Accelerating MPI Message Matching by a Data Clustering Strategy

S. Mahdieh Ghazimirsaeed and Ahmad Afsahi

Department of Electrical and Computer Engineering
Queen's University
Kingston, ON, CANADA K7L 3N6
{s.ghazimirsaeed,ahmad.afsahi}@queensu.ca

Abstract. Message Passing Interface (MPI) is one of the most popular parallel programming models for high-performance computing. In MPI, message matching operations are in the critical path of communication, which could adversely affect the application performance. MPI libraries typically use a linked list data structure to maintain early posted receives and unexpected messages for matching operations. However, they perform poorly at scale due to long message queue traversals. In this paper, we propose an MPI message matching mechanism based on K-means clustering that considers the behavior of the applications to categorize the communicating peers into clusters, and assign a dedicated queue to each cluster. The clustering is done based on the number of queue elements each communicating process adds to the posted receive or unexpected message queue at runtime. The proposed approach provides an opportunity to parallelize the search operation for different processes based on the application‘s message queue characteristic. The experimental study with real applications confirm that the proposed message matching approach reduces the number of queue traversals, the queue search time, and the application runtime.

Keywords: MPI, Message Queues, Message Matching, HPC

1 Introduction

High-performance computing (HPC) is the cornerstone of the scientific community in tackling computationally-intensive problems in fields as diverse as cancer research and drug discovery, green energy and biofuels, weather forecasting and climate change, seismic processing for oil and gas, astrophysics, material science, genomics and bioinformatics, automotive, defense, data mining and analytics, and financial computing. Current systems have accelerated research that were not possible a few years ago. However, the demand for more computational power is never ending.

MPI (*Message Passing Interface (MPI)*, Accessed: 2017-06-20) is the de facto standard for communication in HPC systems, and by far the dominant parallel programming model used in large-scale parallel applications. MPI Processes in such applications compute on their local data while extensively communicating with each other. Communication is therefore the major bottleneck for performance. It must be optimized in MPI for performance and scalability to cope with the demands of large-scale applications.

MPI supports point-to-point communication, collective communication, and one-sided communication. In the point-to-point communication, the sender and receiver take part in the communication explicitly. In the collective communication, the messages can be exchanged among a group of processes, and communication is usually implemented on top of point-to-point communication. In both cases, messages must be matched between the sender and receiver.

To deal with unavoidable out-of-sync communication in MPI, the MPI libraries typically maintain two queues, a Posted Receive Queue (PRQ) and an Unexpected Message Queue (UMQ). When a new message arrives, the PRQ must be traversed to locate the corresponding receive queue item, if any. If no matching is found, a queue element (QE) is enqueued in the UMQ. Similarly, when a receive call is made, the UMQ must be traversed to check if the requested message has already (unexpectedly) arrived. If no matching is found, a new QE is posted in the PRQ. Message queues are in the critical path of communication in MPI, and its search time affects the performance of applications that perform extensive communications. Fire Dynamic Simulator (FDS) (McGrattan et al., 2013), LAMMPS molecular dynamics simulator (Plimpton, 1995) and AMG2006 (Yang & Henson, 2002) are examples of such applications.

Due to the message queue traversal overhead in the MPI applications that generate long message queues, using a suitable matching algorithm is of great importance. Although there are various message matching designs used in MPI implementations, such as MPICH (*MPICH: High-Performance Portable MPI*, Accessed: 2017-06-27), MVAPICH (*MVAPICH/MVAPICH2*, Accessed: 2017-06-24), and Open MPI (*Open MPI: Open Source High Performance Computing*, Accessed: 2017-06-23), or proposed in literature ((Zounmevo & Afsahi, 2014), (Flajslik, Dinan, & Underwood, 2016), (Bayatpour, Subramoni, Chakraborty, & Panda, 2016), (Klenk, Fröening, Eberle, & Dennison, 2017)), none of them takes advantage of clustering mechanisms to improve message queue operations. This paper, for the first time, proposes a mechanism that considers the impact of communication by source/receive peer processes on message queue length, and uses this information to cluster the processes into different groups. Processes are clustered based on the number of queue elements they add to the PRQ or UMQ. We will then allocate a dedicated linked list to each cluster.

The advantage of the proposed approach is twofold. First, it will reduce the average number of queue traversals due to the use of multiple queues. Secondly, assigning a dedicated message queue to the processes in the same cluster will provide the opportunity for the processes with fewer message queue elements to be searched faster. On the other hand, increasing the number of clusters would assign a dedicated message queue to the processes with large number of elements in the queue to the point that the maximum performance could be gained.

In our study with three real applications, LAMMPS, AMG2006 and FDS, on a large-scale cluster, we show that the proposed approach decreases the average queue traversals as well as the queue search time and ultimately improving the application performance. The queue search time for the AMG2006, LAMMPS and FDS applications is improved by up to 2.2x, 1.37x and 2x, respectively. Moreover, we could improve the FDS runtime by 1.33x.

The rest of the paper is organized as follows. In Section 2, we provide some background information. Section 3 presents the motivation behind this work. Section 4 describes our message queue approach. The experimental results and analysis are presented in Section 5. Section 6 discusses the related work. Finally, we conclude the paper and point to the future directions in Section 7.

2 Background

In MPI message queues, the search is done based on the tuple <context_id, rank, tag>. Context_id is an integer value representing the communicator. The rank represents the source process rank in a PRQ request, and the receive process rank in an UMQ request. Tag is also an integer value that specifies the message id. In the case of wildcard communication, the source rank and tag is specified by MPI_ANY_SOURCE or MPI_ANY_TAG, respectively.

2.1 Message Queues in MPI Implementations

MPICH (*MPICH: High-Performance Portable MPI*, Accessed: 2017-06-27), MVAPICH (*MVAPICH/MVAPICH2*, Accessed: 2017-06-24), and Open MPI (*Open MPI: Open Source High Performance Computing*, Accessed: 2017-06-23) are open source implementations of the MPI standard. MPICH is a reference implementation of MPI that is used as a base in other MPI implementations, such as MVAPICH. MPICH and MVAPICH both use the linked list data structure depicted in Figure 1. Based on this data structure, these libraries provide a straightforward implementation of the MPI message queue that searches linearly for the key tuple in $O(n_{\mathrm{q}})$ in which $\mathrm{n_q}$ is the number of elements in the queue. When $\mathrm{n_q}$ is small, the linear search is acceptable. However, at large scale, traversing a long queue is computationally intensive and requires a large number of pointer operations (e.g., access and dereferencing). The advantage of the linked list data structure is that the QEs are saved in the order of their arrival, and this conforms with the MPI point-to-point ordering semantic.

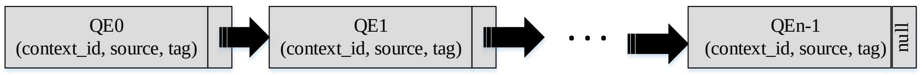

QE0
(context_id, source, tag)
QE1
(context_id, source, tag)
. . .
QEn-1
(context_id, source, tag)
null

Fig. 1: Linked list data structure in MPICH/MVAPICH

In Open MPI, the searches have been made hierarchical, by considering context_id as the first level and source as the second level. Each context_id, associated with a communicator of size n, has an array of size n representing the source ranks. Within each source rank, there is a linked list of tags. In this approach, requests bearing rank i are positioned in the ith element of the array. The advantage of Open MPI message queue data structure is that its queue search time is faster than the linked list data structure. However, contrary to the linked list, it has a large memory footprint as it requires an array of size n for each communicator of size n. Moreover, fragmenting the queues based on the communicator has limited benefit in practice since there are not many applications that use multiple communicators.

