Parallel Computing 89 (2019) 102547

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

A dynamic, unified design for dedicated message matching engines for collective and point-to-point communications[☆]

S. Mahdieh Ghazimirsaeed[a,*], Ryan E. Grant[b], Ahmad Afsahi[a]

[a] ECE Department, Queen's University, Kingston, ON, Canada
[b] Center for Computing Research, Sandia National Laboratories, Albuquerque, NM, USA

ARTICLE INFO

Article history:
Received 11 December 2018
Revised 6 August 2019
Accepted 25 August 2019
Available online 05 September 2019

Keywords:
MPI
Message matching
Message queue
Collective communications
Point-to-point communications

ABSTRACT

The Message Passing Interface (MPI) libraries use message queues to guarantee correct message ordering between communicating processes. Message queues are in the critical path of MPI communications and thus, the performance of message queue operations can have significant impact on the performance of applications. Collective communications are widely used in MPI applications and they can have considerable impact on generating long message queues. In this paper, we propose a unified message matching mechanism that improves the message queue search time by distinguishing messages coming from point-to-point and collective communications and using a distinct message queue data structure for them. For collective operations, it dynamically profiles the impact of each collective call on message queues during the application runtime and uses this information to adapt the message queue data structure for each collective dynamically. Moreover, we use a partner/non-partner message queue data structure for the messages coming from point-to-point communications. The proposed approach can successfully reduce the queue search time while maintaining scalable memory consumption. The evaluation results show that we can obtain up to 5.5x runtime speedup for applications with long list traversals. Moreover, we can gain up to 15% and 94% queue search time improvement for all elements in applications with short and medium list traversals, respectively.

1. Introduction

The Message Passing Interface (MPI) is the de facto standard for communication in High Performance Computing (HPC). The processes in MPI compute on their local data while extensively communicating with each other. In this regard, one of the most important challenges in MPI implementations is the efficiency of inter-process communications that can have considerable impact on the performance of parallel applications.

There are different types of communication supported in the MPI standard such as point-to-point and collective communications. In the point-to-point communication, the sender and receiver both take part in the communication explicitly. In the collective communication, which is extensively used by MPI applications, messages are exchanged among a group of processes. In point-to-point operations and also collective operations that run on top of point-to-point communications, the messages must be matched between the sender and receiver. Modern MPI implementations, such as MPICH [1], MVAPICH [2] and Open MPI [3] separate traffic at coarse granularity, either not at all, on a per MPI communicator level or by communicator and source rank. These solutions can be improved by intelligently separating traffic into logical fine-grained message streams dynamically during program execution. On the other hand, collective operations are implemented using different algorithms. Each of these algorithms can have specific impact on the message queues. We take advantage of this feature to propose a new unified message matching mechanism that, for the first time, considers the type of communication to enhance the message matching performance.

The contributions of the paper are as follows:

- We propose a novel communication optimization that accelerates MPI traffic by dynamically profiling the impact of different types of communications on message matching performance and using this information to allocate dedicated message matching resources to collective communications. Our approach determines the number of dedicated queues dynamically during the application runtime.

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

* Corresponding author.
E-mail address: s.ghazimirsaeed@queensu.ca (S.M. Ghazimirsaeed).

https://doi.org/10.1016/j.parco.2019.102547
0167-8191/© 2019 Elsevier B.V. All rights reserved.

- We extend our message matching architecture by using the partner/non-partner message queue design [4] for point-to-point communications alongside the proposed collective engine in a unified manner to enhance both collective and point-to-point message matching performance.
- We conduct several experiments to evaluate the impact of the proposed approaches on performance gain of the collective and point-to-point elements from different perspectives. We evaluate the impact of collective message matching optimization on both collective and point-to-point elements. Moreover, we evaluate the impact of point-to-point message matching optimization on both collective and point-to-point queue elements. Finally, we show the impact of the unified collective and point-to-point message matching design on queue search time. We demonstrate that our unified approach accelerates the collective and point-to-point queue search time by up to 80x and 71x, respectively, and that it achieves a 5.5x runtime speedup for a full application over MVAPICH.
- We also compare the proposed approach with Open MPI message queue data structure in which allocates a queue for each source process. The results show that we can gain scalable memory consumption and at the same time gain up to 1.14x speedup over Open MPI message queue design.

The rest of the paper is organized as follows. Section 2 provides some background information and discusses the motivation behind the work. Section 3 discusses the related works. Section 4 presents the proposed message matching approach. The experimental results are presented in Section 5. Finally, Section 6 concludes the paper and points to the future directions.

2. Background and motivation

In order to receive a message, a process must post a receive request with a communicator, rank and tag. A communicator is an identifier of a logical grouping of MPI processes, ranks are process addresses in a communicator and tags are special matching data for each message. All processes in a communicator must participate in a collective operation.

Well-known MPI implementations maintain a Posted Receive Queue (PRQ) and Unexpected Message Queue (UMQ) at the receiver side to cope with the unavoidable out-of-sync communications. When a new message arrives, the PRQ is traversed to locate the corresponding receive queue item. If no matching is found, a queue element (QE) is enqueued in the UMQ. Similarly, when a receive call is made, the UMQ is traversed to check if the requested message has already (unexpectedly) arrived. If no matching is found, a new QE is posted in the PRQ. As message queues are in the critical path of communication, the overheads of traversing them can become a bottleneck especially in applications that generate long queues. Therefore, designing an efficient message matching mechanism is very important to obtain high-performance communications.

Modern MPI libraries use different queue data structures for message matching. For example, MPICH and MVAPICH use the linked list data structure that searches linearly for the key tuple (communicator, rank, tag) in $O(n_q)$ in which n_q is the number of elements in the queue. This traversal cost makes the linked list data structure inefficient for long message queues. However, the advantage of linked list data structure is that it has a minimal memory consumption and excellent short list performance.

The linear queue structure is improved in Open MPI [3] by considering the fact that the communicator restricts the rank space and the rank restricts the tag space. Accordingly, in the Open MPI data structure, there is an array of size n for each communicator of size n. Each element of the array represents one rank and it has a pointer to a linked list dedicated to messages corresponding to that rank. The advantage of this data structure is that it is considerably faster than the linked list data structure for long list traversals. This is because the queues can be reached in $O(1)$ after finding the communicator. However, the disadvantage of this data structure is that it maintains an array of size n for each communicator of size n which leads to high memory consumption [5] especially for multi-threaded MPI communications. Newer approaches such as CH4 in MPICH [6] use more than one list. Others [7,8] also propose using multiple linked list queues. However, the problem with these approaches is that they do not determine the number of queues dynamically based on each process' message queue behavior. Our work also differs from these approaches in that it distinguishes between collective and point-to-point queue elements to dynamically allocate sufficient number of queues required for message matching. For point-to-point elements, it uses the partner/non-partner message queue design [4]. For collective elements, it dynamically profiles the impact of each collective call on message queue and uses this information to allocate sufficient number of queues for each collective.

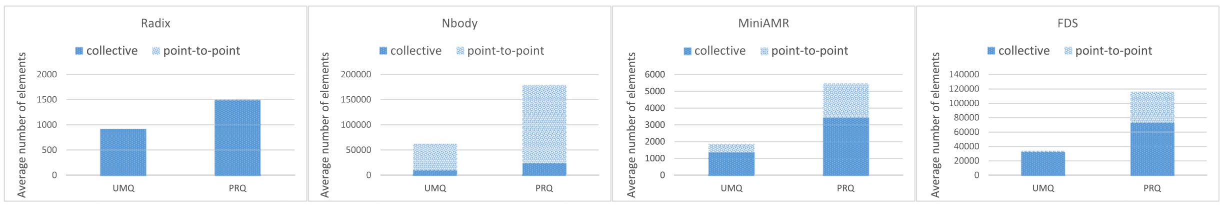
2.1. Motivation

Improving the message matching performance for collective communication operations is only useful if they have considerable contribution in posting elements to the message queues. In order to understand if improving message matching performance for collective communications is useful, we profile several applications to understand their matching characteristics. In this experiment, we count the number of elements that enter the queues from point-to-point or any non-collective communication operation. We also count the number of elements that enter the queues from collective communications. For this, we provide a hint from the MPI layer to the device layer to indicate whether the incoming message is from a point-to-point or collective communication.

Fig. 1 shows the application results for Radix [9], Nbody [10,11], MiniAMR [12] and FDS [13] with 512 processes. The descriptions of these applications are provided in Section 5.3. As can be seen in Fig. 1(a), almost all the elements that enter the queues in Radix are from collective communications. Fig. 1(b) shows that the majority of the elements that enter the queues in Nbody are from point-to-point communications but that it still has a significant number of elements from collective communications (around 11k and 25k for UMQ and PRQ, respectively). It is evident in Fig. 1(c) and (d) that both point-to-point and collective communications have contributed to the message queues in MiniAMR and FDS. However, a larger fraction of queue elements are from collective communications. In general, Fig. 1 shows that both collective and point-to-point communications can have considerable impact on the number of elements posted to the message queues. On the other hand, the list searches of greater than 1k have significant impact on message latency [14]. This shows the importance of improving the message matching performance for collective and point-to-point communications.

MPI libraries use different algorithms to implement collective operations where each of these algorithms can have a specific impact on the number of queue traversals. This motivates us to propose a message queue design that can dynamically profile the impact of the collective communications on the queues and uses that information to adapt the message queue data structure for each and every collective communication. More specifically, the first time that a collective operation is called we profile its message queue behavior. This information is used to allocate some queues for this collective on its subsequent calls.

We enhance our design by using our previously proposed partner/non-partner message queue data structure [4] for elements



(a) Radix (b) Nbody (c) MiniAMR (d) FDS

Fig. 1. Average number of elements in the queues from collective and point-to-point communications across all processes in different applications (512 processes).

coming from point-to-point communications alongside the proposed collective message queue approach [15] in a unified design. Note that the proposed collective message queue design cannot be used for point-to-point messages. The reason for this is that an individual point-to-point communication does not provide enough information for profiling the queues and adapting the queue data structure. We should iterate that the message queue mechanisms that are used in well-known MPI implementations, such as MPICH, MVAPICH and Open MPI, or are proposed in the literature [7,8,16–18] do not consider the type of communication for message matching, and therefore they keep the messages from all types of communication in a single data structure.

