An Efficient MPI Message Queue Mechanism for Large-scale Jobs

Judicael A. Zounmevo and Ahmad Afsahi

Department of Electrical and Computer Engineering

Queen's University

Kingston, ON, Canada

{judicael.zounmevo, ahmad.afsahi}@queensu.ca

Abstract—**The Message Passing Interface (MPI) message queues have been shown to grow proportionately to the job size for many applications. With such a behaviour and knowing that message queues are used very frequently, ensuring fast queue operations at large scales is of paramount importance in the current and the upcoming exascale computing eras. Scalability, however, is two-fold. With the growing processor core density per node, and the expected smaller memory density per core at larger scales, a queue mechanism that is blind on memory requirements poses another scalability issue even if it solves the speed of operation problem. In this work we propose a multidimensional queue traversal mechanism whose operation time and memory overhead grow sub-linearly with the job size. We compare our proposal with a linked list-based approach which is not scalable in terms of speed of operation, and with an array-based method which is not scalable in terms of memory consumption. Our proposed multidimensional approach yields queue operation time speedups that translate to up to 4-fold execution time improvement over the linked list design for the applications studied in this work. It also shows a consistent lower memory footprint compared to the array-based design.**

Keywords— MPI, Message Queues, Multidimensional Searches, Scalability, Exascale

I. INTRODUCTION

The Message Passing Interface (MPI) [3] is the dominant programming paradigm in high performance computing. MPI has delivered excellent performance throughout the terascale and the ongoing petascale computing eras. As of June 2012, the top petascale system [8] has about 1,572,864 CPU cores hosted in close to 98,768 nodes. These resource levels are expected to grow even larger in the exascale computing era. Little issues which are benign or unnoticeable on small systems can become unforgiving at large scales. As a consequence, researchers are working on enhancing the MPI standard and its implementations to live up to its scalability and performance delivery expectations at the aforementioned system sizes. One of the most important aspects of MPI is the set of message queues encountered in most MPI implementations to cope with the unavoidable out-of-sync communications [10][15].

A minimum of two message queues are required, both at the receive-side, to allow MPI communication operations. They are the *unexpected message queue* (UMQ) and the *posted receive queue* (PRQ). When a new message arrives, the PRQ must be traversed to locate the corresponding receive *request,* if any. If no matching is found, a request is queued in the UMQ. Similarly, when a receive call is made, the UMQ must be traversed to check if the requested message has not already (unexpectedly) arrived. If no matching is found, a new receive request is posted in the PRQ.

It has been observed that the message queue length can grow in proportion to the job size [11][12][13][15]. The Portals 4 specification draft [1] mentions that the support of unexpected messages is necessary to avoid flow control and protocol overhead. Message queues are solicited in point-to-point, collective and even modern RDMA-based implementations of Remote Memory Access (RMA) operations [19]. Actually, the UMQ is used so frequently that it has been qualified as the most crucial data structure in MPI [15]. Message queue operations must therefore be fast, efficient and easy on memory as they are on the critical path of most MPI communications.

In MPI, the minimal search key in any message queue is the tuple <contextId, rank, tag>. ContextId designates the communicator. The rank is the source process rank for a PRQ request, and the receive process rank for an UMQ request. The tag is required whenever the same process (sender or receiver) can have more than a single pending message/request in any queue. Internal predefined tags are also used by MPI implementations even for communication models, such as collectives and RMA, which do not require a tag argument in the API.

In this work, we propose a scalable mechanism to provide fast and lean message queue management for MPI jobs at large scales. We resort to multiple decompositions of the search key. The proposal, built around a multidimensional data structure, exploits the characteristics of the contextId and rank components to considerably mitigate the effect of job sizes on the queue search times. We obtain up to 4-fold execution time speedup with our proposed approach over the linked list design; and show that it can be substantially more memory-efficient than an array-based design that we considered as an upper-bound for the speed of operation.

In what follows, Section 2 presents the related works. Section 3 discusses the motivations behind this work. Section 4 presents the proposed message queue approach as well as runtime complexity and memory overhead analyses. Section 5 presents the experimental results. Section 6 concludes the paper and points to the future work.

II. Related Work

Various flavours of message queue implementations exist outside the MPI middleware. MPI over Portals implementations are mentioned for which the UMQ and the PRQ exist in network interface card (NIC) memory [13]. The Cray MPI [14] offloads most of its receive-side matching operations on the Portals network infrastructure [1], which it is layered on. Though, according to the Portals specification draft [1], a resource exhaustion risk exists for large message queues. In the current version of Portals, this situation is handled by dropping packets; and the end result is a slowdown.

There is a mention of MPI message matching offloading on Quadrics [17] and Myrinet [23] network interfaces using on-NIC thread and memory. For InfiniBand [2], TupleQ [16] has been proposed where a Shared Receive Queue (SRQ) is created for each <contextId, rank, tag> tuple to match messages in hardware and improve the Rendezvous protocol [18]. Because each created SRQ is never freed, TupleQ is not suitable for large applications, even when there is no queue buildup. Modified SRAM-equipped NICs [25] have also been used to process message queues in associative list processing units. The tests presented in [25] are only simulations made on a custom stripped-down implementation of MPI-1.2. A NIC–associated accelerator has been proposed [22] to store message headers or small messages into a low-latency dedicated buffer. Unfortunately, the time to process long queues on embedded processors is reported to be substantially longer than host-based implementations because these processors are slower [13][24][25]. Plus, NIC or accelerator memory is usually a very limited resource which can become a scalability barrier when large queues build up [22].