2.2 K-means Clustering

Clustering is the process of organizing objects into groups in a way that objects in the same group are similar to each other, and those from different groups are dissimilar. There are various clustering algorithms for different use cases. Among them, K-means clustering ((Seber, 2009), (*C source code implementing k-means clustering algorithm*, Accessed: 2017-06-24)) is one of the most widely used in literature. In the following, we briefly discuss how the K-means clustering works.

Consider a given data set $\{x_1, , x_{\mathrm{m}}\}$, $x_{\mathrm{i}} \in R^{\mathrm{d}}$ for $i = 1, \cdots, m$ that represents a feature d-dimensional space. Suppose we want to divide these data points in k clusters. To do so, we need to determine two sets of variables: a cluster center for each cluster, μ_{j}, $j = 1, \cdots, k$, and indicator variables (cluster membership), ρ_{lj}, which are defined as follows:

$\rho_{\mathrm{lj}} = 1$ if the data point x_{l} belongs to cluster j
$\rho_{\mathrm{lj}} = 0$ if the data point x_{l} does not belong to cluster j

where $l = 1, ..., m$ and $j = 1, ..., k$

The variables μ_{j} and ρ_{lj} can be determined by solving the following optimization problem to minimize the total distance between the data points and their cluster centers:

$$min \sum_{l=1}^{m} \sum_{j=1}^{k} \rho_{\mathrm{lj}} ||x_{\mathrm{l}} - \mu_{\mathrm{j}}||^2 \tag{1}$$

Where the optimization variables are $\rho_{\mathrm{lj}} \in \{0, 1\}$ and μj. Furthermore, $||x_{\mathrm{l}} - \mu_{\mathrm{j}}||^2 = \sum_{t=1}^{d} (x_{\mathrm{lt}} - \mu_{\mathrm{jt}})^2$ is the Euclidean distance between the data points and the clustering points. Note that it is possible to use other measures of similarity instead of the Euclidean distance.

There are various algorithms to solve this optimization problem ((Forgy, 1965), (Lloyd, 1982), (MacQueen, 1967) and (Seber, 2009)). It should also be mentioned that the number of clusters, k, should be given as an input to the algorithm for solving the above optimization problem. Since in many application domains, the value of k is not known beforehand, one needs to use statistical model selection by running the algorithm several times and choose the best possible value for k such that some error function is minimized.

3 Motivation

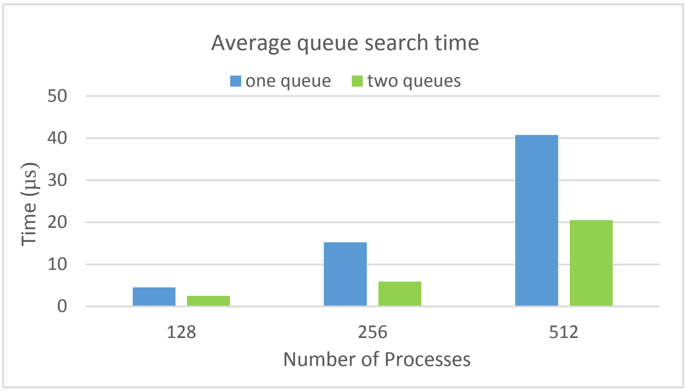
Placing all incoming unexpected messages into the same UMQ, or all early posted receives into the same PRQ, will result in long message queues and traversals when looking for a matching element. Previous research (Flajslik et al., 2016) has shown that increasing the number of message queues will potentially increase the performance. This is also used in Open MPI as an extreme case, where there is a dedicated UMQ allocated for each potential source process. The work in (Flajslik et al., 2016) naively places incoming messages and posted receives into message queues based on a hashing function, and therefore it is quite likely messages from the same source process be placed in different message queues. In this paper, we also use multiple queues to speed up the operation, however we extend our intuition a step further by considering the behavior of the applications in managing parallel message queues in MPI.

Through micro-benchmarking, we provide the evidence that increasing the number of message queues and clustering groups of processes together and assigning them a dedicated message queue based on their communication intensity will increase the message queue performance. In our *reverse search* micro-benchmark, processes P_1 to $P_{\text{n-1}}$ first send m data each to process P_0. Then, the elements are dequeued from the bottom of the queue. We measure the time P_0 spends in searching each one of the elements in the queue coming from the other processes and report the average queue search time across all incoming messages.

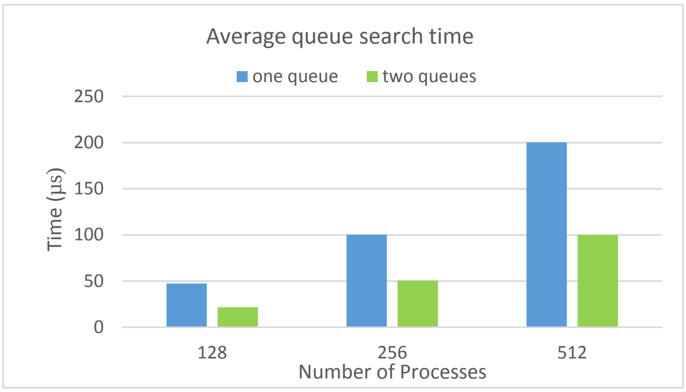
We double the number of message queues to two and compare its performance for the reverse search against the default MVAPICH2 implementation that uses a single UMQ. In the two-queue case, we place half of the incoming messages at the end of the queue in the second queue. Our objective in this case is to confirm that increasing the number of message queues indeed improves the performance.

Figure 2 shows the results for the reverse search that have been conducted on the same platform described in Section 5. As can be seen in the figure, by doubling the queue we can decrease the average queue search time. This improvement is more significant when the number of pending messages per process is larger.

In our second micro-benchmark, we revise our previous micro-benchmark in a way that the processes P_1 to P_2 are now grouped into four groups, P_{s}, P_{m}, P_{l} and P_{vl} where each process within a group sends a small (2 messages), medium (10 messages), large (50 messages), or very large (100 messages) number of messages to process P_0, respectively. We then forward a combination of these messages, such as those from the small and large groups, to one of the two queues for P_0. The remaining (e.g., medium and very large) messages will be sent to the other queue. Table 1 shows the different combination of groups of processes that are assigned to each of the two queues for P_0.



(a) M=10



(b) M=50

Fig. 2: Average UMQ search time for the reverse search

Our objective in this experiment is to understand the impact of clustering on multiple message queues and compare its performance against a single queue as well as a four-queue case where the messages from each group are sent to a distinct queue. Figure 3 compares the average UMQ search time for our micro-benchmark with different clustering of processes shown in Table 1 with a single-queue case and a 4-queue case, respectively.