3. Related work

Several works have been proposed in literature to improve the message matching performance by reducing the number of queue traversals [4,7,8,16,18]. Zoumevo and Afsahi [16] proposed a 4-dimensional data structure that decomposes ranks to multiple dimensions. This way, they could skip traversing a large portion of the queue that the search is guaranteed to yield no result. This data structure has a small fixed overhead for searching any queue item which is only negligible for long list traversals.

Flajslik, et al. [7] propose a message matching mechanism that takes advantage of hash functions to group processes into multiple bins (or queues). Increasing the number of bins speeds up the search operation with the cost of more memory consumption. The problem with the bin-based approach is that it imposes some overhead for short list traversals. Moreover, it does not determine the optimal number of bins. Bayatpour, et al. [8] propose a dynamic message queue mechanism to address the overhead issue for short list traversals in the bin-based approach. In this work, the application always starts with the default linked list message queue mechanism. If the number of traversals reaches a threshold, it switches to the bin-based design. Unlike the bin-based approach [7], this work benefits both short list and long list traversals. However, it still does not determine the optimal number of bins dynamically. Our work differs from these works in that it profiles the message queue behavior of collective operations to determine the optimal number of bins for collective elements dynamically during the application runtime. Moreover, it uses the partner/non-partner message queue design [4] to dynamically determine the adequate number of queues for point-to-point communications.

Ghazimirsaeed and Afsahi [18] propose a static message matching mechanism based on K-means clustering algorithm. This work groups the processes into some clusters based on their message queue behavior and allocates a dedicated queue for each group. The problem with this design is that it is a static approach. Moreover, it is not scalable in terms of memory consumption.

Ghazimirsaeed, et al. [4] take advantage of sparse communication pattern in parallel applications to propose partner/non-partner message matching design that adapts based on the communication frequency between peer processes. Accordingly, they categorize processes into a set of *partners* and *non-partners*. Partner processes have higher frequency of communication and a dedicated message queue is allocated to each of them. On the other hand, the non-partner processes share a single queue. This way, they could reduce the queue search time for high communicating peers while providing scalable memory consumption. A hash table is used to distinguish partner processes from non-partner processes at search time. The authors provide both static and dynamic versions of their design.

Many prior works explore hardware support for message matching [6,17,19–22]. Klenk, et al. [17] use GPU features to improve the message matching performance. Underwood, et al. [19], investigate hardware designs to efficiently perform MPI message matching. Rodrigues, et al. [20] explore mechanisms to use PIM technology features to manage MPI message queues. New approaches such as CH4 in MPICH address the scalability issues in MPI and provide hardware supported message matching [6]. Portals networking API [21] and Bull's BXI interconnect [22] investigate hardware designs to perform efficient message matching. The problem with hardware approaches is that they require special hardware that are not widely available. Moreover, only a certain number of messages can be handled in hardware, typically between 1K-5K elements for modern ternary content addressable memories.

Other works examine the impact of the message queues on the performance of different MPI applications [23–26] or they evaluate MPI message matching performance [14,27,28]. For example, Barret, et al. [14] evaluate MPI matching operations on hybrid-core processors.

Our approach differs from these works in that it considers the type of communication to improve message matching performance. Moreover, unlike most of the message matching approaches it determines the number of queues dynamically during the application runtime. To this aim, the first time that a collective operation (with specific message size and communicator size) is called, we profile its message queue behavior. The profiling information is used to allocate some queues for this collective in its subsequent calls. Moreover, in contrast to our previous work [15] that used a single linked list queue for point-to-point elements, this paper uses the partner/non-partner message queue design [4] in a unified manner with the collective message matching design to evaluate the impact of the point-to-point optimization beside the collective optimization and also to further improve the matching performance.

4. The proposed unified message queue design

Fig. 2 shows the proposed unified message queue design. Whenever a new queue element is to be added to the queue, we check if the element is coming from a point-to-point or a collective communication. If the element is coming from point-to-point communication, we use the partner/non-partner message queue data structure [4], which we call it the *PNP* approach. Otherwise, if the element is coming from a collective operation, we use the proposed message queue design for collective elements, which is

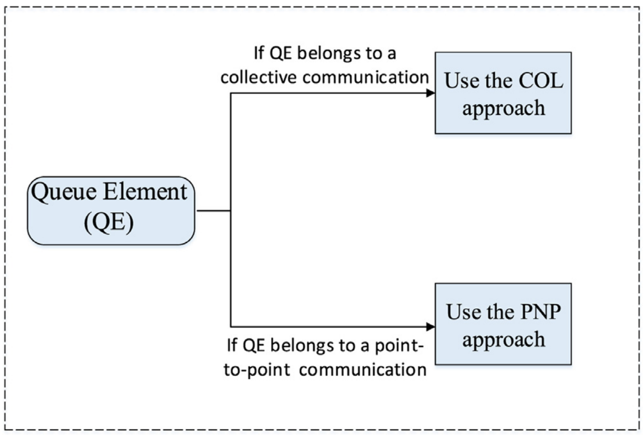

If QE belongs to a collective communication
Use the COL approach
Queue Element (QE)
Use the PNP approach
If QE belongs to a point-to-point communication

Fig. 2. The proposed unified message matching mechanism.

referred to as the *COL* approach. We refer to the proposed unified design as COL+PNP. We discuss the proposed COL message queue design for collective elements in Section 4.1. Then we briefly review the PNP message queue design in Section 4.2. Section 4.3 and 4.4 discuss the unified queue allocation and search mechanism in more details, respectively.

4.1. The COL message queue design for collective elements

There are many different algorithms (such as ring, binomial tree, fan-in/fan-out, etc.) proposed in literature or used in MPI libraries for collective operations. For each collective operation, the choice of the algorithm depends on parameters such as message size and communicator size. Each collective communication algorithm has a specific impact on the behavior of message queues. We take advantage of this feature to design a message matching mechanism that adapts itself to the impact of collective communication algorithms on message queues. Fig. 3 shows the proposed message queue mechanism for collective communications. We provide an overview of our design below.

- A runtime profiling stage is used to determine the number of queues for each collective communication operation with their specific parameters (message size and communicator size). At the profiling stage, all the collective operations share a single *profiling queue (pq)*.
- The profiling queue, pq, is only used for the first call of each collective operation with their specific parameters. After that, each collective operation generates its own set of queues.
- The queues allocated to each collective operation could be defined in multiple levels. At each level, a hashing approach based on the number of queues is used for message matching.
- A new level is used if two conditions are met: First, a similar collective operation is called but with new parameters. Secondly, the required number of queues for this collective is more than the number of the queues that are already allocated for such a collective operation.

As can be seen in Fig. 3, the profiling queue is used for the first call of each collective operation with specific parameters. The information from the profiling queue is used to determine the number of queues that are deemed sufficient to have the minimum queue search time for each collective operation. For example, in Fig. 3, q_1 number of queues are allocated for MPI_Allreduce in the first level.

If the same collective is called with different parameters, we again profile its message queue behavior to calculate the required number of queues (q_2). If q_2 was larger than q_1, it means that the queues that are currently allocated in Level 1 are not sufficient for the new collective operation. Therefore, we define a set of q_2 queues in a new level. This procedure is continued as long as the collective operation is used with the new parameters or until we are limited by the memory consumption cap. The same procedure is used for other collective operations including both blocking and non-blocking collectives such as MPI_Gather, MPI_Iallgather. Note that each collective operation uses specific tags for message matching. Therefore allocating dedicated queues for each collective operation automatically creates dedicated channels for individual tags.

For each collective communication operation, we always insert the new queue elements to the last level. For searching an element that is originated from collective communication, we always start from the profiling queue and then search the dedicated queues for the collective operation from the first level to the last level in order. This mechanism ensures that message matching ordering semantics are preserved. We explain the queue allocation and the search mechanism in more details in the following sections.

4.2. The PNP message queue design for point-to-point elements

We use the partner/non-partner message queue design [4] for the messages coming from point-to-point communications. The core idea of the design is to assign a dedicated queue for each process that sends/posts a significant number of messages to UMQ/PRQ. These processes are called *partners* and they are extracted dynamically at runtime. For extracting the partners, we count the number of point-to-point elements that each process sends to the queue. Then, we calculate the average number of elements that all processes send to the queue. Any process whose number of elements in the queue is more than the average is selected as a partner.

Fig. 4 shows the design of the partner/non-partner message queue data structure that has multiple levels. In the base level, all point-to-point elements share a single queue. When the length of this queue reaches a threshold, θ, we extract some partners and allocate dedicated message queue to each of them. The non-partner processes share a single non-partner queue. When the queue length of the non-partner queue reaches the threshold, θ, we extract some new partners in a new level. Therefore, each level has a number of partner queues and a single non-partner queue.

For searching the queues for non-partner processes, first we search the initial queue at the base level. Then, we search the non-partner queues from level 0 to $L-1$ in order. The green arrows in Fig. 4 show the order of searching the queues for a non-partner process. For searching the queues for partners, we define the term *level-of-partnership$_p$*. The level-of-partnership$_p$ shows the level in which process p becomes a partner. The order of searching the queues for partner process p is: 1) the initial queue, 2) the non-partner queues from level 0 to level-of-partnership$_p$, and 3) the dedicated queue for process p. The red arrow in Fig. 4 shows the order of searching queues for a process that becomes partner at level 0.

We use a hash table to save the partner processes, and to differentiate them from non-partner processes at search time. The hash tables maintain the dedicated queue number for partner processes and their level of partnership. As discussed earlier, to maintain memory scalability, we bound the total number of dedicated queues in both the COL and PNP approaches to $k \times \sqrt{n}$, where k is an environment variable. Note that the number of allocated queues is the main memory overhead in both the COL and PNP approaches.