Finally, hash tables have been proposed for message queue operations [6][21]. They are however reported to have prohibitive insertion times [25]. The work in [22] even mentioned that hashing can actually have a fairly negative impact on communication latency in almost all situations.

III. Motivations

As in MPICH2 [4] and MVAPICH2 [5], a straightforward implementation of the MPI message queue can use a linked list which is searched linearly for the key tuple. However, one can notice that the contextId restricts the rank space and the rank restricts the tag space for a given request. A linear queue structure which is oblivious of that organization can therefore be easily suboptimal.

The searches can actually be made hierarchical, with the contextId being the first level. In a given communicator, the ranks are always 0-based, contiguous and have a fixed known maximum value even in the case of dynamic process management [3]. With these rank properties, arrays come immediately in mind; especially because they are generally the fastest possible data structures. As shown in Fig. 1, each ContextId associated with a communicator of size *k* has an array of *k RankHead* data structures. A request bearing the rank *i* is positioned in the RankHead at position *i* in the array. Each RankHead possesses two pairs of pointers for the head and tails of its UMQ and PRQ. In each of these PRQ and UMQ, only the tag varies. Requests bearing MPI_ANY_SOURCE are hosted in a separate linked list-based sub-queue because their rank field cannot be used as array index. We will discuss the enforcement of the MPI ordering semantic for all concerned approaches in Section IV.

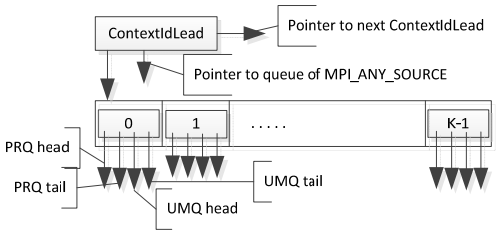

Fig. 1: Array-based message queue design for a communicator of size k

Open MPI [7] uses an approach which is somehow similar to the array-based design. In each process and for each communicator, Open MPI allocates once for all an array of data structures to host a set of information on the other processes. These arrays, while not dedicated for queue processing, are nevertheless leveraged for that purpose.

Like any *O(n)* operation, linear searches are acceptable only at small scales. Long list traversals are actually not only unscalable due to computation intensiveness but their large number of pointer operations (access and dereferencing) tends to translate into an overall larger memory access penalty at cache-level. It is reported in [10] that the linear traversal of a queue of 4095 items took up to 140 ms. This test shows how crucial message queue design is for current and upcoming systems.

The array-based approach uses substantially fewer pointer operations at runtime. Once the right contextId is found, reaching the short list of UMQ or PRQ happens in *O(1)*. Unfortunately, arrays in this case are allocated once for all for the size of the communicator. Such an allocation scheme is acceptable only at small scales. Systems growth is generally being accompanied by a certain reduction of the amount of memory per core [9]. [9] states that "a non-scalable MPI function is one whose time or memory consumption per process grows linearly or (worse) with the number of processes, all other things being equal". It is reasonable to extrapolate this statement to any other MPI mechanism. The memory requirements of the array-based approach do grow linearly with the number of processes. Furthermore, MPI programs must become topology-aware at large scales. Processes in very large jobs will therefore maintain a few additional virtual topology-associated communicators that will translate into even higher memory consumption for array-based queues. This paper presents an MPI message queue design which complies to the scalability definition in [9]. The effect of job size is mitigated on both memory and speed of operation.

IV. A New Approach to the MPI Message Queues

This section describes our new scalable message queue design for MPI. It also presents its runtime and memory consumption complexity analyses.

In the adopted hierarchical approach, each contextId, that we represent with a *ContextIdLead* object, is associated with its own sub-queue data structure. We resort to a runtime parameter which specifies a size threshold below which the sub-queue associated with a ContextIdLead is represented with a mere linked list. Above that threshold, the sub-queue associated with the ContextIdLead is represented with the proposed multidimensional mechanism described next. The details of the rule of thumb used to compute the threshold is skipped for space reasons. We mention nevertheless that the threshold values are 26, 50, 98 and 194 respectively for communicators whose sizes are in the intervals [1, 256], [257, 4096], [4097, 65536] and [65537, 1048576].

A. A Scalable Multidimensional MPI Message Sub-queue

The 4-dimensional (4-D) data-structure described in this section is only meant for large message queues. Our main goal with this approach is to perform localized searches by skipping altogether large portions of the data structure for which the search is guaranteed to yield no result. Our secondary goal is to ensure that any chunk of linear traversal required in the proposed data structure remains very short.

We split the rank in 4 slices (Fig. 2-a). Each slice represents a dimension and can hold *dimensionSpan* different positions. *dimensionSpan* is a power of two in order to allow fast bitwise decompositions. Its relation with communicator sizes is determined by the formula $dimensionSpan = 2^{\left\lceil \frac{\log_2(communicator\ size)}{4} \right\rceil}$. *dimensionSpan* is 4, 8, 16 or 32 for the intervals [1, 256], [257, 4096], [4097, 65536] and [65537, 1048576], respectively. The encoding takes respectively 2, 3, 4 and 5 bits for each of those intervals. The intervals can keep increasing. In general, a 16-fold maximum communicator size is increasingly covered by adding 1 bit to *dimensionSpan*. The aforementioned secondary goal is built upon *dimensionSpan*. For any given communicator size, *dimensionSpan* determines the typical length of chunks of linear searches inside our proposed data structure. Since *dimensionSpan* is always reasonable and grows very slowly, linear search lengths are kept under control.