Two conclusions can be derived from these results. First, as shown in the previous micro-benchmark results in Figure 2, increasing the number of queues improves the queue search time. The second and more important observation from this experiment is that the way we assign the queues to processes has a considerable impact on the queue search time. In other words, if we assign the processes to queues appropriately (clustering 3 and 4), the two-queue case can even beat the case with four queues. However, assigning the processes to queues incorrectly would negate the impact of increasing the number of message queues and may make it as worse as the case with a single queue (clustering 5).

Table 1: Different combination of groups of processes assigned to each queue

*Clustering*1	Queue 1: P_s & P_m	Queue 2: P_l & P_{vl}
*Clustering*2	Queue 1: P_s & P_l	Queue 2: P_m & P_{vl}
*Clustering*3	Queue 1: P_s & P_{vl}	Queue 2: P_m & P_l
*Clustering*4	Queue 1: P_s & P_m & P_l	Queue 2: P_{vl}
*Clustering*5	Queue 1: P_m & P_l & P_{vl}	Queue 2: P_s
*Clustering*6	Queue 1: P_s & P_m & P_{vl}	Queue 2: P_l
*Clustering*7	Queue 1: P_s & P_l & P_{vl}	Queue 2: P_m

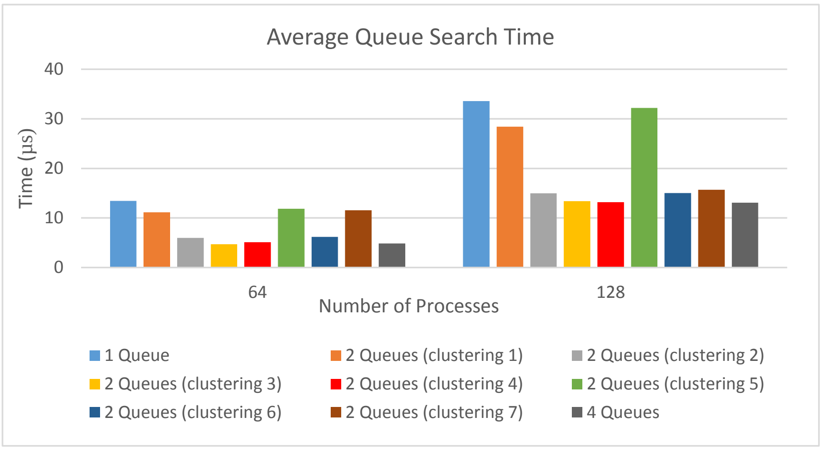


Fig. 3: Average UMQ search time for the reverse search and clustering

4 The New MPI Message Matching Design

The motivational results showed that adding a second message queue can reduce the queue search time significantly. In addition, it became possible to reduce the queue search time further by clustering the processes into different groups based on their communication frequency and allocating a dedicated message queue to each group. Motivated by these results, we propose a new message matching design that classifies the processes into different clusters based on their communication characteristics using the K-means clustering algorithm. In this approach, classification is done based on the number of elements the processes add to the message queues at runtime. Then, we allocate a dedicated message queue to each cluster. Our methodology uses a static approach, in which the application is profiled once to gather the message queue statistics at runtime for clustering purposes. Thereafter, in the future runs, incoming messages or posted receives are enqueued in their respective queues according to the classification made during the clustering phase. The overall process can be summarized as follows:

(a) Run the MPI application once to gather information on how it affects the message queues in the MPI library implementation at runtime.
(b) Use this information to perform a cluster analysis (K-means) of the MPI application processes.
(c) Redesign the MPI library implementation to have a dedicated queue for each cluster of processes.
(d) Use the clustering information obtained in Step (b) to enqueue the incoming messages or posted receives from the same MPI application in the next run to their dedicated queues in the redesigned MPI library implementation.

Figure 4 shows the clustering stage for the unexpected message queue. As the figure shows, we first derive the *message queue feature* matrix representing the total number of messages coming to the queues for peer processes. In this matrix, each row, representing a process i, has an array of size n, where n is the number of processes. The element $A_{\text{i,j}}$ indicates the total number of messages that was added to the queue from the source process j to the destination process i. This array is then used to classify the processes into groups or clusters. Any clustering algorithm can be used here for the classification. In this paper, we use the K-means clustering algorithm, discussed in Section 2.2. The output of the clustering phase is a *clustering identifier* matrix file with elements indicating the cluster number for source-destination pairs.

Figure 5 shows how the clustering information gathered in the first phase is used to build the message queues in future runs. The clustering identifier matrix is given as an input to the message queue identifier. Each destination process i in the clustering identifier matrix has an array of size n. For the peer destination process i, the element $B_{\text{i,j}}$ indicates the cluster in which the source rank j belongs to. In this design, each cluster represents one of the queues. When a message arrives or a receive call is issued, the message queue identifier retrieves the cluster number from the clustering identifier matrix and enqueues it into the corresponding queue, if it cannot be matched with posted receives in PRQ or unexpected messages in the UMQ.

The new MPI message matching design provides two main advantages. First, leveraging multiple queues can reduce the number of traversals and consequently, the queue search time. Secondly, clustering the processes based on the queue

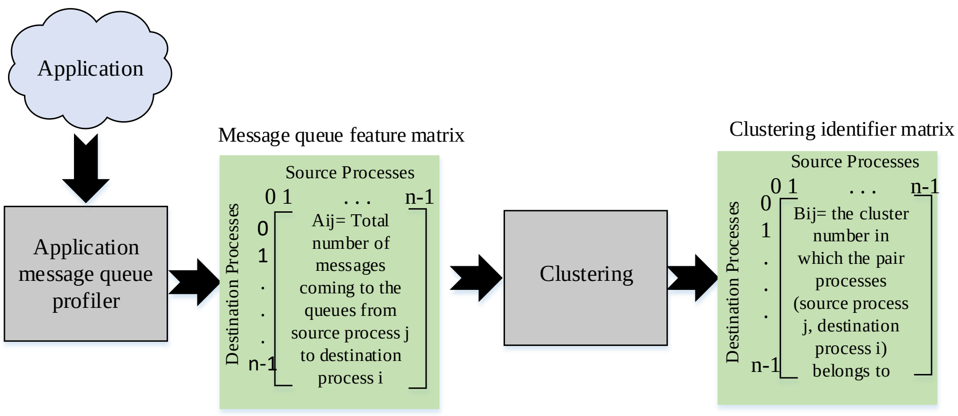

Application
Application message queue profiler
Message queue feature matrix
Source Processes
0 1 . . . n-1
Destination Processes
0 1 . . . n-1
Aij= Total number of messages coming to the queues from source process j to destination process i
Clustering
Clustering identifier matrix
Source Processes
0 1 . . . n-1
Destination Processes
0 1 . . . n-1
Bij= the cluster number in which the pair processes (source process j, destination process i) belongs to

