A Fast and Resource-Conscious MPI Message Queue Mechanism for Large-Scale Jobs

Judicael A. Zounmevo and Ahmad Afsahi

Department of Electrical and Computer Engineering,
19 Union Street, Walter Light Hall,
Queen’s University
Kingston, ON, Canada, K7L 3N6
{judicael.zounmevo, ahmad.afsahi}@queensu.ca

Corresponding Author:
Ahmad Afsahi
Department of Electrical and Computer Engineering,
19 Union Street, Walter Light Hall, Room 403
Queen’s University
Kingston, ON, Canada, K7L 3N6,
ahmad.afsahi@queensu.ca
Tel: +1 613 533-3068

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 management mechanism whose operation time and memory overhead grow sub-linearly with the job size. We show why a novel approach is justified in spite of the existence of well-known and fast data structures such as binary search trees. 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. Finally, compared to the linked list-based queue, our proposed design yields cache miss rate improvements which are on average on par with the array-based design.

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

1 Introduction
The Message Passing Interface (MPI) [1] 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 November 2012, Sequoia, one of the most powerful petascale systems [2] had 1,572,864 CPU cores hosted in 98,304 compute 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 the MPI is the set of message queues encountered in most MPI implementations to cope with the unavoidable out-of-sync communications [3][4].

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 queue item, if any. If no matching is found, a message queue item (MQI) 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 MQI is posted in the PRQ. These message queues are required because senders are not required to synchronize with receivers before sending small (Eager type) or control messages.

It has been observed that the message queue length can grow in proportion to the job size [4][5][6][7]. The Portals 4 specification draft [8] mentions that the support for unexpected messages is necessary to avoid flow control and protocol overhead. Message queues are solicited in point-to-point, collectives and even modern Remote Direct memory Access (RDMA)-based implementations of Remote Memory Access (RMA) operations [9], which marginally resort to point-to-point operations at middleware-level. Actually, the UMQ is used so frequently that it has been qualified as the most crucial data structure in MPI [4]. 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>. In certain implementations such as Open MPI [10] it is a proper superset of that tuple. ContextId designates the communicator. The rank is the source process rank for a PRQ item, and the receive process rank for a UMQ item. As for the tag, it is useful whenever the same process (sender or receiver) can have more than a single pending message/MQI 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 MPI message queue 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; 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. We also show that our proposed approach is as efficient as the array-based design in improving cache miss ratios when message queue search depths become large.

This paper extends our previous work in [11] in a number of directions. We have extended the complexity analysis study of the proposed message queue structure as well as the linked list and array-based designs. We have also included a runtime cost analysis that complements practically and intuitively the asymptotic complexity formulas. We explain, analytically and with test data, why other data structures including search trees are not optimal in building a fast message queue container specifically for MPI. In addition, this paper proposes a heuristic to determine a size threshold over which our multidimensional data structure becomes the message queue structure of choice over a mere linked list design. Finally, this paper provides an extensive set of comparative experimental results and analysis of the proposed design with multiple active communicators. We also present cache miss improvement
results and their analysis in this paper. Moreover, we use a memory measurement technique that is less sensitive to noise in the processes. The method instruments the MPI implementations to compute the memory consumption of the different message queue structures.

The rest of this document is organized as follows. Section 2 presents the related works. Section 3 discusses the motivations behind this work. Section 4 presents the proposed message queue approach. Section 5 presents comparative runtime complexity analyses of the newly proposed approach, and numerically compares it against the linked list and the array-based designs. Section 6 briefly compares the proposed multidimensional message queue with well-known search data structures. Section 6 also makes a case for our new design. In Section 7, we present our experimental results. Section 8 concludes the paper and points to the future work.

2 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 [7]. The Cray MPI [12] implementation offloads most of its receive-side matching operations on the Portals network infrastructure [8], which it is layered on. Though, according to the Portals specification draft [8], a resource exhaustion risk exists for large message queues. MPI implementations over previous versions of Portals abort the application when queue resource exhaustion occurs. In the current version 4 of Portals, this situation is handled by dropping packets; and the end result is a slowdown.