For the sake of conciseness, we choose a single *dimensionSpan* (e.g., 16) to explain the rest of the design. We had two design choices. The first one would build a 4-cube over the four slices. In that case, all the requests bearing the same rank would be organized in a pair of UMQ and PRQ where only the tag varies. This first approach is beneficial if processes usually maintain several pending messages, resulting in several requests having the same rank value and different tag values. Actually, when the same rank can bear a lot of tag values, a linked list for which only the tag varies can already be long enough to justify one such list per rank. If the tag number per rank is very limited, a reasonable number of distinct ranks can share the same linked list without creating long and expensive traversals. The second approach uses that observation and puts all the requests whose ranks vary only by the least significant slice in the same linked list of UMQ or PRQ. As a reminder, each slice of the rank can only take on *dimensionSpan* distinct values; meaning that even for a communicator of size 1,048,576, a maximum of only 32 distinct ranks can be in the aforementioned linked lists.

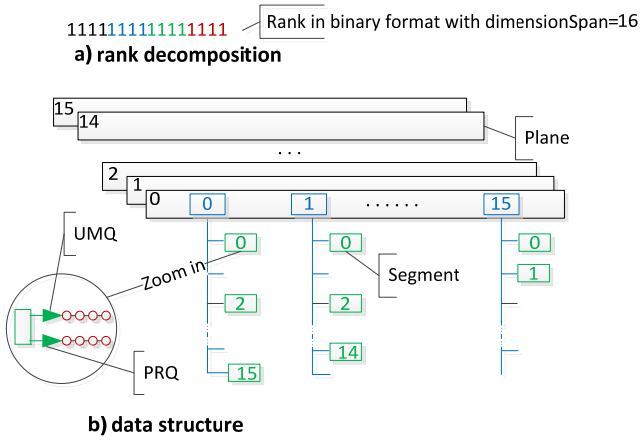

Fig. 2: 4-dimensional sub-queue with *dimensionSpan* = 16

In this second approach, which we adopt because it is more memory-efficient, the three most significant slices of the rank form a cube (Fig. 2-b). The cube is made of *Planes* whose positions are determined by the most significant slice represented in black in Fig. 2. The second and third most significant slices, respectively represented in blue and green in Fig. 2 encode respectively the X and Y coordinates in each Plane. These three most significant slices determine a *Segment* coordinate. A Segment hosts the pair of limited-length UMQ and PRQ described earlier. A Segment is a jump point which can restrict tremendously the overall number of queue items to scan for a given sought request.

Each dimension can be represented either with an array or with a linked list. Dimensions associated with bits of lesser significance (Y mostly, and X to a lesser extent) tend to impact the memory consumption if their slots are not allocated on demand. Higher dimensions are very easy on memory but they tend to be scarcely required; leading to no speed gain if they are made of arrays. We thus represent Ys and Planes with ordered linked lists and the Xs with arrays.

To understand how this proposal optimizes searches, let's consider the concrete case of a communicator of size 1,048,576. The required *dimensionSpan* is 32. A queue item bearing the rank 80,000 decomposes to 00010|01110|00100|00000; meaning that its (Plane, X, Y) coordinate is (2, 14, 4). The external search of Planes checks the coordinates along the most significant dimension until it finds the one bearing 2. Then, the internal search of that

Plane of coordinate 2 first translates into an external search of X for coordinate 14; and then for an external search of Y for coordinate 4. External searches over ordered coordinates (Plane and Y) optimize unfruitful searches. They stop right after the search goes past the sought value and conclude right away that a queue item does not exist. For instance, if coordinate 3 is reached while searching for the Plane; the whole search stops. As for the internal searches, they are performed on at most a single position of each dimension; skipping altogether a large number of irrelevant queue items. In this particular case, $31\times32^3 + 31\times32^2 + 31\times32$ items are skipped from the irrelevant 31 Planes, then the irrelevant 31 Xs of Plane 2 and finally the irrelevant 31 Ys of (Plane, X)=(2, 14).

In the overall organization of the data structure, the multidimensional sub-queue is linked to its ContextIdLead by the Plane of least coordinate. When an MPI_ANY_SOURCE request searches the UMQ, all the ranks must be probed. Then if a match is not found, because the MPI_ANY_SOURCE requests don't have a rank that can be decomposed and positioned in the 4-D structure, they are positioned in a separate linear sub-queue attached to the ContextIdLead. ContextIdLeads, Planes and Segments are small objects allocated and freed on need at the rhythm of communicator creation/destruction and queue length changes.

B. Receive Match Ordering Enforcement

The MPI standard requires receivers to match messages coming from the same sender in posted receive order. Linked list-based queues naturally offer that order. For both the array-based approach and our 4-D proposal, that order is kept as well, as long as there is no MPI_ANY_SOURCE item. As a reminder, for both the array and the 4-D designs, requests of the same ranks are always in short linked lists. For the general case, we add a monotonically increasing sequence number to the key tuple of all the requests of the same contextId. Then, each incoming message searches both the ranked PRQ structure and the MPI_ANY_SOURCE sub-queue. If both yield a match, the ordering is enforced by picking the one with smaller sequence number.