Fig. 4: Clustering phase for the proposed unexpected message queue

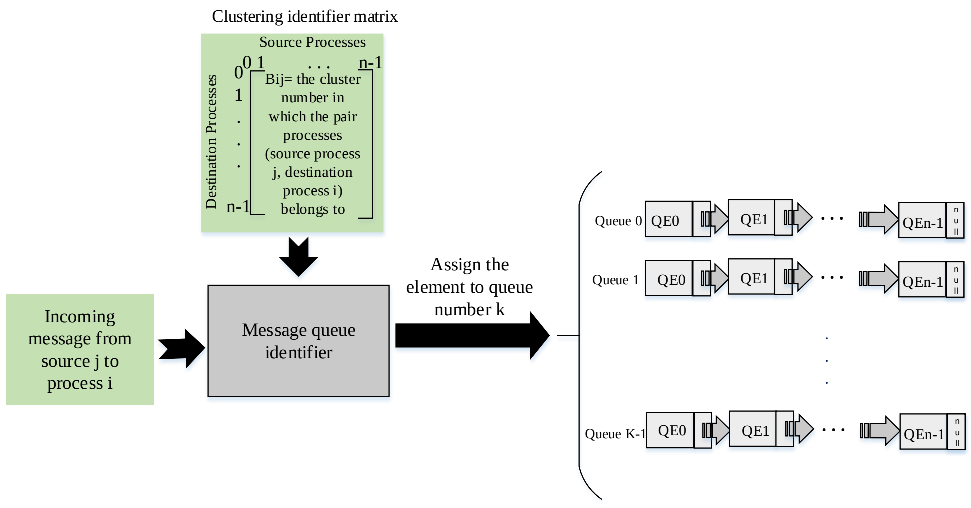

Clustering identifier matrix
Source Processes
0 1 . . . n-1
Destination Processes
0 1 . . . n-1
Bij= the cluster number in which the pair processes (source process j, destination process i) belongs to
Incoming message from source j to process i
Message queue identifier
Assign the element to queue number k
Queue 0 QE0 QE1 ... QEn-1 null
Queue 1 QE0 QE1 ... QEn-1 null
Queue K-1 QE0 QE1 ... QEn-1 null

Fig. 5: Clustering based message queue structure

behavior of the application and assigning a dedicated message queue to each cluster provides them the opportunity to be searched faster and consequently reduces the queue search time and application runtime. However, it should be mentioned that this approach is more suitable for applications with a static communication profile and those that do not create additional communicating processes at runtime. Designing a dynamic clustering approach is part of our future work.

As for the wildcard communication, the MPI_ANY_TAG wildcard is automatically supported in the proposed message queue structure, as elements from a source is stored in the same linked list 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 rank j in the posted receive queue is required, both the posted receive queue k corresponding to rank j (derived from K-means clustering) and the PRQ_ANY queue are searched. If there are multiple matches, 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 queues k $(0 \leq k < K)$ will be searched, and the element with the smallest sequence number will be selected as the matching element. Note that the user can provide a hint to the runtime library to disable the search mechanism for MPI_ANY_SOURCE if the application does not use any wildcard communication.

5 Performance Results and Analysis

This section studies the performance of the proposed clustering based message queue structure against MVAPICH default queue data structure. We present the results for the average number of queue traversals, average queue search time, and the execution time for three real MPI applications, AMG2006, version 1.0 (Yang & Henson, 2002), LAMMPS, version 14May16 (Plimpton, 1995), and FDS, version 6.1.2 (McGrattan et al., 2013). AMG2006 is a parallel algebraic multigrid solver for linear systems arising from problems on unstructured grids. It is a Single Program, Multiple Data (SPMD) code which uses MPI. Parallelism is achieved by data decomposition.

LAMMPS, Large-scale Atomic/Molecular Massively Parallel Simulator, 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. It can be used to model atoms or, more generally, as a parallel particle simulator at the atomic, meso, or continuum scale. In this paper, we have used the rhodopsin protein benchmark in our experiments. This benchmark uses the particle-mesh method to calculate long range forces (Plimpton, Pollock, & Stevens, 1997).

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.

The experimental study is done 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 MVAPICH2-2.0.

5.1 Message Queue Traversals

We measure the average number of traversals to find the desired element in UMQ for AMG2006 and LAMMPS applications. Figure 6 and Figure 7 show the average number of traversals for different sources for each process in a heat map in the proposed approach and the default MVAPICH implementation. The horizontal axis shows the source processes and the vertical axis shows the destination processes. The red points depict high number of traversals while the green and yellow points show medium and low number of traversals, respectively.

Figure 6 shows the average number of traversals for UMQ in AMG2006 in the default MVAPICH implementation (Figure 6a) and the cluster based approach with the K-means algorithm and k = 32 (Figure 6b) for 1024 processes. It is obvious from the figures that the new message queue design can reduce the number of traversals considerably which leads to reducing the queue search time of the application, consequently.

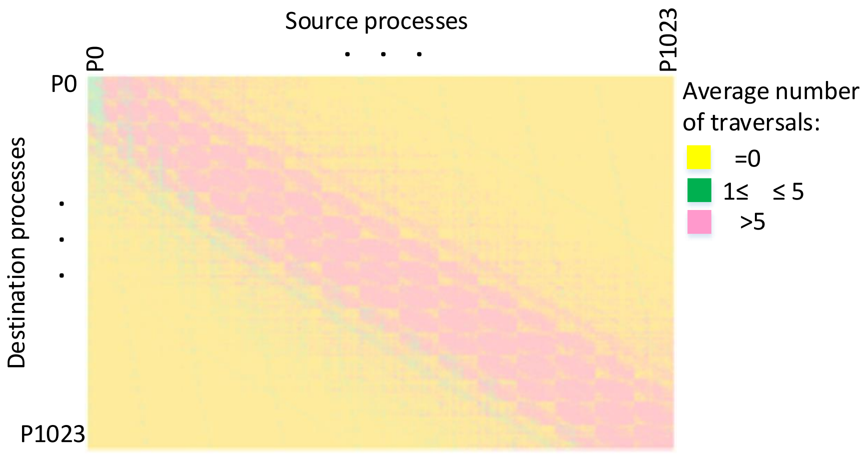
We ran the same experiment for the LAMMPS application. Figure 7 shows the average number of traversals for UMQ in MVAPICH and our approach for 240 processes, as it generates longer queues than 1024 processes. Comparing Figure 7a and 7b, it is obvious that in the cluster based approach the red areas are reduced and the yellow areas are increased, which confirms that the proposed approach reduces the number of traversals significantly.

In FDS, process 0 is the only process that has a significant number of communications, while the number of traversals in other processes is rather negligible. Therefore, we avoid presenting a figure for the number of traversals in the FDS application as it would not provide much information. However, we will show the performance gain of the FDS application by presenting its queue search time and application runtime in Section 5.2 and Section 5.3, respectively.