Quadrics [13] and Myrinet [14] networks also support MPI message matching by offloading it onto their NICs. They resort to on-NIC threads and memory. For the InfiniBand [15] interconnect, TupleQ [16] has been proposed where a verb-level Shared Receive Queue (SRQ) is created for each <contextId, rank, tag> tuple to match messages in hardware. TupleQ was not meant for queue processing but for handling the point-to-point Rendezvous protocol [17] in hardware. For a receiver process P, the SRQ requirements of TupleQ grow in $O(c \times r \times t)$, with $c$, $r$, and $t$ respectively being the number of contextIds, ranks and tags which have ever reached $P$. TupleQ is therefore not suitable for large applications, even when there is no queue buildup.

A hardware proposal has also been reported in [18] where a NIC has been modified to process the queue operations with an associative list processing unit. The NIC is equipped with a local SRAM. The tests presented in their work are only simulations made on a custom stripped-down implementation of MPI-1.2. Finally, a NIC–associated accelerator has been proposed [19] to store message headers or the entire messages (in the case of small messages) into a low-latency dedicated buffer. Unfortunately, the time to process long UMQ and PRQ on embedded processors has been reported to be substantially longer than host-based implementations, simply because these processors are slower [7][18][20]. More importantly, NIC or accelerator memory is usually a very limited resource which can become a scalability barrier when large queues build up [19].

As a purely software solution, hash tables have been proposed to handle the message queue operations [21][22]. They are however reported to have prohibitive insertion times [18]. The work in [19] even mentioned that hashing can actually have a fairly negative impact on communication latency in almost all situations. Linked lists are used in MPICH [23] and many of its numerous derivatives. Linked lists don’t scale over large jobs when lengthy searches occur. An array-based approach is used Open MPI [10]. While arrays are fast, they are not memory-scalable.
3 Motivations

3.1 The Omnipresence of Message Queues Processing in MPI Communications

To guarantee message queues avoidance, the concept of reception must be totally absent; that is, messages should be able to reach their final destination without any action from the receiving process. The MPI one-sided operations form the only communication model that comes close to avoiding the message queues. Though, even in presence of advanced network features such as RDMA, the MPI one-sided communications are still unable to internally one-sidedly transfer non-contiguous derived data type messages. To avoid the adverse effect of issuing a large number of potentially small internal communications for a single application-level communication, non-contiguous messages are internally packed at the origin process, shipped into an intermediate buffer in the target process before being unpacked into their final destinations. The size of the intermediate buffer must be communicated to the target in some kind of buffer request message. Then the target must reply with a buffer response message which contains information required for RDMA transfer. This whole mechanism internally voids the apparent application-level one-sidedness of the communication because it requires a reception. Message queue processing intervenes because of the reception.

On top of the non-contiguous data type issue, the MPI_Accumulate family of functions cannot be fulfilled to completion without internally involving the target process for either sending its side of the operands or performing the operation locally [24]. The MPI_Accumulate issue exists without respect to the non-contiguity of the data.

The prevalence of message queues in two-sided communications is obvious even from a purely semantic point of view. Even the flavours of MPI_Send, such as MPI_Ssend and MPI_Rsend, which bear synchronicity in their behaviours do not guarantee the absence of message queuing. In particular, MPI_Rsend can only ease, without guaranteeing, the absence of UMQ items, thanks to its semantic. At the same time, MPI_Rsend tends to encourage PRQ items as it requires the matching receive to be pre-posted. MPI_Ssend on the other hand has the exact same implementation requirements as MPI_Send for large messages where the Rendezvous protocol is used [25]. When the Eager protocol is used for small messages, MPI_Ssend, compared to the vanilla MPI_Send, simply adds an extra control message communication from the receiver to the sender. In any case, the receiver in the case of MPI_Ssend is still left with at least the same amount of message queue processing as in MPI_Send.

As for collective communications, they internally build their message scheduling on top of two-sided routines and are subject to most of the message matching scenarios inherent to two-sided communications.