For searching an element, first we lookup the hash table. If the element is not in the hash table, it means that the element is coming from a non-partner process so we search the non-partner

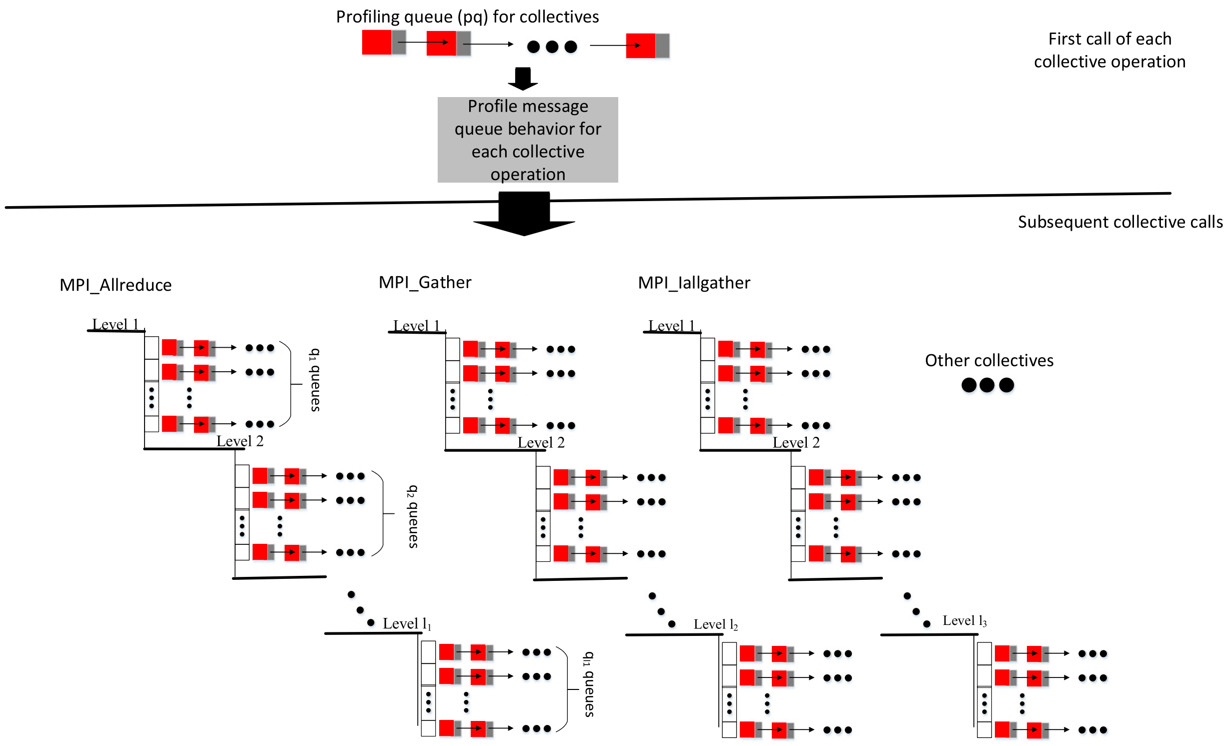

Profiling queue (pq) for collectives
First call of each collective operation
Profile message queue behavior for each collective operation
Subsequent collective calls
MPI_Allreduce
MPI_Gather
MPI_Iallgather
Other collectives
Level 1
Level 2
Level l_1
Level l_2
Level l_3
q_1 queues
q_2 queues
q_{l1} queues

Fig. 3. The proposed message matching mechanism for collective elements.

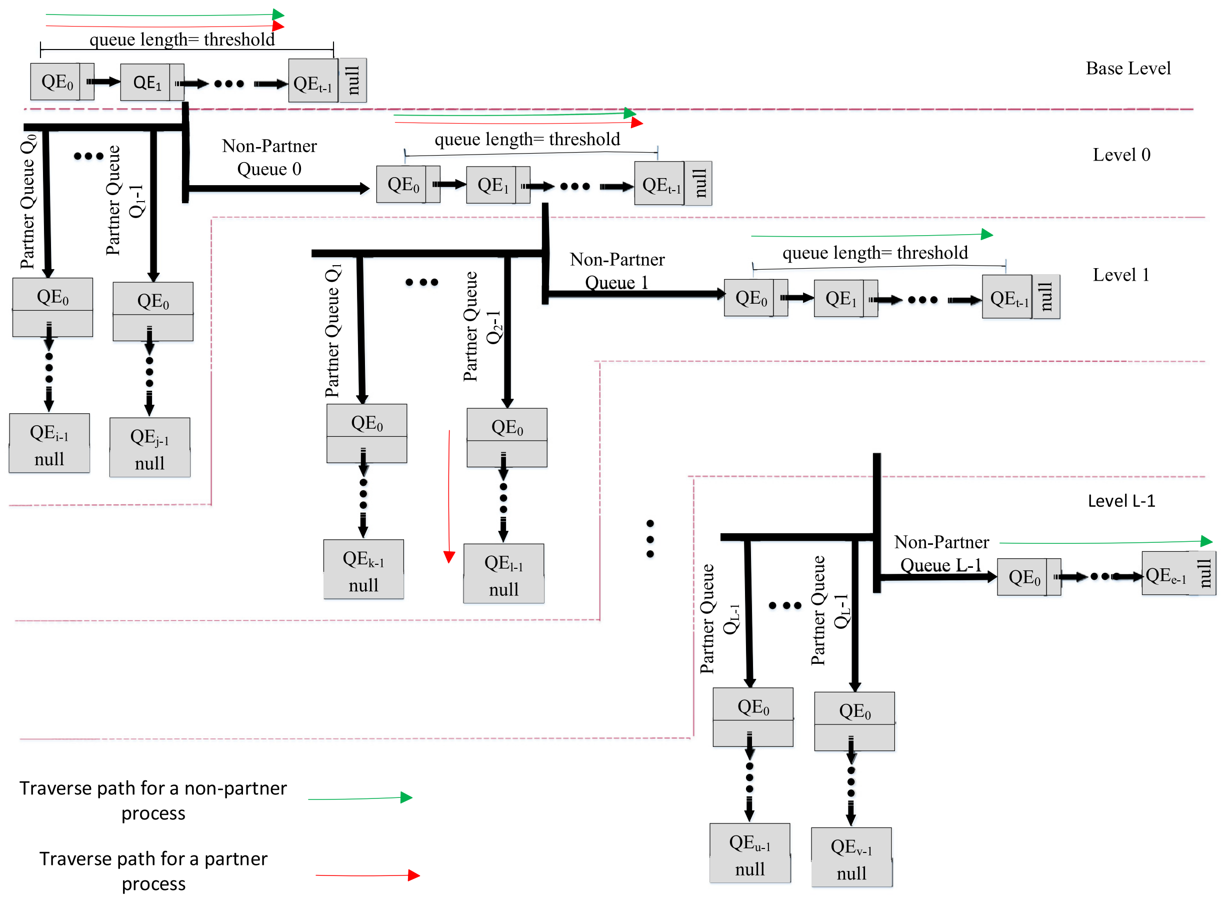


Fig. 4. Partner/non-partner message queue design for point-to-point queue elements.

queues in order. Otherwise, if the element is in the hash table, it means that it is coming from a partner process, so we derive its dedicated queue number and level of partnership and use this information to search the queues accordingly. For a detailed description of the partner/non-partner design, we refer the interested reader to our earlier work [4].

4.3. The unified queue allocation mechanism for collective and point-to-point elements

Algorithm 1 shows the detailed description of the queue allocation mechanism in the proposed design. Table 1 lists the parameters that are used as inputs and outputs of the algorithm and provides their definitions.

Algorithm 1: The unified queue allocation mechanism for collective and point-to-point elements.

```
Input: The communication type ($t$), The collective operation parameters ($p$), The queue for profiling collective communications ($pq$), Total number of the allocated queues ($T$), Number of levels for collective operation $c$ ($l_c$), The number of queues that are allocated for collective operation $c$ in the last level ($nq_c$), Total number of processes ($n$), The memory consumption cap parameter ($k$)
Output: The set of queues generated in the last level of collective operation $c$ ($q_c$)
1  if $t$==point-to-point then
2      Add the element to $Q_{PNP}$;
3  else
4      if is_profiled(p)==0 then
5          $P_p$= profile($pq$);
6      else
7          if is_q_allocated(p)==0 then
8              $nq$=calcul_num_queues($P_p$);
9              if $nq + T < k \times \sqrt{n}$ then
10                 if $nq > nq_c$ then
11                     Generate queues $q_c[0 \dots nq-1]$;
12                     $T = T + nq$;
13                     $l_c++$;
14                     $nq_c = nq$;
15                 end
16             end
17         end
18     end
19 end
```

In the unified algorithm, we first check the type of communication. If it is a point-to-point communication, we add the element to the partner/non-partner message queue data structure in Line 2. Otherwise, when a collective communication is executed, we call the function *is_profiled(p)* (Line 4). This function determines if the collective operation with specific message size and communicator size has already been profiled or not. If it has not been profiled, we profile its message queue behavior and save the profiling information in P_p (Line 5). We use the average number of queue traversals as the profiling information since it is the critical factor that determines the cost of message matching [8]. If the collective operation has already been profiled, we call the function *is_q_allocated(p)* in Line 7. This function determines if queues have already been allocated for the collective operation with this specific message size and communicator size range. If queue is not allocated, we call the function *calcul_num_queues* (P_p) in Line 8. This function gets the profiling information gathered in the previous call of the collective operation and returns the required number of queues (nq).

Table 1
List of parameters used for collective queue allocation and search mechanism.

p	Collective operation parameters (the type of collective operation, its message size and communicator size)
pq	The profiling queue for collective operations
T	The total number of dedicated queues
l_c	The number of levels for collective operation c
nq_c	Number of queues for collective c in the last level
n	Total number of processes
k	Memory consumption cap parameter
q_c	The set of queues in the last level for collective c
t	The type of communication
SE	The searching element
Q_{PNP}	The queue data structure for point-to-point elements
nq_{cl}	The number of queues that are allocated for collective operation c at level l
q_{cl}	The set of queues that are allocated for collective operation c at level l
QE	The matched element in the queue

In the best-case scenario, the average number of traversals to find an element is one. For this to happen, the number of queues should be equal to the average number of traversals. However, this may come at the expense of large memory allocation if the number of traversals is significant. Therefore, we limit the total number of queues allocated for all collective and point-to-point operations. For choosing the cap for the number of queues, we considered the memory consumption in MPICH and Open MPI. MPICH provides scalable memory consumption but it allocates only one queue for message matching, resulting in poor search performance for long list traversals. On the other hand, Open MPI is faster than MPICH for long match lists but it has unscalable memory consumption as it allocates n queues for each process. In our design, we take an in-between approach and bound the total number of queues that are allocated in the COL and PNP approaches to $k \times \sqrt{n}$. k is an environment variable to evaluate the impact of increasing the memory cap on message matching performance.