C. Runtime Complexity Analysis

In this section, we provide a comparative asymptotic analysis of the linked list, the array and our proposed 4-D message queue data structures. As stated, the linked-list design is the default approach used in MVAPICH2 [5]. We implemented the array-based method in MVAPICH2 for comparison purposes.

In all the analyses, we separate search and deletion complexities. Though, we bear in mind that a deletion is always preceded by a search in the very queue which is being deleted from. Furthermore, insertion in any MPI PRQ always happens after the UMQ has been searched in vain and vice versa. This last behaviour allows our proposed complex data structures to offer insertion in *O(1)* without breaking its positional wiring. For instance, a request of rank *r* about to be inserted in the PRQ means that the Plane and Segment corresponding to the coordinate decomposition of *r* have already been reached when searching the UMQ. Even if the Plane or Segment were not present, their insertion coordinates were at least reached while searching the UMQ.

We designate by k the number of currently active communicators in the considered process; and by n_j the number of ranks in the communicator j. We also designate by t_i the number of requests associated with the same rank i in any queue of the same communicator. In the UMQ, t_i designates the number of simultaneously pending unexpected messages coming from the process with rank i and meant for the current process. In the PRQ, t_i is the number of posted receives from the current process and meant to match messages coming from the process with rank i. In what follows, all the sum indices start at 0. We first perform the complexity analysis without MPI_ANY_SOURCE.

In the linked list-based implementation, insertion and deletion are *O(1)*. The search is $O(\sum_j^k \sum_i^{n_j-1} t_i)$ because the scanning goes through all the t_i requests of all the n_j ranks of all the k contextIds. The runtime complexity is similar for both the UMQ and the PRQ. The worst possible scenario for this approach happens when the sought request is at the tail of the queue. It also occurs when the request does not exist in the queue; the queue in this case must be traversed entirely just to realize that the search is unfruitful.

In the array-based approach, both insertion and deletion are *O(1)*. A search always requires the traversal of the ContextIdLeads in *O(k),* the access to the rank slot in the array in *O(1)* and then the traversal of the requests associated with the same rank i in $O(t_i)$. The UMQ and PRQ search complexity is thus $O(k+t_i)$.

For the 4-D structure, insertion and deletion happen in *O(1)* as well. For searches, the right Plane is reached in $O(dimensionSpan)$. The X coordinate is reached in *O(1)* because it is modeled with arrays. The Y coordinate is reached in $O(dimensionSpan)$ and the Segment is found. The Segment contains a maximum of $dimensionSpan$ distinct ranks as well; with the rank i having t_i requests. The Segment traversal therefore happens in $O(\sum_i^{dimensionSpan-1} t_i)$. Since $dimensionSpan=2^{\lceil log_2(n_j)/4 \rceil}$, the overall search complexity for both the UMQ and the PRQ is thus $O(k+2\times 2^{\lceil log_2(n_j)/4 \rceil}+1+\sum_i^{2^{\lceil log_2(n_j)/4 \rceil}-1} t_i)$, or simply $O(k+\sum_i^{2^{\lceil log_2(n_j)/4 \rceil}-1} t_i)$.

The linked list complexity does not change at all when there is MPI_ANY_SOURCE. Similarly, for any of the three designs, the insertion and deletion complexities do not change with or without MPI_ANY_SOURCE; they are always *O(1)*. For searches, however, when a posted receive bears MPI_ANY_SOURCE, once the right contextId is reached, all its UMQ ranks must be probed; and this holds

without respect to the design. Assuming that the ContextIdLead has *n* distinct ranks, the UMQ complexity thus becomes $O(k+\sum_i^{n-1} t_i)$ for both the array and the 4-D approaches; that is, $O(k)$ to search for the ContextIdLead and $O(\sum_i^{n-1} t_i)$ to go through its whole content. For the PRQ, an MPI_ANY_SOURCE sub-queue can exist. Thus, when a message arrives and tries to match a previously existing receive in the PRQ inside the relevant contextId, that MPI_ANY_SOURCE sub-queue, if it exists, must be searched on top of the regular PRQ sub-queue. If we designate by *a* the length of the MPI_ANY_SOURCE sub-queue, each of the array and 4-D designs get their PRQ complexity augmented with *a*.

In summary, all the insertion and deletion complexities are $O(1)$ for all three approaches. The search complexities are summarized in Table 1 below.