3.2 The Impact of Next Generation HPC Performance-Oriented Features on Message Queues

At least two trends in MPI usage will increase even further the prevalence and usage of message queues in the next generation HPC jobs. Each of them responds in a certain way to the need of MPI and MPI applications to adapt to ever increasing system scales. The first of these trends is the higher prevalence of non-blocking operations. MPI has recognized the importance of non-blocking two-sided communications as a parallelization enhancing strategy since the first version of its specification. The one-sided communications, later introduced in MPI-2.0, are exclusively non-blocking. Finally, non-blocking collective communications have known a substantial advocacy in the HPC research community.
which points out the scalability issues associated with the blocking nature of MPI-2.2 collectives [26]. Non-blocking collectives have finally been introduced in the recent MPI-3.0 specification.

Non-blocking communications favour massive pre-postings and therefore, massive PRQ buildups. Consider for instance a set of blocking receives; and a set of non-blocking receives. In the case of the blocking receives, the PRQ will never contain more than a single item at a time; because no receive will be posted before the previous one completes. In the case of the non-blocking receives, all the communications can be posted even before the first byte of the first receive arrives. Consequently, non-blocking collectives are bound to generate a large number of PRQ items.

The second trend that will rise message queue processing is the need to move toward multithreaded MPI and hybrid use of MPI with other paradigms such as Intel TBB [27] and PGAS [28] languages at large scales [29][30]. Message progression in MPI is a global activity; leading to multithreaded MPI processes having centralized message queues shared among all the threads. Therefore, multithreaded MPI communications increase the occurrence of out-of-order message discovery, which translates into a certain thread discovering more UMQ items meant for messages destined for another thread in the same process.

### 3.3 Performance and Scalability Concerns

MPI performance tuning strategies must be concerned with message queues, which accounted for up to 60% of communication latency in certain tests [20]. In fact, communication-intensive HPC jobs are simply as fast as the underlying MPI message queue processing. While ensuring fast communications is an obvious requirement for the HPC-related paradigm that is MPI, memory consumption is another issue that matters at any scale. At runtime, MPI is a communication middleware; i.e., a layer mostly meant to merely transfer the data brought and manipulated by the actual HPC application. As a result, the HPC process does not expect to lose a substantial amount of its available memory to its communication substrate. MPI gains by being lean at large scales to give room to the potentially large amount of application data.

At extreme scales, a few issues such as contention avoidance invariably get a lot of attention. Less obvious however, is the issue of load per CPU core when the job size grows. This issue is totally orthogonal to the much feared contention problem; but it can be as detrimental. For certain resources such as memory, a linear degradation per process is a quadratic degradation job-wise. Furthermore, scalability is rarely strength-controlled; it tends to be weakness-controlled instead. For instance, if a message queue architecture can remain reasonably fast at 100 millions CPU cores while its memory consumption becomes prohibitive at 1 million CPU cores, then its scalability is effectively 1 million CPU cores.

In order to speed up message queue processing, hardware-based approaches have been attempted. Unfortunately, accelerator or NIC processors have been reported to be slower than host CPU [7][18][20] for message queue processing. More importantly, hardware solutions are inherently non scalable. The main issue is resource limits [19]. Unlike host-based approaches which can access a virtually limitless amount of memory, embedded processors are limited to their accessible physical memory which is usually far smaller than host RAM. In situations of embedded memory exhaustion, hardware solutions become a scalability bottleneck. Solutions like embedded memory overprovisioning are expensive; but more importantly, they do not scale throughout generations of system size.

The fastest possible software approach could host the MQIs in a fixed-size array for $O(1)$ accesses. One can trivially notice that resizable arrays suffer performance degradation due to reallocation and moving. Therefore, any mention of array refers to the C-style fixed-size variant. Array indexing requires a contiguous key whose maximum value is known ahead of its allocation. Out of the message queue search
key tuple, only the rank fulfills these conditions. The rank always ranges contiguously from 0 to \( n-1 \) for any communicator of size \( n \). In comparison, the MPI standard does not specify any starting and end value for contextIds. As for the tag, it bears no contiguity constraint but more importantly, it ranges from 0 to MPI_TAG_UB, which can be prohibitively large because it reaches INT_MAX in many MPI implementations [10] [31]. The resource issues inherent to allocating an array of two billion slots per communicator per process are obvious; tags are therefore not exploitable. In any case, empirical observations show that message queues grow mostly with job sizes [4][5][6][7], making a rank-based optimization the most promising.