If the number of queues, nq, plus the total number of queues, T, that are already allocated, was less than $k \times \sqrt{n}$ (Line 9), it means that we are still allowed to allocate the new queues, and so we will check the second condition in Line 10.

The second condition compares nq with the number of queues that are currently allocated for collective operation c in the last level (nq_c). If nq was less than nq_c, there is no need to define a new level and allocate a new set of queues as nq_c number of queues is sufficient for this collective. However, if nq was greater than nq_c, the new set of queues should be allocated in a new level (Line 11). Finally, we update T, l_c and nq_c in Line 12 to 14.

4.4. The unified search mechanism for collective and point-to-point elements

Algorithm 2 shows the search mechanism in the proposed unified message queue design. A brief description of the inputs and outputs of the algorithm is provided in Table 1. The inputs of this algorithm are as follows: the type of communication (t), whether it is point-to-point or collective. If the communication was collective, the parameter c determines the type of collective operation. The parameters (c and t) are ported from the MPI layer to the device layer. Other inputs of the algorithm are the searching element (SE) which is the tuple rank, tag and communicator, the queue data structure for point-to-point elements (Q_{PNP}), the profiling queue for collective operations (pq), the number of levels for collective operation c (l_c), and the number of queues that are allocated for collective operation c at level l (nq_{cl}). The output of the algorithm is the search result (QE). If the element is not found, QE will be null.

Algorithm 2: The unified queue search mechanism for collective and point-to-point elements.

```
Input: The communication type ($t$), The collective operation
       ($c$), The searching element ($SE$), The queue data
       structure for point-to-point elements ($Q_{PNP}$), The
       queue for profiling collective communications ($pq$),
       The number of levels for collective operation $c$ ($l_c$),
       The number of queues that are allocated for
       collective operation $c$ at level $l$ ($nq_{cl}$), the set of
       queues that are allocated for collective operation c at
       level l ($q_{cl}$)
Output: The result of the search ($QE$)
1  if $t$==point-to-point then
2      $QE$=Search $Q_{PNP}$ for $SE$;
3      Return $QE$;
4  else
5      Search $pq$;
6      for $i = 1$ to $l_c$
7      do
8          $q\_num$= extract_queue_number($SE, nq_{ci}$);
9          $QE$=Search $q_{ci}[q\_num]$ for $SE$;
10     end
11     Return QE;
12 end
```

At the time of searching, we first check the type of the communication in Line 1. If it is a point-to-point communication, the partner/non-partner queue data structure for point-to-point elements (Q_{PNP}) is searched (Line 2). If the message is originated from a collective operation, we search the profiling queue pq since it might have some elements (Line 5). Then we search the queues allocated for this collective operation from the first level to the last level in order (Lines 6 to 10). Each level consists of a specific number of queues (nq_{ci}), and the queue elements are enqueued using a hashing approach. For searching each level, first we call the function *extract_queue_ number (SE,* nq_{ci}*)* which takes the search element and nq_{ci} as input and returns the queue number for the search element. For this, it simply divides the rank number of the searching element by the number of queues nq_{ci} and returns the remainder as the output.

5. Experimental results and analysis

In this section, we first describe the experimental setup. We then evaluate the impact of the proposed COL+PNP approach on the performance of some blocking and non-blocking collective operations including MPI_ Gather, MPI _Allreduce and MPI_Iallgather in Section 5.2. In Section 5.3, we present and analyze the queue search time speedup of the proposed COL+PNP message matching approach on four real applications and compare them against different approaches described as below:

COL+LL: In this approach, the COL approach is used for collective elements and a single linked list queue is used for point-to-point elements.

LL+PNP: In this approach, a single linked list queue is used for collective elements and the PNP approach is used for point-to-point elements.

OMPI: This refers to OpenMPI message queue data structure in which each rank has its own queue.

The COL+LL and LL+PNP experiments provide us the opportunity to discuss the effect of individual performance gain from COL+LL and LL+PNP on general performance gained in COL+PNP.

Section 5.4 presents the number of dedicated queues in collective, point-to-point and OMPI approaches. Finally, Section 5.5 and 5.6 discuss the application runtime speedup and the overhead of the proposed COL+PNP approach, respectively.

5.1. Experimental platform

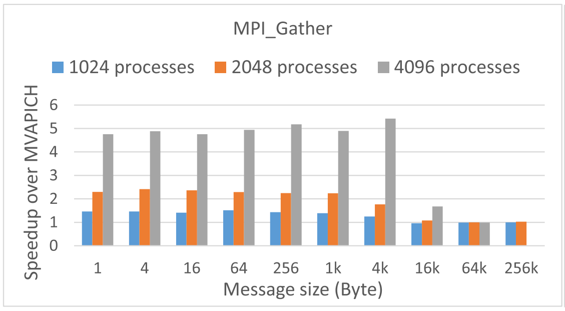
The evaluation was conducted on two clusters. The first cluster is the General Purpose Cluster (GPC) at the SciNet HPC Consortium of Compute Canada. GPC consists of 3780 nodes, for a total of 30240 cores. Each node has two quad-core Intel Xeon E5540 operating at 2.53 GHz, and a 16GB memory. We have used the QDR InfiniBand network of the GPC cluster. We refer to this cluster as cluster A in this paper. The second cluster is Graham cluster from Compute Canada. Graham has 924 nodes, each having 32 Intel E5-2683 V4 CPU cores, running at 2.1 GHz. We use 1G memory per core in our experiments. The network interconnect is EDR InfiniBand. We refer to Graham cluster as cluster B in the paper. The MPI implementation is MVAPICH2-2.2. While our design is implemented in MVAPICH, it can be applied to other MPI implementations such as MPICH and Open MPI.

The applications that we use for the experiments are Radix [9], Nbody [10,11], MiniAMR [12] and FDS [13]. The Radix application is an efficient and practical algorithm for sorting numerical keys which is used in different areas such as computer graphics, database systems, and sparse matrix multiplication. Nbody is a simulation of a dynamical system of particles, usually under the influence of physical forces, such as gravity. MiniAMR is a 3D stencil calculation with adaptive mesh refinement. We use MiniAMR's default mesh refinement options for the experiments. FDS or Fire Dynamic Simulator is a large-eddy simulation code for low-speed flows, with an emphasis on smoke and heat transport from fires. All the application results are averaged across the entire application runtime. Note that we have used these applications since they have different message queue behavior in terms of the number of point-to-point and collective elements in the queue. For example, in Radix, almost all the elements in the queue are from collectives (Fig. 1(a)). In Nbody, most of the elements in the queue are from point-to-point communications (Fig. 1(b)). Moreover, these applications span short list traversals (Radix and MiniAMR) to long list traversals (FDS). This provides us the opportunity to evaluate our design on applications with different message queue behavior.

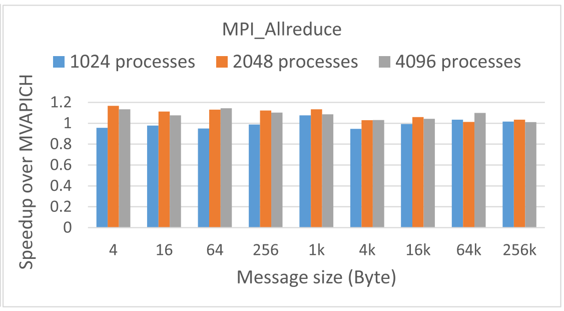
As discussed earlier, we bound the total memory overhead of the COL+PNP approach to $k \times \sqrt{n}$. We define $k = k_C + k_P$, where k_C represents the number of allocated queues in the COL approach, and k_P refers to the number of allocated queues in the PNP approach. This would allow us to evaluate the impact of memory consumption on the performance of both COL and PNP approaches, individually. Note that in the COL+LL approach, there is no memory overhead for point-to-point queue elements and $k = k_C$. Moreover, in the LL+PNP approach, there is no memory overhead for collective queue elements and $k = k_P$.

5.2. Microbenchmark results

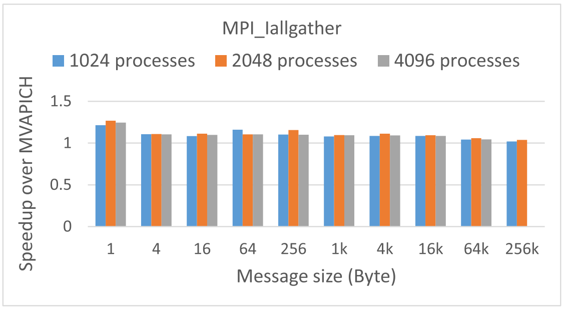
This section evaluates the impact of the proposed COL +PNP approach on the performance of some collective operations. Fig. 5(a) shows that for MPI_Gather we can gain up to 1.5x, 2.4x and 5.4x latency reduction for 1024, 2048 and 4096 processes, respectively. In this collective operation, process 0 gathers data from all the other processes which result in long message queues for this process. Therefore, the proposed message matching mechanism generates as many queues as it can to reduce the queue



(a) Latency improvement in MPI_Gather



(b) Latency improvement in MPI_Allreduce



(c) Latency improvement in MPI_Iallgather

Fig. 5. Latency improvement in MPI_Gather, MPI_Allreduce and MPI_Iallgather, for $k = 1$ in Cluster A.

search time for process 0. For example, the number of PRQs that are generated for 1024, 2048 and 4096 processes is 32 ($\sqrt{1024}$), 45 ($\sqrt{2048}$) and 64 ($\sqrt{4096}$), respectively. In other words, process 0 reaches the memory consumption cap for the number of queues for these message sizes. Other processes generate only a few queues (around 1 or 2) as their queue length is small. Fig. 5(b) and (c) show that we can gain up to 1.16*x* and 1.26x latency reduction for MPI_Allreduce and MPI_Iallgather, respectively. The queues in these collective operations are not as long as MPI_Gather. Therefore, around 10 to 20 queues will be enough for them to gain this speedup.

One observation from Fig. 5 is that the performance improvement decreases with increasing message size. The reason for this is that as we increase the message size, the network's data transfer speed becomes the bottleneck rather than message matching performance.