TABLE 1: RUNTIME SEARCH COMPLEXITY SUMMARY

Linked list PRQ and UMQ	$O(\sum_j^k \sum_i^{n_j-1} t_i)$
Array PRQ/UMQ without MPI_AS	$O(k+t_i)$
Array UMQ with MPI_AS	$O(k+\sum_i^{n-1} t_i)$
Array PRQ with MPI_AS	$O(k+t_i+a)$
4-D UMQ/PRQ without MPI_AS	$O(k+\sum_i^{2^{\lceil log_2(n_j)/4 \rceil}-1} t_i)$
4-D UMQ with MPI_AS	$O(k+\sum_i^{n-1} t_i)$
4-D PRQ with MPI_AS	$O(k+\sum_i^{2^{\lceil log_2(n_j)/4 \rceil}-1} t_i + a)$

A few observations stem from the aforementioned complexity analysis. When there are more than a single contextId (i.e., $k>1$), the runtime complexity of the linked list design has a cubic behaviour in terms of the used parameters. In comparison, the array-based approach and the 4-D proposal remain entirely quadratic without regard to the presence of MPI_ANY_SOURCE. Moreover, when there is no MPI_ANY_SOURCE, the parameter n_j which usually impacts the queue operation times the most is absent from the array-based approach. In the 4-D approach, this parameter has a logarithmic behaviour, meaning that its impact on the cost grows more slowly at large scales. As shown by the parameter *k*, each new communicator adds a single pointer visitation to the array and 4-D searches while it adds the quadratic term $\sum_i^{n_j-1} t_i$ to the linked list searches. Finally, none of the complexities is impacted by the use of MPI_ANY_TAG.

D. Memory Overhead Analysis

The memory overhead of the proposed 4-D approach is difficult to model concisely because it increases by steps and according to *dimensionSpan*. We therefore resort to a few concrete tests from which we make a few observations. The memory overhead is computed as how much more memory is required by the array-based method and the proposed 4-D structure compared to the linked list-based approach. Table 2 shows the comparative results for communicators of different sizes. For each size, the results are presented for a few different request numbers in the queues. It is important to mention that only the number of distinct ranks can potentially impact the memory overhead for both the array and the 4-D approach. The memory overhead of the array-based design is always fixed, equal to sizeof(ContextIdLead)+commSize*sizeof(RankHead). The 4-D structure overhead is: sizeof(ContextIdLead)+numberOfRequiredPlanes*sizeof(Plane)+numberOfRequiredSegments*sizeof(Segment). Planes are polymorphic objects whose size depends on the value of *dimensionSpan*. Segments have fixed sizes. Consequently, for higher *dimensionSpans*, the memory requirements of the 4-D structure starts high because of the Planes; and then increase relatively more slowly because more requests can be hosted in a single segment and in a single Plane.

TABLE 2: COMPARATIVE MEMORY OVERHEAD OF THE ARRAY-BASED APPROACH AND THE 4-DIMENSIONAL APPROACH

CommunicatorSize	Linked-list	Array-based	4-D
4,096	1 request	96.05KB	176B
	100 requests	96.05KB	752B
	1000 requests	96.05KB	6.06KB
	full	96.05KB	26.67KB
65,536	1 request	1.5MB	240B
	100 requests	1.5MB	528B
	1000 requests	1.5MB	3.14KB
	full	1.5MB	194.3KB
1,048,576	1 request	24MB	368B
	100 requests	24MB	512B
	1000 requests	24MB	1.81KB
	full	24MB	1.5MB

We first observe that the array-based approach can waste a lot of memory compared to the 4-D approach when there are a few requests in the queues; consider the case with only 1000 requests in a 1,048,576 rank communicator queue. The tests also show two interesting behaviours for our 4-D method. First the memory overhead increase is sub-linear in terms of the number of requests; and second, its increase rate tends to flatten considerably when the communicator size grows; consider the memory overheads increase of the 4-D approach over 1, 100 and 1000 requests for the 1,048,576 and the 65,536 communicator size queues. The two behaviours are very consistent with the scalable comportment we expect to provide. The case when all the requests of the communicator are represented, the test labelled "full", is meant to show the maximum memory overhead for the 4-D design. Even in that case, the 4-D approach is 3.6 times lighter on memory than the array-based design for a 4096-rank communicator. This ratio becomes 7.91 and 16 for 65,536 and 1,048,576 communicator sizes respectively.

V. EXPERIMENTAL EVALUATION

Our experimental setup is an 11-node InfiniBand cluster. Each node is a quad-socket AMD Opteron 6276 (Interlagos), having a total of 64 CPU cores and 128 GB of memory. The nodes are equipped with Mellanox ConnextX-3 QDR HCAs linked through 36-port Mellanox InfiniBand switches. All

the nodes run the 2.6.32 version of the Linux kernel and MVAPICH2-1.7. In all the tests, the linked list design, which is the default MVAPICH2 implementation, is used as the reference for speedup and memory overhead results.

A. Micro-benchmark Results

The micro-benchmark test program contains a single receiver and *n*-1 senders in an *n*-process job setting. The goal is to build a certain length of PRQ or UMQ before the receiver starts searching through them. We use a special tag value to pass a synchronization token that enforces a ring ordering of send or receive message posting. To build a long PRQ, the token is first given to the receiver which posts all its receives before the token circulates. To build a long UMQ, the token is given to the receiver only after all the senders are done with it.

Fig. 3 shows the speedup yielded by the array-based and our 4-D methods. When the requests are consumed from the bottom of the queue (reverse search) as in the test described in [10], the speedups are always above 1. For 10 pending messages per sender, the PRQ search is accelerated 44 times by the array-based design and 32 times by our 4-D design (Fig. 3-a). The UMQ is accelerated 31 times and 27 times respectively by the array design and the 4-D approach (Fig. 3-e). The reverse search is the worst possible case for the linked list design because the amount of traversal is maximized. In contrast, the linked list leaves no room for performance improvement for forward searches where the item of interest is always at the top (Fig. 3-b and Fig. 3-f). It can even be noticed that the speedups of the 4-D and the array approaches are sometimes slightly below 1 because their fixed multi-level traversal cost is not compensated for.