Unfortunately, the once-for-all allocation scheme of arrays corresponds exactly to a linear degradation of memory consumption and is therefore not scalable. This allocation scheme is not even correlated with the actual needs; but with the communicator size. Unless each process in the job exhibits a fully connected communication pattern where each process talks to every other process in the communicator, most indices of the array will never be used. We emphasize that well-crafted MPI programs avoid the fully-connected communication patterns as much as possible to avoid its inherent scalability problems. Instead, most good MPI programs prioritize localized small sub-groups of communications. Furthermore, the current and next generation HPC programs are being more and more demanding in terms of development time and performance tuning. These programs are crafted to solve more complex problems. They are also meant to run on larger systems, all subject to the diminishing return effect that characterize parallel computing. As a consequence, as much as possible, new HPC programs are composed with existing time-tested, feature-rich and highly tuned domain-specific reusable HPC parallel libraries such as PETSc [32] and Blast [33]. A strongly recommended good practice [34] requires those reusable HPC libraries to duplicate MPI_COMM_WORLD in order to isolate their internal communications from the application and other libraries in the same program. Depending on the number of active parallel libraries, a simple MPI job can therefore have a significant number of job-size communicators; each having at least one contextld. As a result, very large petascale and upcoming exascale MPI jobs will waste a large amount of scarcely used memory if array-based message queues are adopted. If the job is topology-aware, each process will pre-allocate even more potentially unused memory for message queues. Unfortunately systems growth is generally being accompanied by a certain reduction of the amount of memory per core [30]. It is ok for the overall memory consumption of the whole job to grow linearly with the system size; this is a case of a growing demand coupled with a growing resource. However, it is not practical for the memory consumption per process or per CPU core to grow linearly with the system size. For instance, consider an HPC job of \( n \) processes, whose overall dataset memory requirement \( D \) is the maximum the system can host in its aggregated RAM. Assuming that the data is distributed equally over the processes, each process needs an amount of memory \( d=D/n \). When both the job size and dataset grow 10 times, each process is still expected to host approximately the same \( d=(10D)/(10n) \). The memory consumption pattern of an array-based message queue voids this expectation because the available RAM per process gets smaller and smaller than \( d \) when the job size grows.

Purely on-demand memory allocation achieves the best memory scalability behaviour. Linked list-based approaches achieve as much as possible this second form of scalability. Unfortunately, they exhibit a linear degradation with respect to speed of processing. For instance, it is reported in [3] that the linear traversal of a message queue of 4095 items took up to 140 ms on a Blue Gene/P system. Linear searches are all but acceptable at petascale and exascale. With \( O(n) \) searches and at fixed CPU core speed, MPI message queues linked list searches, and consequently communications, simply get slower on the exact same CPU core when the system size increases. Aggravating matters is the need for next generation...
supercomputers to become substantially more energy-efficient. Power issues are one of major challenges mentioned in [29] for the roadmap towards exascale systems. Prominent supercomputer builders such as IBM have even long opted for slower but more energy-efficient CPUs [3]. This trend means that slow queue processing will get even slower in HPC processes doing exactly the same job.

3.4 Rethinking Message Queues for Scalability by Leveraging MPI's Very Characteristics

We present a new MPI message queue design which is simultaneously fast and resource-conscious even at large scale. Our approach draws its advantages from a few domain-specific observations. In particular, on top of observing the properties already mentioned for the rank, we noticed that MPI message queue operations follow strictly the following patterns:

1. Search PRQ and Delete MQI if found; otherwise Insert MQI into UMQ
2. Search UMQ and Delete MQI if found; otherwise Insert MQI into PRQ
3. Search UMQ and Delete MQI if found
4. Search PRQ and Delete MQI if found
5. Search UMQ (and simply return search status; do not delete)

The first two scenarios occur for regular communications; including those that happen inside MPI without being initiated by the application. They are by far the most common. Scenario 3 can happen when MPI_Cancel is called from the sender-side. When that happens, control messages meant for the sender are retrieved, if they exist, and discarded. In the new MPI-3.0, scenario 3 can also happen when MPI_Mprobe and MPI_Iprobe are invoked. Scenario 4 happens when MPI_Cancel is issued from the receive-side. Scenario 5 occurs for MPI_Probe and the new MPI_3.0 MPI_Iprobe.