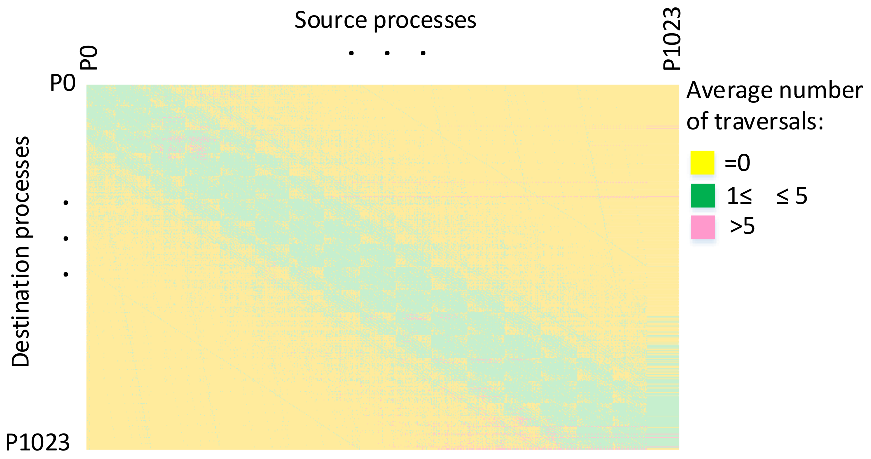
5.2 Queue Search Time

In this section, we compare the PRQ and UMQ search time in linked list data structure used in MVAPICH with the cluster based approach for AMG2006, LAMMPS and FDS applications. Figure 8 through Figure 10 show the results, respectively. In these figures, k is the number of clusters in K-means clustering and consequently the number of message queues. When k = 1, there is only one queue, representing the default MVAPICH matching algorithm.

Figure 8 shows the PRQ and UMQ search time and speedup for AMG2006. As can be seen in the figure, up to 1.5x speedup can be achieved for UMQ. The

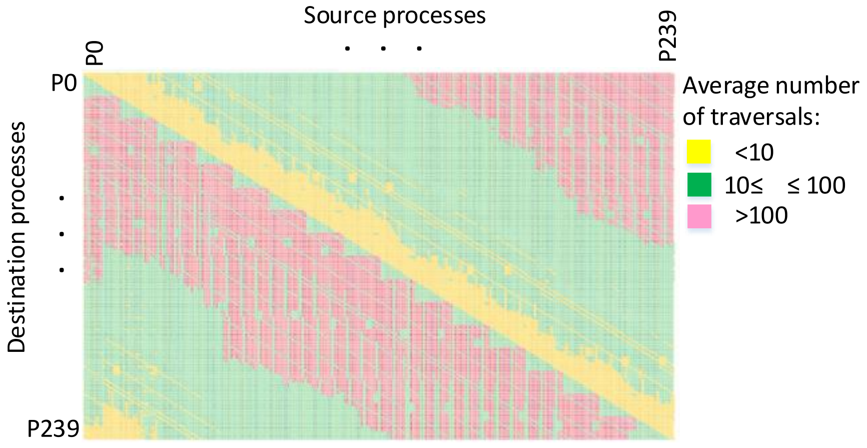


(a) default MVAICH

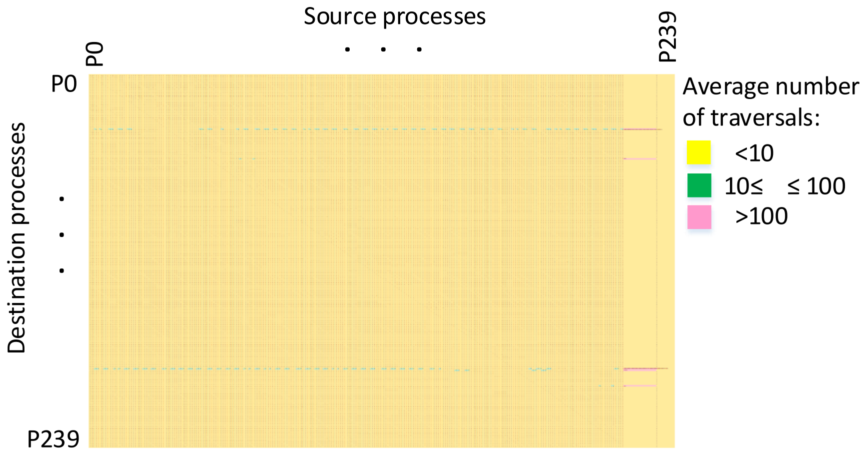


(b) cluster based (k=32)

Fig. 6: Average number of traversals for AMG2006 in MVAPICH and K-means clustering for 1024 processes



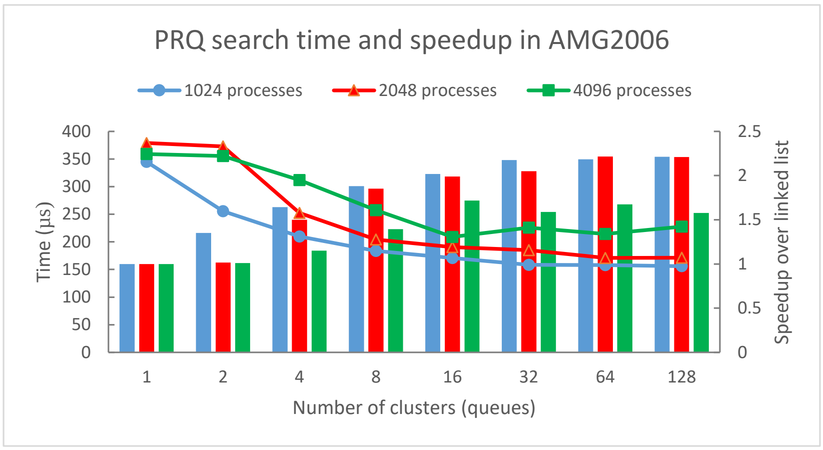
(a) default MVAICH



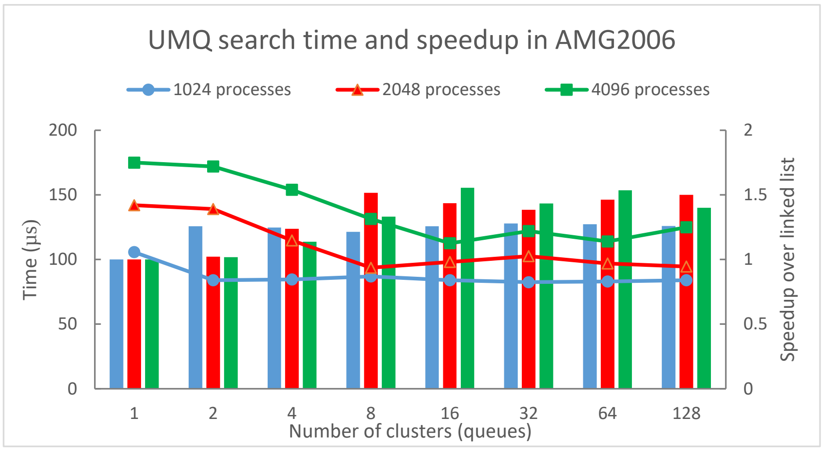
(b) cluster based (k=32)

Fig. 7: Average number of traversals for LAMMPS in MVAPICH and K-means clustering for 240 processes

average UMQ search time is decreased when the number of clusters is increased from 2 to 16. However, increasing the number of clusters further does not improve the queue search time. Similar trend is seen for PRQ with 4K processes, however cases with more than 16 clusters can still provide some performance improvement for 1K and 2K process counts. The PRQ speedup is maxed out at 2.2x for 1K and 2K processes and 1.7x for 4K processes, respectively.