5.3. Application queue search time

In this section, first we discuss the queue search time performance for collective elements in the COL+PNP approach with different memory consumption cap parameter *k*. This allows us to evaluate the impact of different *k* values on the performance. Then, we discuss the impact of four different approaches on the performance of point-to-point elements, collective elements and all the elements in the queue. These approaches include COL+PNP, COL+LL, LL+PNP and OMPI. All the results are compared to the linked list data structure for Radix, Nbody, MiniAMR and FDS.

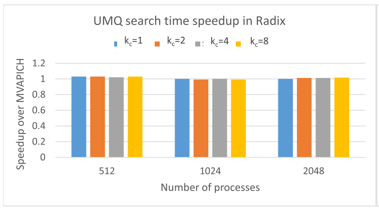
5.3.1. COL+PNP impact with different k on collective queue search time

Fig. 6 presents the queue search time speedup of the COL +PNP approach on the queue search time of the collective elements. It also evaluates the impact of different memory cap parameter k_C on the queue search time speedup. The threshold for the partner/non-partner design is $\theta = 100$ since it provides the maximum performance and scalable memory consumption as discussed in [4]. Moreover, we choose $k_P = 8$ so as not to exceed k of 16 which is the maximum memory cap for 512 processes. The results in Fig. 6 are conducted on Cluster B.

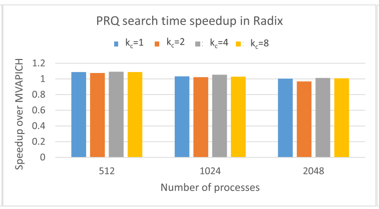
Fig. 6(a) and (b) show the average UMQ and PRQ search time speedup across all processes for Radix, respectively. As can be seen in these figures, increasing k_C does not impact the queue search time significantly. That is because of the short list traversals of the queues (around 10 elements) for this application that make a small queue memory footprint sufficient to get a minor speedup. We observe almost the same behavior for Nbody in Fig. 6 (c) and (d). As can be seen in these figures, we can gain up to 1.09x and 1.33x speedup for UMQ and PRQ in Nbody, respectively.

Fig. 6(e) and (f) show the average queue search time speedup for collective communications across all processes in MiniAMR. In this application, we can gain up to 1.37x and 1.46x search time speedup for UMQ and PRQ, respectively. As can be seen in this figure, by increasing k_C from 1 to 2 (or doubling the memory consumption), we can improve the queue search time speedup. However, increasing k_C further does not have considerable impact on queue search time. For example, with 512 processes, increasing k_C from 1 to 2 improves the search time speedup around 22% for UMQ. However, increasing it further does not improve the search time speedup considerably. This shows that for 512 processes, less than 44 ($2 \times \sqrt{512}$) queues is enough to gain the maximum search time speedup. We will discuss the number of generated queues for different number of processes in each application with more details in Section 5.4.

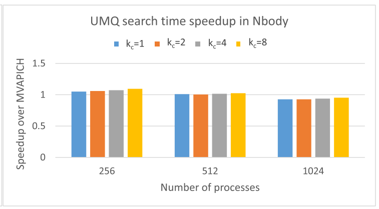
Fig. 6(g) and (h) present the UMQ and PRQ search time speedup for collective communications in FDS. For this application, we show the search time speedup for process 0 since this process has the majority of communications. As can be seen in the figures, we can gain around 42x queue search time speedup for collective queue elements in this application. Note that in FDS, each process sends a number of messages to process 0 through MPI_Gather(v). This hotspot behavior places significant stress on the MPI matching engine. Therefore, FDS results show the potential maximum performance that can be gained by the proposed message matching



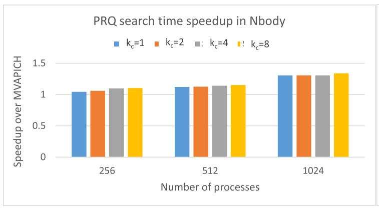
(a) UMQ in Radix



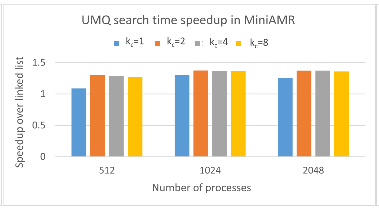
(b) PRQ in Radix



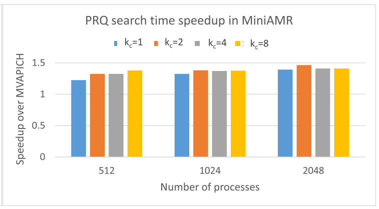
(c) UMQ in Nbody



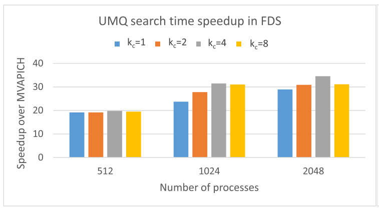
(d) PRQ in Nbody



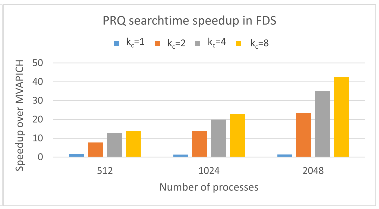
(e) UMQ in MiniAMR



(f) PRQ in MiniAMR



(g) UMQ in FDS



(h) PRQ in FDS

Fig. 6. Average UMQ and PRQ search time speedup for collective elements with the COL+PNP approach and different k_C values ($k_P = 8$) in Radix, Nbody, MiniAMR and FDS in Cluster B

mechanism. Moreover, they provide the opportunity to indirectly compare the performance gain of our approach with other message matching proposals that use this application [4,7,18]. Finally, these results show that with an MPI implementation that support long message queue traversals, we can provide the opportunity to the programmer to design less complicated code while maintaining high performance.

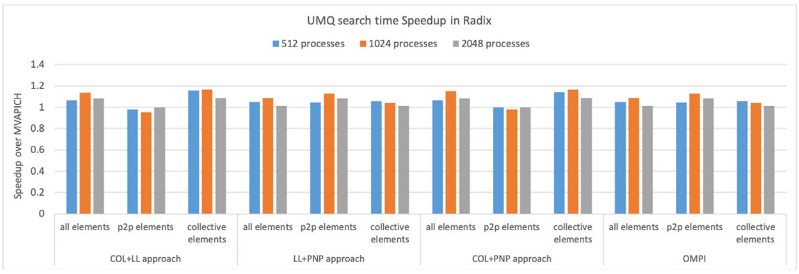
5.3.2. COL+PNP, COL+LL, LL+PNP and OMPI impact on collective, point-to-point and total queue search time

Figs. 7 and 8 show the queue search time speedup of COL+PNP, COL+LL, LL+PNP and OMPI for Radix, Nbody, MiniAMR and FDS. We present the speedup for all the elements in the queue, the point-to-point elements and also the collective elements. The experiment are conducted on Cluster B.

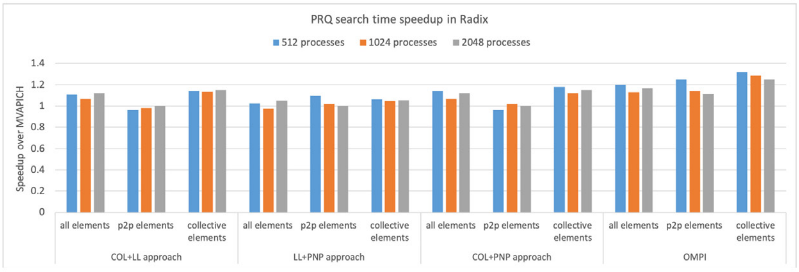
This figure shows the speedup when $k_C = 8$. The reason we choose $k_C = 8$ is that it provides a scalable memory consumption and at the same time it has the maximum performance gain in almost all cases as shown in Fig. 6. For the point-to-point elements, we use the same θ and k_P discussed in previous section.

Fig. 7(a) and (b) show the average UMQ and PRQ search time speedup for Radix, respectively. As can be seen in these figures, in all approaches, the speedup for point-to-point elements is around 1. The reason for this is that, in Radix, almost all the elements in the queue are coming from collective communications and there are a few point-to-point elements in the queue (Fig. 1(a)). These figures also show that in total we can gain up to 1.15x and 1.14x search time speedup for all the elements in the queue in COL+PNP. One observation from Fig. 7(a) and (b) is that the performance of COL+LL and COL+PNP is almost similar. In other words, the use of LL or PNP approaches for point-to-point elements does not affect the performance since there are just a few, if any, point-to-point elements in the queue. Another observation from the figure is that the total performance gained for all the elements in COL+LL and COL+PNP is more that that of LL+PNP. This is again because of a few number of point-to-point elements in this application that present no room for search time optimization of these elements. Comparing OMPI queue search time speedup with that of COL+PNP, we can observe that the performance of COL+PNP is close to OMPI without its memory overheads. The memory overhead is discussed in detail in Section 5.4.