Our design only has to be optimized for the above-listed patterns. We first observe that isolated insertions and deletions never occur in MPI message queue operations. We also observe that searches are part of every single use case. Consequently, a goal of the design strategy is to fundamentally leverage the prelude occurrence of searches as strength for the other operations that it invariably precedes. For instance, the use cases 1 and 2 prompt us to avoid using two separate data frameworks for the PRQ and the UMQ. By hosting them in the same searchable object, a search in the UMQ (use case 2) can serve as a free ride towards the insertion point where MQI would be put in the PRQ if the match is not found in the UMQ. The same reasoning applies for use case 1.

Our main speed of processing goal is to perform localized searches by skipping altogether large portions of the message queue for which the search is guaranteed to yield no result. The new queue design is therefore focused on making the identification of those unfruitful portions easy to detect deterministically without error. To achieve that purpose, we use a multidimensional approach which allows the exploitation of a coordinate mechanism. The coordinate mechanism operates jumps that successively narrow down the search space after the traversal of each dimension. A jump is the mechanism of reaching a coordinate along a dimension. In n dimensions, the rank is transformed in the coordinate \((c_{n-1}, c_{n-2}, ..., c_0)\). We define dimensionSpan as the maximum number of distinct coordinate positions on each dimension. This work uses the same dimensionSpan on all the dimensions. The rank is expressed as \(rank = \sum_{i=0}^{n-1} c_i \times \text{dimensionSpan}^i\). To explain the efficiency of the jump mechanism, let’s consider a communicator with 1000 ranks. The ranks range from 0 to 999. A linear search could have to scan all the 1000 ranks before finding an MQI of interest. If dimension is 3; then dimensionSpan is 10. Any rank can be found by scanning at most 10 positions on each dimension; leading to 10+10+10 = 30 rank traversals in the worst case; instead of 1000 for a vanilla linear linked list. Our secondary speed of
**processing goal** is to ensure that any chunk of linear traversal required in the proposed data structure remains very short. Linear searches, as used in the linked-based design, are actually not unacceptable when they are short; they become harmful only when they are long.

Our **memory consumption behaviour goal** is to steer away from any fixed full-provision memory allocation scheme, as used in the array approach, which becomes impractical at large scales. We seek to allocate structural objects to organize hosted MQIs in a search-friendly manner. Those structural objects are not only allocated on-demand but they are used in a way that provides a quick amortization of the memory that they consume.

### 4 New Message Queue Design for Large Scales

This section exposes our new scalable message queue design for MPI. In the rest of the document, we use the term *queue* generically to designate any data structure meant to host MPI message queue items. As a reminder, there is little constraints on contextIds and tags; making them difficult to reason about for efficient resource management. The reasoning which sustains our proposal is mostly built over the rank because 1) it is the principal message queue growth factor; and 2) it bears the already mentioned constraints of contiguity and upper value. For message queue management purposes, we represent a contextId with a *ContextIdLead* object which is attached to its own message queue object.