Those two tests are the extremes. There are two kinds of in-between situations. The obvious one, which is not presented due to space limitation, is when a fraction of the matches occur toward the bottom and another fraction occurs towards the top. Then, there is also the case where a fraction of the receives are MPI_ANY_SOURCE (Fig. 3-c and Fig. 3-d for PRQ; Fig. 3-g and Fig. 3-h for UMQ). MPI_ANY_SOURCE matches tend to produce three effects. First they somehow reproduce the forward search scenario in the PRQ as their matches occur at most after the number of pending messages per sender; that is, 1, 5 or 10 requests. Second, because MPI_ANY_SOURCE sub-queues are strictly linear no matter the queue design, they increase the fraction of linear searches. Then in the UMQ, MPI_ANY_SOURCE receives tend to perform complete searches against all the existing ranks. While those combined effects quickly decrease the search improvements, the faster approaches remain better than the linked-list unless the fraction of MPI_ANY_SOURCE gets close to 100%.

B. Application Results

The tested applications are Radix [20] and Nbody [20]. Radix and Nbody are interesting for the queue issue because they represent the two cases of superlinear and linear queue growth respectively. We emphasize that we did test our implementations for the correctness and ordering constraints assessment with applications that use MPI_ANY_SOURCE. The tested applications additionally present very shallow search depths and present no room for search optimization.

For Radix, the radix parameter is 16. The application is run respectively to sort 2^{28}, 2^{29} and 2^{30} 64-bit integers, as presented in the first column of Table 3. All three executions are done over 512 processes because our experiments show that the job size is less of an impact than the data size for Radix. Nbody is run over 256, 512, and 704 processes, respectively. The first column of Table 3 shows the job sizes appended to the names of Nbody. We provide in Table 3 some queue behaviour metrics that are used in the analysis of the performance results. The behaviours are presented for both the PRQ and the UMQ. The values in Table 3 are the averages computed over four iterations; this explains why certain maximum lengths are not integers. The Max Queue Length (MQL) represents the maximum queue length reached in the application. The UMQ MQL of Radix reaches several times its job size. As for Nbody, both its PRQ and UMQ grow almost linearly with the job size. Its PRQ MQL is actually exactly jobSize-1. The Max Average

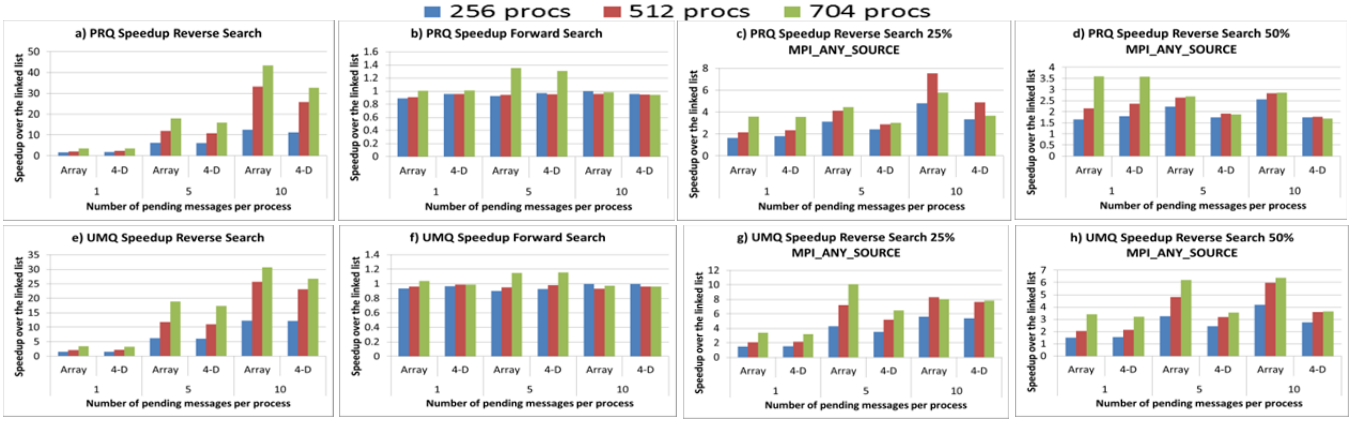

Fig. 3: PRQ and UMQ queue operation time speedup for the array-based and multi-dimensional design over Linked-list for the micro-benchmark

TABLE 3: QUEUE BEHAVIOUR DATA FOR RADIX AND NBODY

	PRQ				UMQ			
Application and parameters	Max Queue Length (MQL)	Average Queue Length (AQL)	Max Average Search Length (MASL)	Average Search Length (ASL)	Max Queue Length (MQL)	Average Queue Length (AQL)	Max Average Search Length (MASL)	Average Search Length (ASL)
Radix.n28	4	4	0.998	0.817	3075.4	437.871	1519.743	35.214
Radix.n29	4	4	0.998	0.823	6110	450.395	3425.498	43.852
Radix.n30	4	4	0.979	0.832	12577.6	493.384	7032.490	64.914
Nbody.256	255	255	0.978	0.948	247.25	85.745	17.449	3.572
Nbody.512	511	505.770	0.972	0.885	497.5	361.997	31.137	17.721
Nbody.704	703	702.968	0.861	0.817	694.75	656.653	39.982	30.423

Search Length (MASL) column shows the largest of the average search length returned by each process. The Average Search Length (ASL) shows the average search length over the whole job. The bigger these two metrics, the closer the search behaviour gets to the reverse search scenario presented in the micro-benchmark section; and the smaller they are, the closer it gets to the forward search scenario. Table 3 shows that the MASL of Radix reaches several thousands, making this application a good example of situation where fast queue traversal can make a noticeable difference.