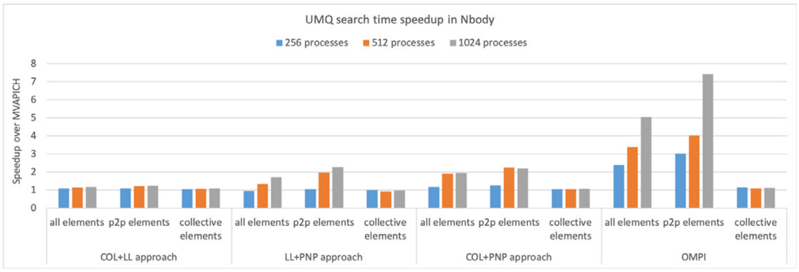
Fig. 7(c) and (d) show the average UMQ and PRQ search time speedup for Nbody. In this application, there are a significant number of elements in the queue from point-to-point communication (Fig. 1(b)). When we separate the queue for collective elements in the COL+LL approach, the queue length for point-to-point elements is reduced and its queue search time improves by up to 1.23x and 1.37x in UMQ and PRQ, respectively. The total search time speedup for all elements in UMQ and PRQ is 1.18x and 1.27x, respectively. Using the partner/non-partner message queue design for point-to-point communication in the LL+PNP approach improves the performance of point-to-point elements by 2.26x and 2.28x for UMQ and PRQ, respectively. Comparing COL+LL with LL+PNP, we can observe that LL+PNP provides more speedup for all the elements. The reason for this is that LL+PNP is improving the queue search time of point-to-point elements which are significant in Nbody compared to collective elements. In other words, LL+ PNP has more opportunity to improve performance compared to COL+LL. In the COL+PNP approach, we combine the two approaches and reach the speedup of 1.94x and 1.89x for all elements in UMQ and PRQ, respectively. As expected, this speedup is greater that the speedup of individual COL+LL and LL+PNP approaches. Comparing point-to-point speedup in COL +PNP with COL+LL and LL+PNP, we can observe that some part of the point-to-point speedup is because of separating point-to-point elements from collective elements. However, the majority of performance gain comes from the partner/non-partner message queue design. One interesting observation in Fig. 7(c) and (d) is that the perfor-



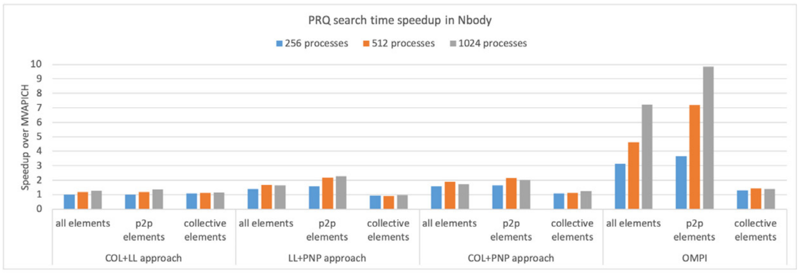
(a) UMQ in Radix



(b) PRQ in Radix



(c) UMQ in Nbody



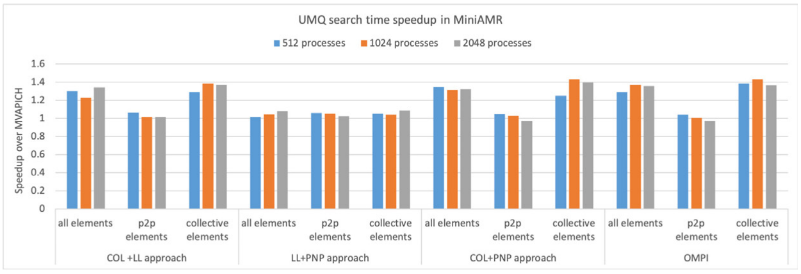
(d) PRQ in Nbody

Fig. 7. Average UMQ and PRQ search time speedup for collective, point-to-point and total elements with COL+PNP and COL+LL approaches and $k_C = k_P = 8$ for Radix and Nbody in Cluster B.

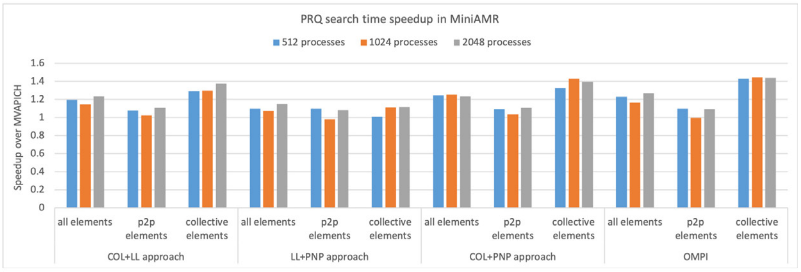
mance of collective elements in OMPI is close to that of COL+PNP (around 1.1x and 1.22x for UMQ and PRQ, respectively). This shows that $(k_c = 8) \times \sqrt{n}$ queues is sufficient to gain maximum speedup for collective elements in this application. This observation complies with the results in Fig. 6(b) and (c). For point-to-point elements, the speedup of OMPI is greater than COL+PNP. However, this improvement comes with the expense of unscalable memory consumption which is discussed in detail in the next section.

Fig. 8(a) and (b) show the queue search time speedup in MiniAMR. In this application, there are just a few number of point-to-point elements in the queue (Fig. 1(c)). Therefore, the speedup for point-to-point elements is around 1 in all cases. Consequently, the speedup of the COL+LL approach and COL+ PNP approach is almost the same for collective elements as well as all the elements. Moreover, the speedup of all elements in LL+PNP is around 1 since it uses a linked list queue for collective elements and there is not much point-to-point elements to take advantage of the PNP approach. Another observation from this figure is that the performance gain of COL+PNP is close to that of OMPI. This complies with the results in Fig. 6(g) and (h) that shows $(k_c = 8) \times \sqrt{n}$ should be enough to gain maximum queue search time speedup.

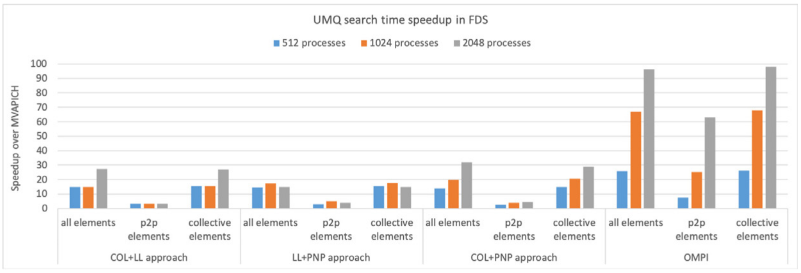
Finally, the COL+PNP results in Fig. 8(c) and (d) show that we can gain up to 4.4x and 73x search time speedup for UMQ and PRQ point-to-point elements in FDS, respectively. Moreover, the COL+LL results show that by taking out the collective elements from the queue in FDS, the search time of point-to-point elements in UMQ and PRQ improves by up to 3.34x and 71x, respectively. The total UMQ and PRQ search time speedup in the COL+LL approach is 27x and 52x, respectively. As discussed in Section 5.3.1, this large performance gain is possible due to the long list traversals in this application. Comparing the COL+LL results with that of LL+PNP, we can observe that the total performance gained for all elements in COL+LL in more than LL+PNP. This is because most of the queue elements in FDS are coming from collective communication (Fig. 1(d)) which provides more opportunity for improve performance in COL+LL. One interesting observation in Fig. 8(c)



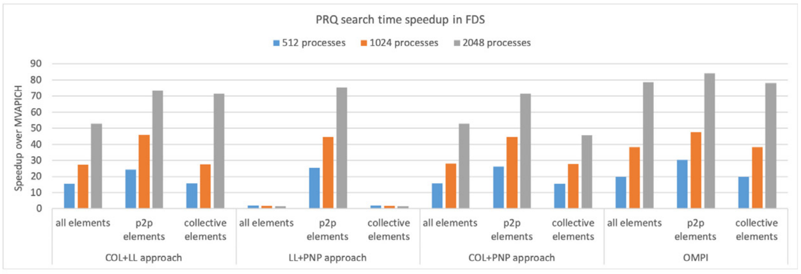
(a) UMQ in MiniAMR



(b) PRQ in MiniAMR



(c) UMQ in FDS

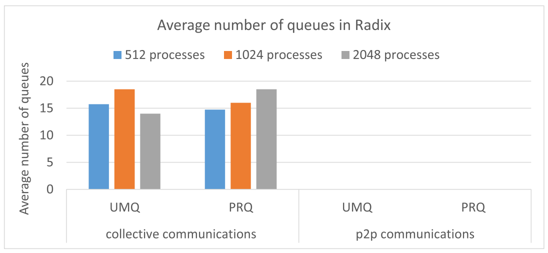


(d) PRQ in FDS

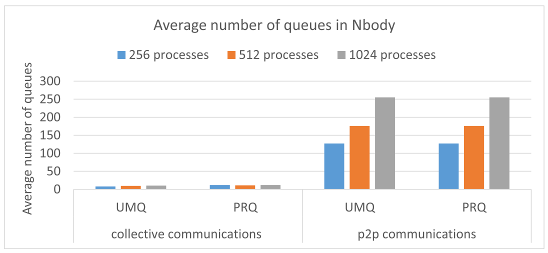
Fig. 8. Average UMQ and PRQ search time speedup for collective, point-to-point and total elements with COL+PNP and COL+LL approaches and $k_C = k_P = 8$ for MiniAMR and FDS in Cluster B

and (d) is that LL+PNP can improve the performance of collective elements up to 17x and 2x for UMQ and PRQ by just separating point-to-point elements from collective elements. COL+LL further improves the performance of collective elements by using the COL approach for collectives. Comparing the performance of point-to-point elements in CPL+PNP with COL+LL and LL+PNP in Fig. 8(c), we can observe that around 3.2x of UMQ speedup is because of separating the point-to-point elements from collectives and the rest of the performance gain is coming from partner/non-partner message queue design. On the other hand, comparing COL+PNP with COL+LL and LL+PNP in Fig. 8(d) shows that most of the performance gain for point-to-point PRQ messages is because of separating the collective and point-to-point elements. This shows that in PRQ, a significant number of collective elements should be traversed to search for a point-to-point message. The last observation from Fig. 8(c) and (d) is that OMPI provides better performance compared to COL+PNP. This complies with the results in Fig. 6(e) and (f). As can be seen in these figures, the queue search time speedup in FDS increases as we increase the number of allocated queues (k_c) as compared to other applications that $k_c = 2$ or $k_c = 4$ is sufficient for them to gain the maximum speedup. In Section 5.4.3, we will discuss that the OMPI performance gain comes with the expanse of unscalable memory consumption while COL+PNP keeps trade off between search time speedup and memory consumption.

