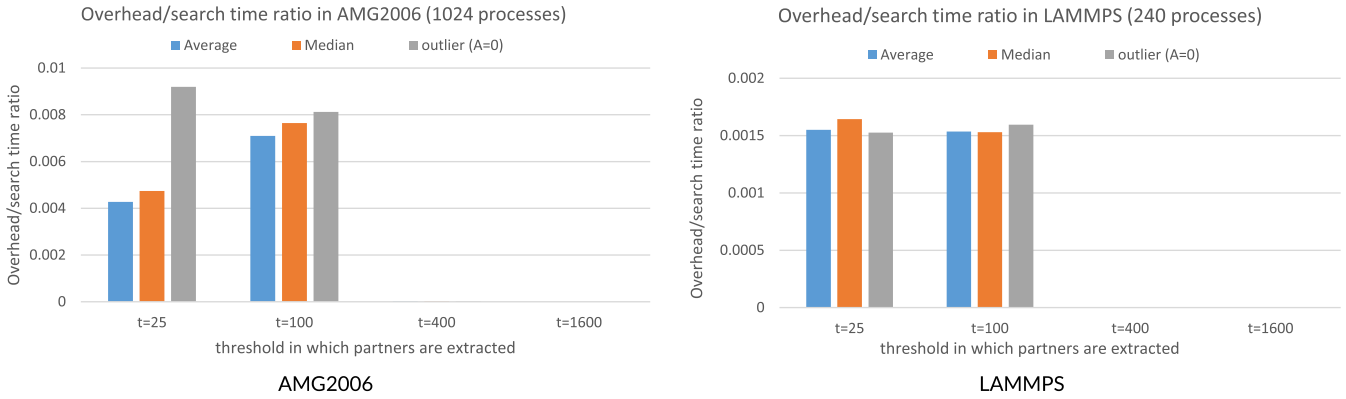

FIGURE 10 Partner extraction overhead/search time ratio

6.5 | Overhead of extracting partners

The partner/non-partner message queue imposes some overhead when partners are extracted. In the static approach, the partners are selected in the profiling stage, so there is no overhead in the future runs. However, in the dynamic approach, partners are selected during the application runtime, and therefore the overhead should be quantified. Figure 10 shows the ratio of the average overhead for partner extraction over the average queue search time across all processes in AMG2006 and LAMMPS. It is clear that the overhead of extracting partners is negligible compared to the queue search time. For $t \geq 400$, the overhead is close to 0.

We should note that there is also some overhead involved in searching the hash table in our design. However, as searching the hash table is a part of searching the queue, the overhead is already included in the queue search results shown in Section 6.2.

7 | CONCLUSION AND FUTURE WORK

Many parallel applications expose a sparse communication pattern in which each process has more communications with a certain group of other processes. We take advantage of this feature to propose a new message matching design that adapts based on the communication traffic of the applications. To do so, we measure the frequency of the messages sent from different processes to the queues and use this information to select a list of partners for each process. Then, we allocate a dedicated message queue for the partner processes. Other processes share a single non-partner queue. We design the work using a static and a dynamic approach. The static approach works based on the information from a profiling stage while the dynamic approach uses the runtime information. Our proposed approach can successfully decrease the queue search time for long list traversals while maintaining the performance for short list traversals. Moreover, it provides better scalability in terms of memory consumption. The evaluation results show that we can speed up the queue search time and application runtime by up to 28x and 5x for applications with extreme message matching requirements, respectively. As for future work, we would like to extend our work to support multi-threaded communication and experiment with other applications and larger systems.

ACKNOWLEDGMENTS

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada under grant #RGPIN/05389-2016, and the Canada Foundation for Innovation #7154. Computations were performed on the GPC supercomputer at the SciNet HPC Consortium. SciNet is funded by the Canada Foundation for Innovation under the auspices of Compute Canada; the Government of Ontario; the Ontario Research Fund - Research Excellence; and the University of Toronto. We thank Mellanox Technologies for the resources to conduct the early evaluation of this research.

ORCID

S. Mahdieh Ghazimirsaeed http://orcid.org/0000-0001-8510-3145

REFERENCES

1. Message Passing Interface (MPI) Forum. http://www.mpi-forum.org. Accessed June 20, 2017.
2. MPICH: High-Performance Portable MPI. http://www.mpich.org. Accessed June 27, 2017.
3. MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE. http://mvapich.cse.ohio-state.edu/. Accessed June 24, 2017.
4. Open MPI: Open Source High Performance Computing. https://www.open-mpi.org. Accessed: June 23, 2017.
5. Zounmevo JA, Afsahi A. A fast and resource-conscious MPI message queue mechanism for large-scale jobs. *Futur Gener Comput Syst*. 2014;30:265-290.