Fig. 4 and Fig. 5 present the performance results for Radix and Nbody respectively. For Radix, the queue operation speedup reaches 8.09 for the array-based approach and 4.99 for the 4-D approach (Fig. 4-a). More importantly, the queue operation speedups are conveyed almost entirely to the communication and execution time improvements (Fig. 4-b, Fig. 4-c). For 2^{30} integers, the application ran more than 7 times faster with the array-based approach and more than 4 times faster with the 4-D approach. These large speedups in communication and execution times are due to the process of rank zero being a bottleneck and propagating the resulting latencies when the linked list queue is used. The MASL of Radix comes from its rank 0 and completely overshadows its ASL to be the key factor in its performance.

For Nbody (Fig. 5), the 4-D approach accelerates the queue processing time by up to 2 times. The array-based approach degrades the processing time instead. The array approach actually incurs an overhead when the queues become completely empty. The deallocation that ensues can be costly when it is frequent. These effects might not be compensated for when the search lengths are not large enough. The 4-D approach is shielded against that deallocation effect because it uses very small objects that are created and freed from various pools. Fig. 5-b and Fig. 5-c show that the 4-D approach improvement and the array-based design degradation do not noticeably impact the communication and execution times of Nbody. Nbody is very balanced as shown by the very small differences between the max metrics (MQL, MASL) and the average metrics (AQL, ASL) in Table 3.

There is finally a clear trend that the array-based approach is always more memory-intensive than the 4-D design (Fig. 4-d and Fig. 5-d). The memory overheads are very consistent with the expectations. For Nbody (Fig. 5-d) there is a clear increase of memory for the array when the job size increases. In comparison, the 4-D approach does not show that linear memory overhead growth. With Radix (Fig. 4-d), the 4-D approach shows a decreasing memory overhead when the number of sorted integers grows. That decrease of

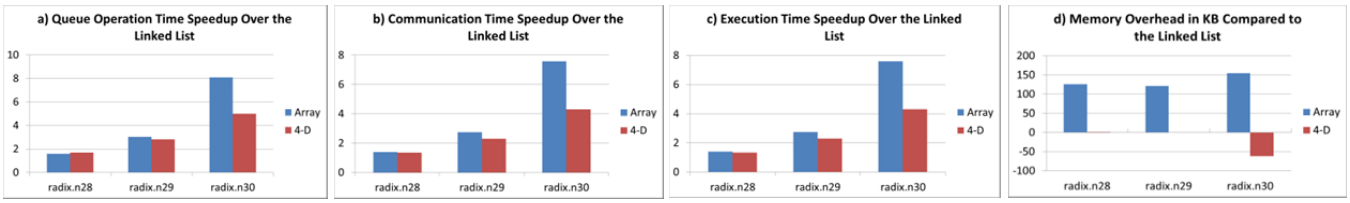

Fig. 4: Radix speedups and memory consumption for the array-based and multi-dimensional design over Linked-list (512 processes)

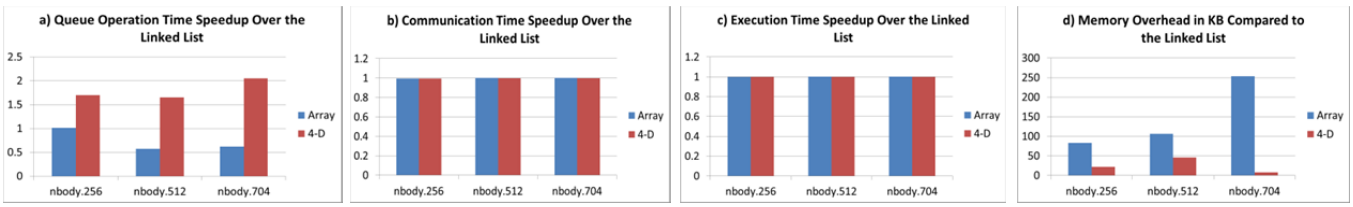

Fig. 5: Nbody queue behaviour, and speedup and memory consumption for the array-based and multi-dimensional design over Linked-list

the 4-D approach memory overhead is observed with Nbody as well (Fig. 5-d) for 704 processes. There is finally a negative memory overhead effect when there is less queue build-up due to the speed of processing. The memory overhead of a faster approach can end up being more than compensated for because it keeps low the average number of items in the queues. The negative memory overhead effect is fairly present with Radix.n30 (Fig. 4-d) when executed with the 4-D approach. The array approach can hardly yield the negative memory overhead effect because its differential memory consumption is strongly related to the communicator size instead of the actual number of queue items.