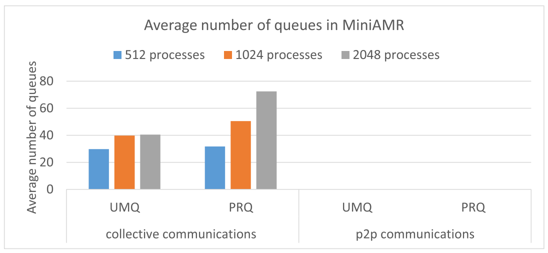
Note that the performance gain for all the elements in the queue has a direct relationship with the performance gain for point-to-point and collective elements. However, whether it is more due to point-to-point or collective performance depends on the distribution of these elements in each application. In Radix, almost all the elements in the queue are from collective communication (Fig. 1(a)). Therefore, the queue search time speedup for all elements is roughly similar to the search time speedup for collective elements (Fig. 7(a) and (b)). On the other hand, in Nbody, the number of point-to-point elements in the queue is more (Fig. 1(b)). Therefore, the search time speedup for all elements is more



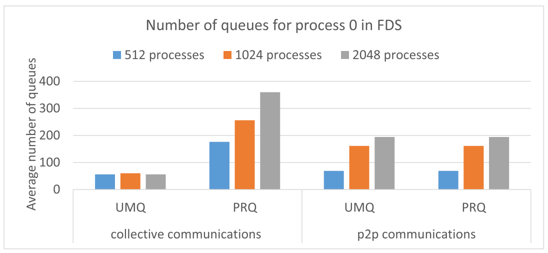
(a) Radix



(b) Nbody



(c) MiniAMR



(d) FDS

Fig. 9. Number of dedicated UMQs and PRQs for collective and point-to-point operations with the COL+PNP approach and $k = 16$ in Radix, Nbody, MiniAMR and FDS in Cluster B.

dependent on search time speedup of point-to-point elements (Fig. 7(c) and (d)). In MiniAMR and FDS, both point-to-point and collective elements contribute to the message queues. However, the contribution of collective elements is more (Fig. 1(c) and (d)). As a result, in these applications, the performance gain for all elements is due to both point-to-point and collective message matching improvement, but it is mainly because of the collective speedup (Fig. 8(a)–(d)).

5.4. Number of dedicated queues for the applications

As discussed earlier, the memory consumption of each message queue data structure is directly related to the number of allocated queues in these data structures. Therefore, in this section, we present the number of dedicated queues for the applications studied in this paper. The same parameters discussed in Section 5.3.2 are used for the experiments.

First, we discuss the number of dedicated queues for collective and point-to-point communications in the COL+PNP approach in Section 5.4.1 and 5.4.2, respectively. Then, Section 5.4.3 compares the number of allocated queues for all the elements in COL+PNP with that of OMPI.

5.4.1. Number of dedicated queues for collective communications

Fig. 9 shows the number of allocated queues in Radix, Nbody, MiniAMR and FDS for collective and point-to-point communications with the COL+ PNP approach.

Fig. 9(a) shows the average number of queues across all processes in the Radix application. This application uses the collective operations MPI_Iallgather, MPI_ Allreduce, MPI_ Reduce and MPI_ Reduce_scatter. Among these collectives, MPI_ Iallgather has the most contributions in generating long list traversals for this application, and around 65% of the queues for collective elements in Fig. 9(a) are for this operation.

Fig. 9(b) presents the average number of dedicated queues across all processes for Nbody. This application has the collective operations MPI_Allgather, MPI_Allreduce, MPI_Bcast and MPI_Reduce. Among these collectives, MPI_Allgather has the most contribution in generating long list traversals and thus, most of the dedicated queues for collective elements in Fig. 9(b) belong to this collective. Other collectives either do not generate long message queues or they are called just a few times in the application.

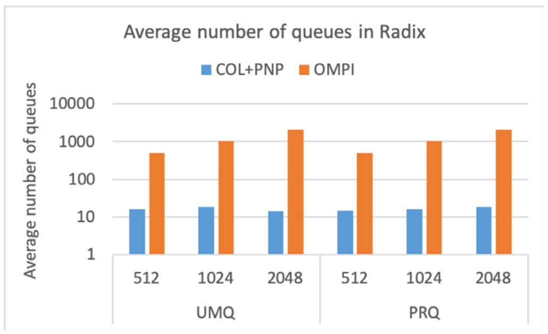
Fig. 9(c) shows the average number of generated UMQs and PRQs across all processes for MiniAMR with different number of processes. This application uses the collective operations MPI_Allreduce, MPI_Bcast, MPI_Allgather and MPI_Reduce. For this application, MPI_Allreduce has the most contribution in generating long list traversals. Therefore, most of the dedicated collective queues in Fig. 9(c) belong to this operation.

In Fig. 9(d), we present the number of UMQs and PRQs that are generated in the FDS application. Here again, we show the results for rank 0 since this process has the majority of communications (as discussed in Section 5.3.1). This figure shows that process 0 of FDS generates as many PRQs as it can for collective communications. For example, when the number of processes is 1024, 256 collective queues are generated which is the memory cap for the number of queues ($8 \times \sqrt{1024} = 256$). Note that FDS uses the collective operations MPI_Gather, MPI_ Gatherv, MPI_Allgatherv, MPI_ Allreduce and the majority of the queues that are generated for process 0 belong to MPI_Gatherv.

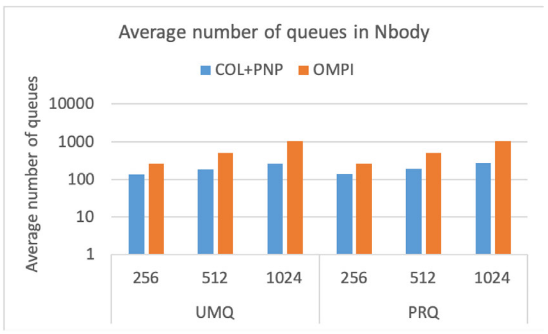
Comparing Fig. 9 with Fig. 1 shows that the number of allocated queues for collective communications is in concert with the number of collective elements in the queues for each application. For example, Nbody does not have significant number of collective elements in both UMQ and PRQ (Fig. 1(b)), so the number of collective UMQ and PRQ allocated for this application is around 10 (Fig. 9(b)). On the other hand, there are significant number of collective elements in FDS (Fig. 1(d)) and the number of collective queues allocated for this application reaches 360 (Fig. 9(d)). In general, Fig. 9 shows that the applications with longer collective traversal such as MiniAMR and FDS have a larger number of collective queues compared to Radix and Nbody with short collective traversal.

5.4.2. Number of dedicated queues for point-to-point communications

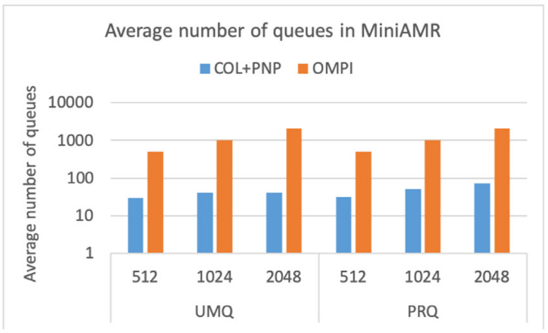
Fig. 9 shows that the number of generated queues for point-to-point communications in the COL+PNP approach directly relates to the number of point-to-point queue elements. For example, there are a few number of point-to-point elements in Radix and MiniAMR (Fig. 1). Therefore, the queue length of the original linked list



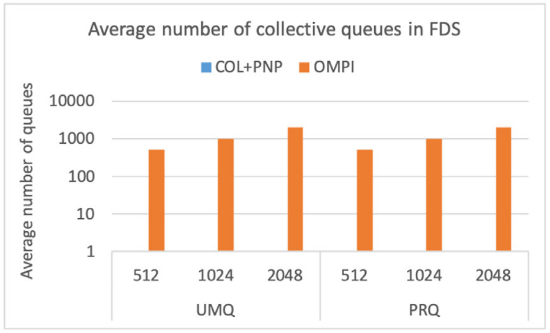
(a) Radix



(b) Nbody



(c) MiniAMR



(d) FDS

Fig. 10. Number of allocated UMQs and PRQs in the COL+PNP approach vs. OMPI for Radix, Nbody, MiniAMR and FDS on Cluster B.

queue does not reach the threshold, θ, and the number of generated queues is 0. On the other hand, the number of point-to-point elements in Nbody and FDS is more significant. This causes the queue length to reach the threshold, θ, more frequently which results in more queues.

Comparing Fig. 9 with Figs. 7 and 8, we can observe that the number of the allocated queues in the proposed COL+PNP message matching design is in concert with the queue search time speedup for both collective and point-to-point communications. One observation from Fig. 9 is that in many cases, the number of allocated queues increases with increasing number of processes. However, it never exceeds the memory consumption cap $k \times \sqrt{n}$. The number of dedicated queues is limited to a few queues for applications with short list traversals, up to the max $k \times \sqrt{n}$ for applications with long list traversals. This shows the scalability of the proposed approach in terms of memory consumption.

5.4.3. Total number of dedicated queues in COL+PNP vs. OMPI

In this section, we compare the memory consumption of the proposed COL+PNP approach with OMPI. Fig. 10 shows the average number of allocated UMQs and PRQs across all processes for Radix, Nbody, MiniAMR, and FDS. As the figure shows, the number of allocated queues in OMPI is by far more than the number of queues in COL+PNP for all four applications. This shows the unscalability of OMPI in terms of memory consumption. As discussed earlier, in OMPI each source rank has its own queue so the number of allocated queues is $O(n)$ for n processes.

As discussed in Section 5.4.2 and 5.4.1, the COL+PNP approach allocates $k \times \sqrt{n}$ queues for rank 0 in FDS. For all the other processes, one linked list queue is sufficient as they do not receive many messages. Fig. 10(d) shows the average number of queues across all processes in FDS which is around one with COL+PNP. On the other hand, OMPI allocates n queues for all the processes. We should note that Fig. 9(d) shows the number of queues allocated in COL+PNP for rank 0 while Fig. 10(d) shows the average number of queues across all processes.

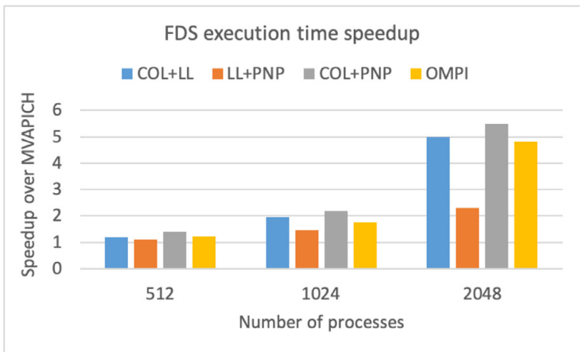


Fig. 11. FDS runtime speedup over MVAPICH in Cluster B.