#### 4.1 A Scalable Multidimensional MPI Message Sub-queue

High dimensionality tends to favour larger speedups for extremely large searches. A concrete example with a communicator of fixed size $10^{16}$ follows. $10^{16}$ is not a realistic communicator size; it is purely meant to show very easily the effect of higher dimensions. We choose only even dimensions so as to make the computations straightforward. With 2 dimensions, $\text{dimensionSpan} = 10^8$. The worst case traversal time for this 2-D structure would be $\text{dimensionSpan} + \text{dimensionSpan} = 2 \times 10^8$; and the speedup is $10^{16} / (2 \times 10^8)$. With 4-D, the speedup is $10^{16} / (4 \times 10^8)$; and with 8-D, the speedup is $10^{16} / (8 \times 10^8)$. We notice that the speedup in the worst case increases very fast with the dimensionality. There is however a few problems in using high dimensionality. The first one is that higher dimensionality has a higher fixed initial search cost; leading to poorer performance for shallow search depths. Higher dimensionalities have the adverse effect of forcing longer and longer traversals before reaching the first MQI; as each dimension must absolutely be traversed without respect to the coordinate decomposition of the rank. With each dimension represented as an ordered linked list, the rank 1 for instance is expressed as $0 \times \text{dimensionSpan}^1 + 1 \times \text{dimensionSpan}^0$ in 2-D; and this expression means that the axis representing $\text{dimensionSpan}^1$ must nevertheless be visited at position 0. In $n$-D, all the axes representing $\text{dimensionSpan}^{n-1}$ to $\text{dimensionSpan}^1$ must still be traversed for rank 1. As a reminder, each dimension ranges from 0 to $\text{dimensionSpan} - 1$; meaning that a coordinate of 0 on an axis does not mean that the dimension is skipped during the traversal. Furthermore, how many coordinate objects are deleted or created following insertions or deletions depends on the dimensionality. Actually, each (discrete) coordinate on each dimension is an object. For instance if a rank decomposes into $(c_{n-1}, \ldots, c_4, c_3, c_2, c_1, c_0)$ in a $n$-D space, its insertion will lead to the creation of an object at position $c_i$ along the axis representing $\text{dimensionSpan}^i$ if no MQI having that coordinate for that dimension currently exists in the queue; and this must happen for every dimension. The deletion of an MQI must also trigger the deletion of every coordinate object that the disappearing MQI was the only one to bear. For this work, we chose 4 dimensions to considerably mitigate these effects on shallow searches, insertions and deletions.
The 4-dimensional (4-D) data-structure described in this section is only meant for large message queues. We split the rank in 4 slices (Figure 1-a). The rank is decomposed in the coordinate quadruple \((c_3, c_2, c_1, c_0)\) such that rank \(= c_3*32^3 + c_2*32^2 + c_1*32 + c_0\). \(\text{dimensionSpan} \) is a power of two in order to allow fast bitwise decompositions. Its relation with communicator sizes is determined by the formula 
\[
\text{dimensionSpan} = 2^\left(\log_2(\text{communicator size})/4\right).
\]
\(\text{dimensionSpan} \) is 4, 8, 16 or 32 for the intervals \([1, 256], [257, 4,096], [4,097, 65,536] \) and \([65,537, 1,048,576] \), 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 \(\text{dimensionSpan}\). The secondary speed goal is built upon \(\text{dimensionSpan}\). For any given communicator size, \(\text{dimensionSpan}\) determines the typical length of chunks of linear searches inside our proposed data structure. Since \(\text{dimensionSpan}\) is always reasonable and grows very slowly, linear search lengths are kept under control.

For the sake of conciseness, we choose a single \(\text{dimensionSpan}\) (e.g., 32) 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 this approach all 4 coordinate components are used in the jump process to reach a pair of limited length UMQ and PRQ where all the MQIs bear the same rank and only the tag varies. This first approach is beneficial if processes usually maintain several pending messages, resulting in several MQIs 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 MQI whose ranks vary only by the least significant coordinate in the same linked list of UMQ or PRQ. As a reminder, each slice of the rank can only take on \(\text{dimensionSpan}\) 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.

In this second approach, which we adopt because it is more memory-efficient, the three most significant coordinates \(c_3,c_2,c_1\) of the rank form a cube (Figure 1-b). Only these first three most significant coordinates participate in the jump process; the least significant coordinate is ignored. We define a Plane as the object reached by the most significant coordinate \(c_3\), represented in black in Figure 1. A Plane is the first jump point in the 4-D data structure. The second and third most significant coordinates \(c_2\) and \(c_1\), respectively represented in blue and green in Figure 1 encode respectively the X and Y coordinates in each Plane. We define a Segment as the final jump point, uniquely addressed by the triplet \((c_3,c_2,c_1)\). A Segment is an object which contains a pair of limited length PRQ and UMQ where ranks can vary only by their \(c_0\) coordinate.