6. Flajslik M, Dinan J, Underwood KD. Mitigating MPI message matching misery. *International Conference on High Performance Computing*. Cham, Switzerland: Springer; 2016:281-299.
7. Klenk B, Fröning H, Eberle H, Dennison L. Relaxations for high-performance message passing on massively parallel SIMT processors. Paper presented at: IEEE International Parallel and Distributed Processing Symposium (IPDPS); 2017; Orlando, FL.
8. Bayatpour M, Subramoni H, Chakraborty S, Panda DK. Adaptive and dynamic design for MPI tag matching. Paper presented at: 2016 IEEE International Conference on Cluster Computing (CLUSTER); 2016; Taipei, Taiwan.
9. Ghazimirsaeed M, Afsahi A. Accelerating MPI message matching by a data clustering strategy. Paper presented at: High Performance Computing Symposium (HPCS); 2017; Virginia Beach, VA.
10. Seber GAF. *Multivariate Observations*. Hoboken, NJ: John Wiley & Sons; 2009.
11. McGrattan K, Hostikka S, McDermott R, Floyd J, Weinschenk C, Overholt K. Fire Dynamics Simulator: User's Guide. 6th ed. NIST special publication. NIST; 2013.
12. Balaji P, Chan A, Gropp W, Thakur R, Lusk E. The importance of non-data-communication overheads in MPI. *Int J of High Perform Comput Appl*. 2010;24(1):5-15.
13. Yang UM. BoomerAMG: a parallel algebraic multigrid solver and preconditioner. *Appl Numer Math*. 2002;41(1):155-177.
14. Plimpton S. Fast parallel algorithms for short-range molecular dynamics. *J Comput Phys*. 1995;117(1):1-19.
15. Underwood KD, Hemmert KS, Rodrigues A, Murphy R, Brightwell R. A hardware acceleration unit for MPI queue processing. In: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05); 2005: Denver, CO.
16. Barrett BW, Brightwell R, Grant R, Hammond SD, Hemmert KS. An evaluation of MPI message rate on hybrid-core processors. *Int J High Perform Comput Appl*. 2014;28(4):415-424.
17. Barrett BW, Brightwell R, Grant RE, et al. The Portals 4.0. 2 Networking Programming Interface, Sandia National Laboratories. Technical Report. Albuquerque, NM: SAND; 2014.
18. Mellanox Technologies. Mellanox HPC-X Software Toolkit. www.mellanox.com/products/hpcx/. Accessed June 28, 2017.
19. Libfabric Programmer's Manual: Libfabric OpenFabrics. https://ofiwg.github.io/libfabric/. Accessed June 24, 2017.
20. Gropp W, Lusk E, Doss N, Skjellum A. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Comput*. 1996;22(6):789-828.
21. Brightwell R, Pedretti K, Ferreira K. Instrumentation and analysis of MPI queue times on the SeaStar high-performance network. In: 2008 Proceedings of 17th International Conference on Computer Communications and Networks; 2008; St. Thomas, VI.
22. Keller R, Graham RL. Characteristics of the unexpected message queue of MPI applications. *Recent Advances in the Message Passing Interface: European MPI Users' Group Meeting*. Berlin, Germany: Springer; 2010:179-188.
23. Brightwell R, Goudy S, Underwood K. A preliminary analysis of the MPI queue characteristics of several applications. Paper presented at: 2005 International Conference on Parallel Processing (ICPP'05); 2005; Oslo, Norway.
24. Brightwell R, Underwood KD. An analysis of NIC resource usage for offloading MPI. In: Proceedings of the 18th International Parallel and Distributed Processing Symposium; 2004; Santa Fe, NM.
25. Rodrigues A, Murphy R, Brightwell R, Underwood KD. Enhancing NIC performance for MPI using processing-in-memory. Paper presented at: 19th IEEE International Parallel and Distributed Processing Symposium; 2005; Denver, CO.
26. Underwood KD, Brightwell R. The impact of MPI queue usage on message latency. In: Proceedings of the 2004 International Conference on Parallel Processing (ICPP'04); 2004; Montreal, Canada.
27. Vetter JS, Yoo A. An empirical performance evaluation of scalable scientific applications. In: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC'02); 2002; Baltimore, MD.
28. Jenkins RR. Hash functions for hash table lookup. 1997.
29. Gettys T. Generating perfect hash function. http://www.drdobbs.com/architecture-and-design/generating-perfect-hash-functions/184404506. Accessed: May 20, 2017.

How to cite this article: Ghazimirsaeed SM, Mirsadeghi SH, Afsahi A. Communication-aware message matching in MPI. *Concurrency Computat Pract Exper*. 2018;e4862. https://doi.org/10.1002/cpe.4862