VI. Conclusion and Future Work

With more and more jobs expected to run on millions of CPU cores in the petascale and exascale computing era, MPI implementations must fix many little details that sink performance and increase memory consumption at scales. The MPI message queue mechanism is one of those details. The message queues are on the critical path of MPI communications, and a good implementation cannot afford to leave them unscalable.

It is possible to adopt an array-based design for which the operations which are usually expensive happen in $O(1)$. Unfortunately, this comes at the expense of increased memory consumption at large scales. In this work, we proposed a novel message queue mechanism that is scalable with respect to both speed and memory consumption. The approach, based on rank decomposition, mitigates the effect of job size on MPI queue operations. In certain cases, our proposal reduces the memory consumption while speeding up the queue processing.

For future work, we first intend to observe the behaviour of our proposed approach on larger-scale systems for jobs exhibiting linear or superlinear queue buildup. We also intend to study how we can integrate it with the array-based method in the same MPI implementation to truly approach the best of both speed and memory scalability.

Acknowledgment

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada Grant #RGPIN/238964-2011, Canada Foundation for Innovation and Ontario Innovation Trust Grant #7154. We thank the HPC Advisory Council and Mellanox Technologies for the resources to conduct this study.

References

[1] B. Barrett, R. Brightwell, S. Hemmert, K. Pedretti, K. Wheeler, K. Underwood, A. B. Maccabe and T. Hudson, " The Portals 4.0 Message Passing Interface – Draft 10/20/2011", http://portals4.googlecode.com/files/portals40.pdf

[2] InfiniBand Trade Association, http://www.InfiniBandta.org/index.php.

[3] MPI Forum, http://www.mpi-forum.org/

[4] MPICH2, http://www.mcs.anl.gov/research/projects/mpich2/

[5] MVAPICH/MVAPICH2, http://mvapich.cse.ohio-state.edu/

[6] Myricom, Inc. Myrinet Express (MX): A high performance, low-level, message-passing interface for Myrinet, July 2003, http: //www.myri.com/scs/MX/doc/mx.pdf

[7] Open MPI, http://www.open-mpi.org/

[8] Top 500 Supercomputers, http://www.top500.org/

[9] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur and J. L. Träff, "MPI on a million processors," *16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface,* 2009, pp. 20-30

[10] P. Balaji, A. Chan, W. Gropp, R. Thakur and E. Lusk, "The Importance of Non-Data-Communication Overheads in MPI," *International Journal of High Performance Computing Applications*, 2010, pp. 5-15

[11] R. Brightwell, S. P. Goudy, A. Rodrigues and K. D. Underwood, "Implications of application usage characteristics for collective communication offload," *International Journal of High Performance Computing Applications,* vol. 4, 2006, pp. 104-116

[12] R. Brightwell, S. Goudy and K. Underwood, "A Preliminary Analysis of the MPI Queue Characterisitics of Several Applications", *International Conference on Parallel Processing*, 2005, pp. 175-183

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

[14] R. L. Graham, R. Brightwell, B. Barrett, G. Bosilca and Pjesivac-Grbovic, "An evaluation of open MPI's matching transport layer on the cray XT," *14th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface*, 2007, pp. 161-169

[15] R. Keller and R. L. Graham, "Characteristics of the unexpected message queue of MPI applications," *17th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface*, 2010, pp. 179-188.

[16] M. J. Koop, J. K. Sridhar and D. K. Panda, "TupleQ: Fully-asynchronous and zero-copy MPI over InfiniBand," *2009 International Parallel & Distributed Processing Symposium,* 2009, pp. 1-8.

[17] F. Petrini, W. chun Feng, A. Hoisie, S. Coll, and E. Frachtenberg, "The Quadrics network: High-performance clustering technology", *IEEE Micro,* 22(1), January/February 2002, pp. 46–57

[18] M. J. Rashti and A. Afsahi, "A speculative and adaptive MPI Rendezvous protocol over RDMA-enabled interconnects," *International Journal of Parallel Programming,* 2009, pp. 223-246

[19] G. Santhanaraman, P. Balaji, K. Gopalakrishnan, R. Thakur, W. Gropp and D. K. Panda, "Natively supporting true one-sided communication in MPI on multi-core systems with InfiniBand," *9th International Symposium of Cluster Computing and the Grid,* 2009, pp. 380-387

[20] H. Shan, J. P. Singh, L. Oliker and R. Biswas, "Message Passing And Shared Address Space Parallelism On An SMP Cluster," *Parallel Computing,* vol. 29, February, 2003, pp. 167-186

[21] P. Shivam, P. Wyckoff, and D. K. Panda, "EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing", *Supercomputing*, November 2001, pp. 49

[22] N. Tanabe, A. Ohta, P. Waskito and H. Nakajo, "Network interface architecture for scalable message queue processing," *15th International Conference on Parallel and Distributed Systems,* 2009, pp. 268-275

[23] B. Tourancheau and R. Westrelin "Support for MPI at the network interface level," *8th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface*, 2001, pp. 52-60

[24] K. D. Underwood and R. Brightwell, "The impact of MPI queue usage on message latency," *International Conference on Parallel Processing,* 2004, pp. 152-160

[25] K. D. Underwood, K. S. Hemmert, A. Rodrigues, R. Murphy and R. Brightwell, "A hardware acceleration unit for MPI queue processing," *19th International Parallel and Distributed Processing Symposium.,* 2005, pp. 96b