With each dimension represented with a linked list, let us consider a search in a queue of 1,048,576 distinct ranks; each having a single pending message. Even in the worst case, the sought item will be found in \(32+32+32+32=128\) scans; leading to a speedup of \(1,048,576/128=8192\). Such a tremendous speedup is entirely tributary to the dimensional decomposition. If processes have more than a single pending message, the first three dimensions are still searched in a maximum of \(32+32+32\) scans; the last one could be searched in more than 32 scans. Furthermore, every coordinate on each dimension is allocated only if at least one MQI exists whose decomposition bears that coordinate value on the dimension; leading to a complete on-demand memory allocation scheme.
We can further notice that by representing any of the first three dimensions with an array, its scan time could drop from 32 to 1, leading to further speed improvements; this time with a certain memory trade-off. As a reminder, the last dimension \( C_0 \) cannot be represented by an array because it does not participate in the jump process. In particular, the 3-cube instead of the 4-cube approach allows that last dimension to represent the same coordinate value more than once in situations where the same rank appears with different tag values. The dimension-wise (not overall) speedup yielded by the array use on any dimension depends on how many distinct coordinates \( n \) are usually in use on that dimension. That speedup is actually \( n \). If all the first three dimensions bear the same number of in-use coordinate values, as far as speed is concerned, the dimension to represent with an array does not matter. In particular, if the search must scan the same number \( n \) \((n \leq 32)\) of positions on each dimension, then representing Planes, Xs or Ys with arrays respectively lead to \((1+ n + n + n)\), \((n +1+ n + n)\) or \((n + n +1+ n)\) overall scans. The likelihood of a dimension having more in-use coordinate values than another one is hard to generalize; it depends on the application and how sparsely or densely its ranks are represented in the queue. It helps to notice that two ranks separated by at least \(32^3\) falls in two distinct Planes. If the separation amplitude is at least \(32^2\) without being a multiple of \(32^3\), they fall in two distinct Xs. Finally, if the separation is at least \(32\) without being a multiple of \(32^2\) or \(32^3\), they fall in two distinct Ys. However, while all 4-D queues will always have all their \( dimensionSpan \) X and Y coordinates present, most will usually have less Planes than \( dimensionSpan \). The number of Planes is linked to how close the communicator size is to the upper limit of its interval. For instance, with \( dimensionSpan=32 \), the smallest communicator size (65,537) needs \( \lceil 65,537/(32^2) \rceil \) Planes, which is 3. This means that the communicator is big enough to completely fill Planes of coordinates 0 and 1 and then overflow into Plane of coordinate 2. This also means that all the 32 Xs and all the 32 Ys are in use in Planes 0 and 1. X and Y are thus the two dimensions that could potentially benefit from an array representation. The choice of array use on any dimension must also consider the resulting memory degradation impact. Lower dimensions bear a large memory penalty. For instance, there is a total of \(32^3\) (i.e., 32,768) Ys and only \(32^2\) (i.e., 1024) Xs for a communicator of 1,048,576 distinct ranks. Therefore, we elect to represent the X dimension with arrays because they bear the smallest memory penalty. Planes and Ys are represented with ordered linked lists. The worst case
search can then be performed in \(32+1+32+32 = 97\) scans; for a speedup of \(1,048,576/97 = 10810\). We emphasize that while the use of array for one of the dimensions yields some additional speedup, the bulk of the improvement is provided by the breaking down into dimensions. In this particular case, the array brings an additional speedup of \(10810/8192=1.32\). In general, when the communicator size \((\text{size})\) grows, the speedup \(S_p\) associated with just the dimensional decomposition tends towards

\[
\lim_{\text{size} \to \infty} S_p = \lim_{\text{size} \to \infty} \frac{\text{size}}{4 + \text{dimensionSpan}} = \lim_{\text{size} \to \infty} e^{\frac{1}{4}(\text{dimensionSpan} + \log(\text{size}) - \log(4))} = \infty.
\]