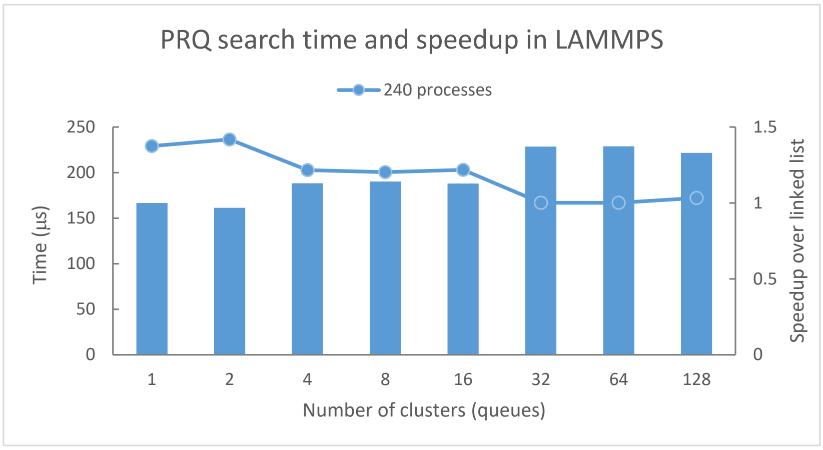
(a) PRQ



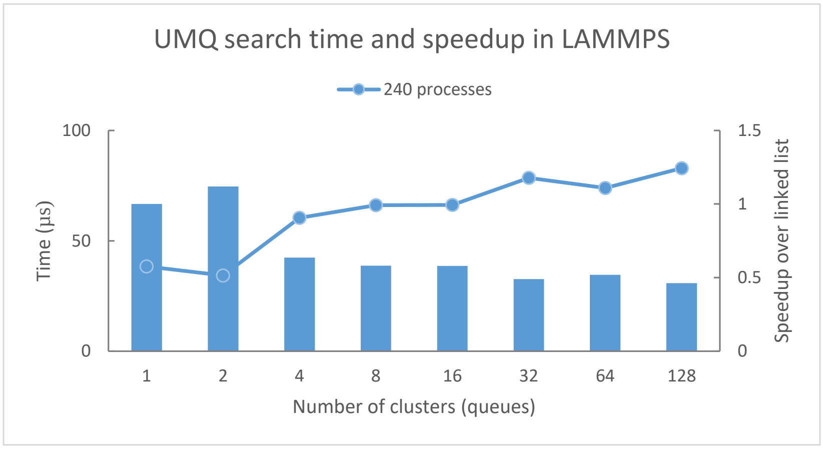
(b) UMQ

Fig. 8: Average PRQ and UMQ search time for AMG2006 in K-means clustering and MVAPICH (k = 1)

Figure 9 shows the PRQ and UMQ search time and speedup for LAMMPS. Except for k = 2, the K-means clustering approach does not improve the UMQ search time. This is because of the short list traversals of UMQ in LAMMPS that does not compensate the overhead of retrieving the queue number in the K-means approach. On the other hand, PRQ has longer list traversals, so its queue search time improvement is more significant with the K-means approach, and we can achieve up to 1.37x speedup.



(a) PRQ

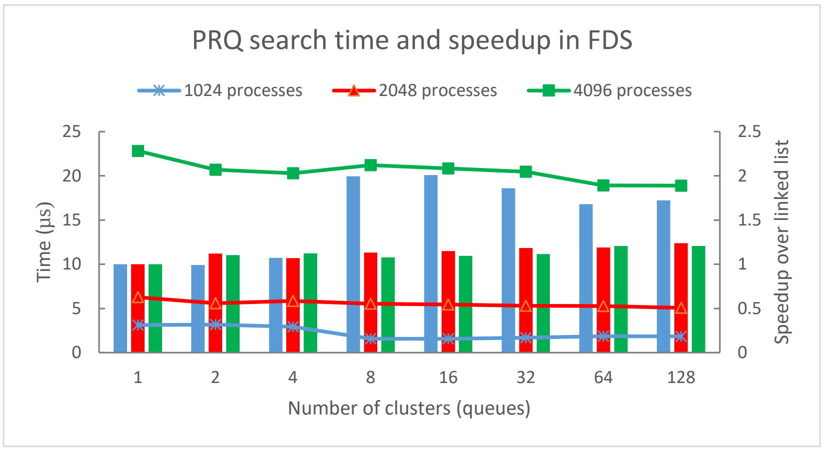


(b) UMQ

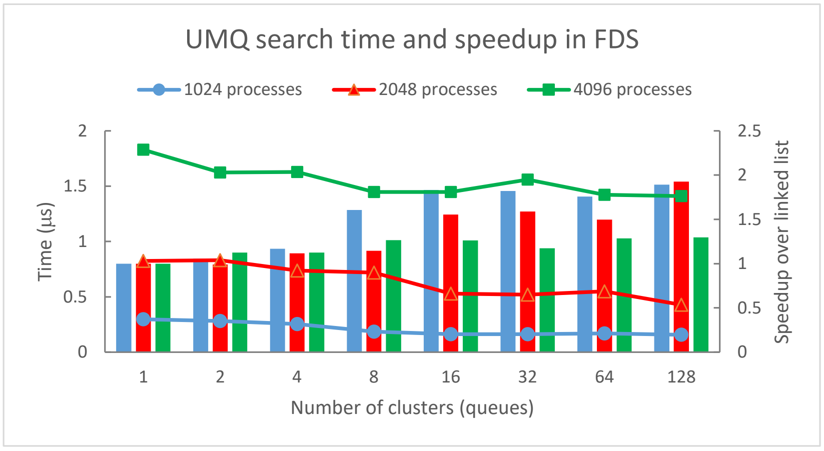
Fig. 9: Average PRQ and UMQ search time for LAMMPS in K-means clustering and MVAPICH (k = 1)

Figure 10 shows the PRQ and UMQ search time and speedup 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.

For PRQ, a speedup of around 2 can be achieved for 1K processes with 8/16 clusters, but for other process counts a steady 1.2x speedup can be reached. The trend is somewhat diffrent for UMQ, as higher speedups (up to 2x) can still be reached at larger number of clusters.



(a) PRQ



(b) UMQ

Fig. 10: Average PRQ and UMQ search time for FDS in K-means clustering and MVAPICH (k = 1)

5.3 Application Runtime

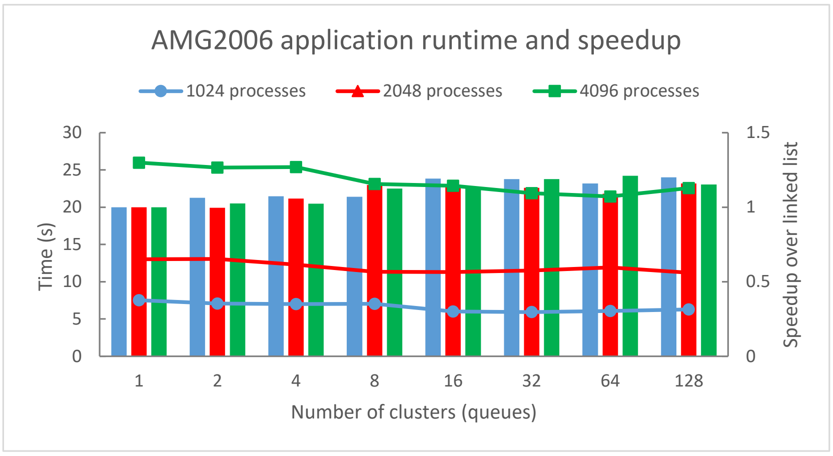
In this section, we compare the performance of the AMG2006, LAMMPS and FDS applications in our approach against the linked list data structure in MVAPICH. The results in Figure 11 is in concert with the results shown in Figures 8 through 10. AMG2006, LAMMPS, and FDS achieve 1.2x, 1x, and 1.33x speedup, respectively. This is to some extent a reflection of the performance improvements in queue search time. The reason for lesser performance improvement for AMG2006 and LAMMPS lies in the number of times their queues are searched. Figure 12 shows the maximum number of times the UMQ and PRQ are searched in FDS, AMG2006 and LAMMPS applications. One can observe the sharp contrast between AMG2006 and LAMMPS with the FDS application. Reducing the average queue search time in FDS considerably impacts its execution time. However, the improvement is less in AMG2006 and LAMMPS, mainly due to much smaller number of queue searches.