5.5. Application execution time

Fig. 11 shows the FDS runtime in the proposed COL +PNP approach and compares it against COL+LL, LL+PNP, and OMPI message queue designs. The results show that we can gain up to 5x and 2.3x runtime speedup with COL+LL and LL+PNP, respectively. The reason COL+LL provides more speedup compared to LL+PNP is that the number of collective elements in FDS is more (Fig. 1(d)), so COL+LL has more opportunity to improve queue search time as discussed in Section 5.3.2 (Fig. 8(c) and (d)).

The improvement in COL+LL comes from two factors: 1) Message matching speedup for point-to-point elements since they are separated from collective elements; and 2) message matching speedup for collective elements. In Section 5.3, we discussed the impact of each of these factors on total search time improvement. The gray bar shows that we can gain up to 5.5x runtime

Table 2
Overhead/search time ratio in COL+PNP.

Applications	Number of processes	Overhead/search time ratio
Radix	512	0.0359
	1024	0.03027
	2048	0.0175
Nbody	256	0.1563
	512	0.0589
	1024	0.0343
MiniAMR	512	0.0192
	1024	0.0366
	2048	0.0138
FDS	512	0.0027
	1024	0.0021
	2048	0.0020

speedup by combining COL+LL with LL+PNP. This shows that using partner/non-partner message queue design for point-to-point elements can further improve the performance gain from 5x in COL+LL to 5.5x in COL+PNP.

The results for COL+LL, LL+PNP, and COL+PNP in Fig. 11 follows the results in Fig. 8(c) and (d) and also Fig. 9(d). As more queues are generated (Fig. 9(d)), the UMQ and PRQ search time speedup improves (Fig. 8(c) and (d)) and consequently, the FDS execution time speedup increases (Fig. 11).

One observation from Fig. 11 is that the speedup of COL+ PNP (up to 5.5x) is more than that of OMPI (up to 4.8x). On the other hand, as shown in Fig. 8(c) and (d), OMPI provides more queue search time improvement compared to COL+PNP. The reason OMPI does not provide better execution time speedup than COL+PNP, despite its great queue search time speedup, is that it allocates memory for an array of size n, which is a time consuming operation and degrades total execution time performance.

Note that we do not show the results for Radix, Nbody and MiniAMR since their queue search time speedup does not translate to considerable improvement in their application execution time (their runtime speedup is around 1).

5.6. Runtime overhead of the message queue design

The COL approach imposes some runtime overhead for calculating the required number of queues for each collective and allocating the queues. On the other hand, the PNP approach has some overhead for extracting the partners. Table 2 presents the ratio of the average runtime overhead of the COL+PNP approach across all processes over the average queue search time across all processes in Radix, Nbody, MiniAMR and FDS for different number of processes. The results show that for all applications the overhead of the proposed design is negligible compared to their queue search time.

6. Conclusion and future work

In this paper, we propose a unified message matching mechanism that considers the type of communication to improve the queue search time for collective and point-to-point elements. For this, we separate the queue elements based on their type of communication. For collective operations, we dynamically profile the impact of each collective call on message queue traversals and use this information to adapt the message queue data structure. For the point-to-point queue elements, we use the partner/non-partner message queue data structure [4]. The proposed unified approach together can improve the message matching performance while maintaining a scalable number of queues (memory consumption). Our experimental evaluation shows that by allocating 194 queues for point-to-point elements and 416 queues for collective elements, we can gain 5.5x runtime speedup for 2048 processes in applications with long list traversals. For applications with medium list traversals such as MiniAMR, it allocates the maximum of 74 queues for 2048 processes to reduce the queue search time of collective communications by 46%.

We also compare our results with OpenMPI message queue data structure and show that we can gain up to 1.44x execution time speedup over it while maintaining scalable memory consumption.

Given the recent trends towards multi-threaded MPI communication [29] and matching challenges when using multiple threads [5], we intend to extend this work to support multi-threaded communications. Note that the collective approach is more targeted for multiple threads as compared to rank-based message queue designs such as Open MPI queue data structure. That is because the rank space memory requirements in these approaches become more of an issue with multi-threaded communications.

Declaration of Competing Interest

None.

Acknowledgments

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada Grant #RGPIN/ 05389-2016, and Canada Foundation for Innovation #7154. Computations were performed on the GPC supercomputer at the SciNet HPC Consortium. SciNet is funded bythe Canada Foundation for Innovation under the auspices of Compute Canada; the Government of Ontario; Ontario Research Fund - Research Excellence; and the University of Toronto. This work was made possible by the facilities of the Shared Hierarchical Academic Research Computing Network (SHARCNET:http://www.sharcnet.ca) and Compute/Calcul Canada.

References

[1] High-performance portable aMPI, http://www.mpich.org, 2018.
[2] MPIover infiniband, omni-path, ethernet/iwarp, and roce, http://mvapich.cse.ohio-state.edu/, 2017.
[3] Open source high performance computing, https://www.open-mpi.org/, 2017.
[4] M. Ghazimirsaeed, S. Mirsadeghi, A. Afsahi, Communication-aware message matching in MPI, Concurrency and Computation: Practice and Experience (CCPE) journal. Presented in 5th Workshop on Exascale MPI (ExaMPI), 2017.
[5] W. Schonbein, M.G.F. Dosanjh, R.E. Grant, P.G. Bridges, Measuring multi-threaded message matching misery, Int. Eur. Conf. Parallel Distrib. Comput. (EuroPar) (2018) 480–491.
[6] High-performance portable MPI, http://git.mpich.org/mpich.git/shortlog/v3.2..v3.3a1, 2016.
[7] M. Flajslik, J. Dinan, K.D. Underwood, Mitigating MPI message matching misery, in: International Conference on High Performance Computing, Springer, 2016, pp. 281–299.
[8] M. Bayatpour, H. Subramoni, S. Chakraborty, D.K. Panda, Adaptive and dynamic design for MPI tag matching, in: Cluster Computing (CLUSTER), 2016 IEEE International Conference on, IEEE, 2016, pp. 1–10.
[9] G. Inozemtsev, A. Afsahi, Designing an offloaded nonblocking MPI_Allgather collective using CORE-Direct, in: Cluster Computing (CLUSTER), 2012 IEEE International Conference on, IEEE, 2012, pp. 477–485.
[10] H. Shan, J.P. Singh, L. Oliker, R. Biswas, Message passing and shared address space parallelism on an SMP cluster, Parallel Comput. 29 (2) (2003) 167–186.
[11] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, et al., The landscape of parallel computing research: a view from berkeley, Technical Report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.
[12] A. Sasidharan, M. Snir, MiniAMR-A miniapp for Adaptive Mesh Refinement, Technical Report, 2016.
[13] K. McGrattan, S. Hostikka, R. McDermott, J. Floyd, C. Weinschenk, K. Overholt, Fire dynamics simulator, users guide, NIST special publication 1019 (2013) 6th Edition.
[14] B.W. Barrett, R. Brightwell, R. Grant, S.D. Hammond, K.S. Hemmert, An evaluation of MPI message rate on hybrid-core processors, Int. J. High Perform. Comput. Appl. 28 (4) (2014) 415–424.
[15] S.M. Ghazimirsaeed, R.E. Grant, A. Afsahi, A dedicated message matching mechanism for collective communications, in: 11th International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2), ACM, 2018, p. 26.

[16] J.A. Zounmevo, A. Afsahi, A fast and resource-conscious MPI message queue mechanism for large-scale jobs, Future Gener. Comput. Syst. 30 (2014) 265–290.
[17] B. Klenk, H. Froening, H. Eberle, L. Dennison, Relaxations for high-performance message passing on massively parallel simt processors, IEEE International Parallel and Distributed Processing Symposium (IPDPS), Orlando, FL, 2017.
[18] M. Ghazimirsaeed, A. Afsahi, Accelerating MPI message matching by a data clustering strategy, High Performance Computing Symposium (HPCS), Lecture Notes in Computer Science (LNCS), Springer, in press, 2017.
[19] K.D. Underwood, K.S. Hemmert, A. Rodrigues, R. Murphy, R. Brightwell, A hardware acceleration unit for MPI queue processing, in: Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International, IEEE, 2005, pp. 10–pp.
[20] A. Rodrigues, R. Murphy, R. Brightwell, K.D. Underwood, Enhancing NIC performance for MPI using processing-in-memory, in: Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International, IEEE, 2005, pp. 8–pp.
[21] B.W. Barrett, R. Brightwell, R. Grant, S. Hemmert, K. Pedretti, K. Wheeler, K. Underwood, R. Riesen, A.B. Maccabe, T. Hudson, The portals 4.1 network programming interface, Sandia National Laboratories, April 2017, Technical Report SAND2017-3825 (2017).
[22] S. Derradji, T. Palfer-Sollier, J.-P. Panziera, A. Poudes, F.W. Atos, The BXI interconnect architecture, in: High-Performance Interconnects (HOTI), 2015 IEEE 23rd Annual Symposium on, IEEE, 2015, pp. 18–25.
[23] R. Keller, R.L. Graham, Characteristics of the unexpected message queue of MPI applications, in: European MPI Users' Group Meeting, Springer, 2010, pp. 179–188.
[24] R. Brightwell, S. Goudy, K. Underwood, A preliminary analysis of the MPI queue characterisitics of several applications, in: Parallel Processing, 2005. ICPP 2005. International Conference on, IEEE, 2005, pp. 175–183.
[25] R. Brightwell, K. Pedretti, K. Ferreira, Instrumentation and analysis of MPI queue times on the seastar high-performance network, in: Computer Communications and Networks, 2008. ICCCN'08. Proceedings of 17th International Conference on, IEEE, 2008, pp. 1–7.
[26] R. Brightwell, K.D. Underwood, An analysis of NIC resource usage for offloading MPI, in: Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, IEEE, 2004, p. 183.
[27] K.D. Underwood, R. Brightwell, The impact of MPI queue usage on message latency, in: Parallel Processing, 2004. ICPP 2004. International Conference on, IEEE, 2004, pp. 152–160.
[28] J.S. Vetter, A. Yoo, An empirical performance evaluation of scalable scientific applications, in: Supercomputing, ACM/IEEE 2002 Conference, IEEE, 2002, p. 16.
[29] D.E. Bernholdt, S. Boehm, G. Bosilca, M. Gorentla Venkata, R.E. Grant, T. Naughton, H.P. Pritchard, M. Schulz, G.R. Vallee, A survey of MPI usage in the US exascale computing project, Concurr. Comput.: Pract. Exp. (2017) e4851.