6 Related Work

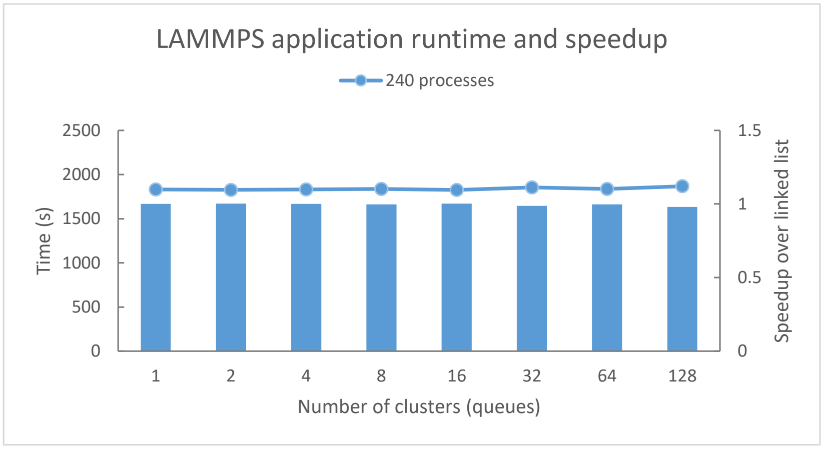
Several works have studied the impact of the message queue features on the performance of different MPI applications ((Keller & Graham, 2010), (Brightwell, Goudy, & Underwood, 2005), (Brightwell, Pedretti, & Ferreira, 2008) and (Brightwell & Underwood, 2004)). Keller and Graham (Keller & Graham, 2010) show that the characteristics of UMQ, such as the size of UMQ, the required time for searching the UMQ and the length of time such messages spend in these queues, have considerable impact on the scalability of some MPI applications (GTC, LSMS and S3D). The works in (Brightwell et al., 2005), (Brightwell et al., 2008) and (Brightwell & Underwood, 2004) focus on evaluating the latency and message queue length of some specific applications.

There have been various works on improving the message queue operations in MPI ((Zounmevo & Afsahi, 2014), (Flajslik et al., 2016), (Bayatpour et al., 2016) and (Klenk et al., 2017)). The main idea behind most of these approaches is to improve the queue search time by reducing the number of queue traversals. This can be done by taking advantage of multidimensional queues (Zounmevo & Afsahi, 2014), hash tables (Flajslik et al., 2016) or parallelizing the search operation by Graphics Processing Units (GPUs) (Klenk et al., 2017). Zounmevo and Afsahi (Zounmevo & Afsahi, 2014) proposed a 4-dimensional data structure to decompose ranks to multiple dimensions. The aim of this data structure is to skip searching a large portion of the data structure for which the search is guaranteed to yield no result.

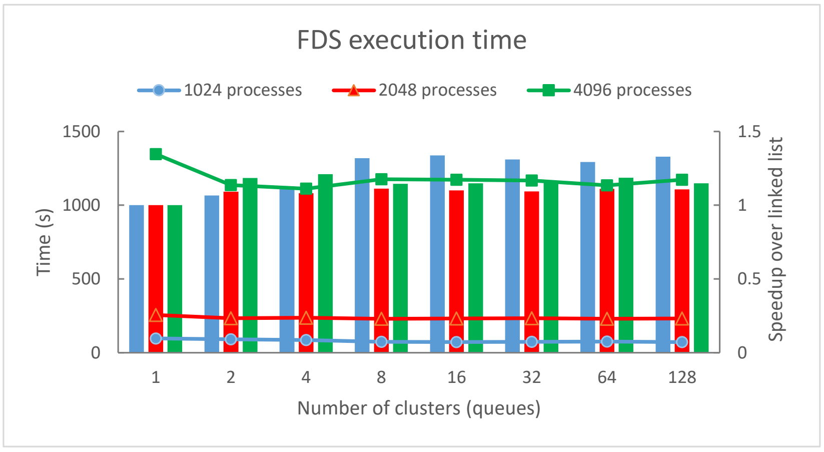
Flajslik, et al. (Flajslik et al., 2016) use a hash function to assign each matching element to a specific queue. The hash function is based on a full set of matching criteria (context_id, rank and tag). This data structure consists of multiple bins with a linked list within each bin. The number of bins and the hash function can be set at configuration time. The evaluation results show that by increasing the number of bins, the number of traversals and consequently the queue search time and application runtime would be reduced significantly.



(a) AMG2006 application



(b) LAMMPS application



(c) FDS application

Fig. 11: Application runtime in FDS, AMG2006 and LAMMPS applications

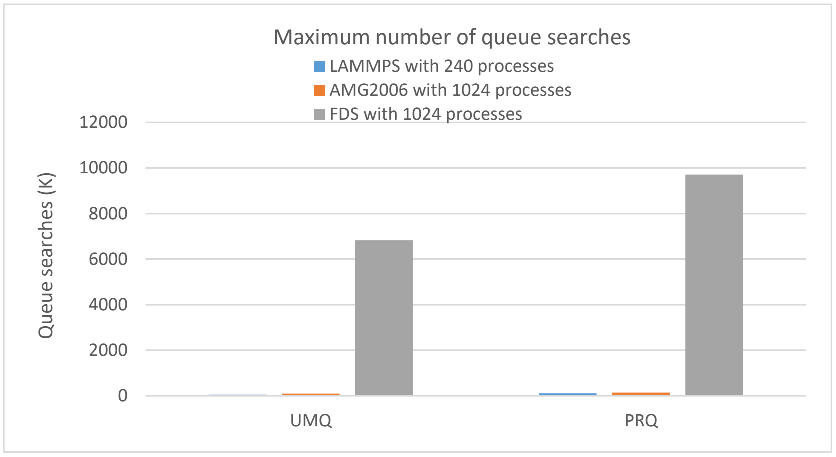


Fig. 12: Maximum number of times the queue is searched

In (Bayatpour et al., 2016), the authors propose a message matching design that dynamically selects between one of the existing approaches: a default linked list, a bin-based design (Flajslik et al., 2016) and a rank-based design in which each source process has its own dedicated message queue. This work starts running the application with the default linked list design. During the application runtime, the number of traversals for each process is measured. If the number of traversals is more than a threshold, they switch to the bin-based design. They can also switch to the rank-based design if it is specified by the user at the configuration time. Our proposed cluster based approach differs from this work in that it profiles the communication traffic between each communicating peers and takes advantage of a clustering mechanism to improve message matching performance.

Klenk, et al. (Klenk et al., 2017) propose a new message queue design to take advantage of the availability of large number of threads on GPUs. This message matching algorithm searches the queue in two phases (scan and reduce) and provides 10 to 80 times improvement in matching rate.

Other researches investigate MPI matching list improvement in hardware ((Underwood, Hemmert, Rodrigues, Murphy, & Brightwell, 2005) and (Barrett et al., 2013)). Underwood, et al. (Underwood et al., 2005) accelerate the processing of UMQ and PRQ by offloading the MPI message matching to specialized hardware. By increasing the scale of the current systems, offloading the MPI message matching is gaining attention once again to drive down the MPI messaging latency. The Portals networking API (Barrett et al., 2013) enables such offloads. Several current low-level networking APIs also provide interfaces that include support for message matching ((*Mellanox HPC-X Software Toolkit*, Accessed: 2017-06-28) and (*Open fabric interfaces (OFI)*, Accessed: 2017-06-24)) to enable the use of specialized hardware and software techniques.

Our work improves the message matching operations in MPI by reducing the number of queue search traversals. To do this, it takes advantage of multiple queues along with clustering. The difference between the proposed work in this paper and other approaches is that our approach clusters the queue elements wisely, based on the number of queue elements they add to the message queues. This way, we can reach the message matching performance by allocating a small number of queues rather that allocating a large number of queues.

7 Conclusion and Future Work

In this paper, we propose a new message matching algorithm that profiles the message queue behavior of applications to categorize the processes into some clusters. It will then assign a dedicated message queue to each cluster. The advantage of this approach is that it parallelizes the search operation by using multiple queues. Moreover, it considers the message queue behavior of the application to speed up the search operation. The evaluation results show that the proposed algorithm can reduce the number of traversals significantly. It can also improve the queue search time and application runtime by up to 2.2x and 1.33x, respectively.

The proposed clustering approach in this paper is more suitable for applications with a static communication profile and those that do not create additional communicating processes at runtime. We intend to extend our work to a dynamic clustering approach that could dynamically capture the application queue characteristics and manage the message queues accordingly. One of the disadvantages of K-means is that the optimal number of clusters (k) is not known in advance, and that there is no definitive way to determine the number of clusters. While there exist many direct and statistical testing methods to find the number of clusters, this falls outside the scope of this study. As another direction for future work, we would like to develop a heuristic that can automatically determine a reasonable number of clusters based on the message queue behaviour of the application in order to provide a better performance than the K-means clustering without its computational complexity or memory footprint.

Acknowledgment

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada Grant $\#RGPIN - 2016 - 05389$, and Canada Foundation for Innovation Grant #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; Ontario Research Fund - Research Excellence; and the University of Toronto. We also thank Mellanox Technologies for the resources to conduct the early evaluation of this research.

References

Barrett, B. W., Brightwell, R., Hemmert, S., Pedretti, K., Wheeler, K., Underwood, K., ... Hudson, T. (2013). The portals 4.0 network programming interface. *Sandia National Laboratories, Tech. Rep. SAND2013-3181*.

Bayatpour, M., Subramoni, H., Chakraborty, S., & Panda, D. K. (2016). Adaptive and dynamic design for mpi tag matching. In *2016 ieee international conference on cluster computing (cluster)* (pp. 1–10).

Brightwell, R., Goudy, S., & Underwood, K. (2005). A preliminary analysis of the mpi queue characterisitics of several applications. *International Conference on Parallel Processing, 2005. ICPP 2005.*, 175–183.

Brightwell, R., Pedretti, K., & Ferreira, K. (2008). Instrumentation and analysis of mpi queue times on the seastar high-performance network. In *Proceedings of 17th international conference on computer communications and networks, 2008. icccn'08.* (pp. 1–7).

Brightwell, R., & Underwood, K. D. (2004). An analysis of nic resource usage for offloading mpi. In *Proceedings of 18th international conference on parallel and distributed processing symposium, 2004.* (p. 183).

C source code implementing k-means clustering algorithm. (Accessed: 2017-06-24). Retrieved from `https://www.medphysics.wisc.edu/~ethan/kmeans`

Flajslik, M., Dinan, J., & Underwood, K. D. (2016). Mitigating mpi message matching misery. In *International conference on high performance computing* (pp. 281–299).

Forgy, E. W. (1965). Cluster analysis of multivariate data: Efficiency vs. interpretability of classifications. *Biometrics*, *21*, 768–769.

Keller, R., & Graham, R. L. (2010). Characteristics of the unexpected message queue of mpi applications. In *Proceedings of the 17th european mpi users' group meeting conference on recent advances in the message passing interface eurompi* (pp. 179–188).

Klenk, B., Fröening, H., Eberle, H., & Dennison, L. (2017). Relaxations for high-performance message passing on massively parallel simt processors. In *2017 ieee international on parallel and distributed processing symposium (ipdps)* (pp. 855–865).

Lloyd, S. (1982). Least squares quantization in pcm. *IEEE transactions on information theory*, *28*(2), 129–137.

MacQueen, J. (1967). Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth berkeley symposium on mathematical statistics and probability* (Vol. 1, pp. 281–297).

McGrattan, K., Hostikka, S., McDermott, R., Floyd, J., Weinschenk, C., & Overholt, K. (2013). Fire dynamics simulator, users guide. *NIST special publication*, *1019*, 6th Edition.

Mellanox HPC-X Software Toolkit. (Accessed: 2017-06-28). Retrieved from `www.mellanox.com/products/hpcx/`

Message Passing Interface (MPI). (Accessed: 2017-06-20). Retrieved from `http://www.mpi-forum.org`

Mpich: High-performance portable mpi. (Accessed: 2017-06-27). Retrieved from `http://www.mpich.org`

Mvapich/mvapich2. (Accessed: 2017-06-24). Retrieved from `http://mvapich.cse.ohio-state.edu/`

Open fabric interfaces (OFI). (Accessed: 2017-06-24). Retrieved from `https://ofiwg.github.io/libfabric/`

Open mpi: Open source high performance computing. (Accessed: 2017-06-23). Retrieved from `https://www.open-mpi.org`

Plimpton, S. (1995). Fast parallel algorithms for short-range molecular dynamics. *Journal of computational physics*, *117*(1), 1–19.

Plimpton, S., Pollock, R., & Stevens, M. (1997). Particle-mesh ewald and rrespa for parallel molecular dynamics simulations. In *Ppsc.*

Seber, G. A. (2009). *Multivariate observations* (Vol. 252). John Wiley & Sons.

Underwood, K. D., Hemmert, K. S., Rodrigues, A., Murphy, R., & Brightwell, R. (2005). A hardware acceleration unit for mpi queue processing. In *Proceedings of 19th ieee international conference on parallel and distributed processing symposium, 2005.* (p. 96.2).

Yang, U. M., & Henson, V. E. (2002). Boomeramg: a parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, *41*(1), 155–177.

Zounmevo, J. A., & Afsahi, A. (2014). A fast and resource-conscious mpi message queue mechanism for large-scale jobs. *Future Generation Computer Systems*, *30*, 265–290.