In comparison, the additional speedup yielded by representing a dimension with an array is asymptotically capped by

\[
\lim_{\text{size} \to \infty} \frac{4 + \text{dimensionSpan}}{3 \times \text{dimensionSpan} + 1} = \frac{4}{3} = 1.33.
\]

To understand how this proposal achieves localized searches, let's consider an MQI bearing the rank 80,000 in a communicator of 1,048,576 ranks. The required \(\text{dimensionSpan}\) is 32. 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, \(31 \times 32^3 + 31 \times 32^2 + 31 \times 32\) 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 MQI searches the UMQ, all the ranks must be probed. Then if a match is not found, because the MPI\_ANY\_SOURCE MQIs 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 demand at the rhythm of communicator creation/destruction and queue length changes.

### 4.2 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 when they are always inserted into from the tail pointer and always searched from the head pointer. For our 4-D proposal, that order is kept as well, as long as there is no MPI\_ANY\_SOURCE item. As a reminder, for the 4-D design, queue items of the same ranks are always in the same common short linked lists. For the general case, we add a monotonically increasing sequence number to the key tuple of all the queue items 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.

### 4.3 Size Thresholds Heuristic for Sub-Queue Structures

There is a small fixed cost for searching any queue item in the 4-D structure; this cost is made of decomposing the rank as well as finding the right Plane and Segment. For insertion, this cost is made of creating a new Plane and/or a new Segment if they were not present before the new queue item could be inserted. Finally, for deletion, a Segment that becomes empty must be removed from the 4-D queue; same
goes for a Plane that becomes empty. The 4-D structure becomes better than the linked list only when
searches are (or could potentially be) long enough to overbalance all those small fixed costs. As a
reminder, the 4-D container is for large jobs, so this constraint is reasonable. We cannot predict the
average search lengths before searches actually happen. However, we always know how big a
communicator is at the time of its creation. The communicator size is the maximum number of distinct
ranks that the queue can hold; it is also the maximum queue size if each process has a single pending
message. Since search lengths are unknown ahead of time, we use those maximum possible queue sizes.
We define a threshold below which the sub-queue associated with a ContextIdLead is merely a linked list
(Figure 2). If the communicator size is above that threshold, the 4-D data structure is used as sub-queue
(Figure 2). We define the PointerTraversalLengthThreshold of a contextId as the number of MQI of
distinct ranks after which the last item in a linear sub-queue would incur more pointer traversals than
searching any item in the corresponding 4-D container. Then we define the threshold as
PointerTraversalLengthThreshold*Adjustment. Adjustment incorporates aspects that are unpredictable
such as how often small objects (Planes, Segments) are created and freed. Furthermore, a linked list
traversal is very repetitive. In comparison, with an equal number of pointer traversals, the 4-D structure
requires more complex and diverse branchings. Consequently, the threshold cannot be lesser than
PointerTraversalLengthThreshold. As a result, the minimum value of Adjustment is 1.0.

For all our tests related with this work, we used an Adjustment value of 2.0. The value is
empirically based on microbenchmark-level trial-and-error with average message queue search depth
being half the message queue size. Concrete values of the threshold are computed as follows. For
instance, when the communicator size is in the interval [65,537, 1,048,576] with dimensionSpan being 32,
searching an item would require in the worst case 32+1+32+32 for Plane, X coordinate, Y coordinate and
Segment queue scanning. PointerTraversalLengthThreshold is thus 97 and the threshold is 194. That
threshold becomes 26, 50 and 98 respectively for communicators whose sizes are respectively in the
intervals [1, 256], [257, 4096], [4097, 65536].

Figure 2: New overall MPI message queue design

PointerTraversalLengthThreshold is not user-modifiable; it is a characteristic of the 4-D structure
for each communicator size. PointerTraversalLengthThreshold is computed automatically from the
communicator size. Adjustment is user-modifiable. However, it incorporates aspects which are impossible
to generalize even with a given MPI implementation. It accounts for variables such as how often the
message queue gets empty. Another variable that could influence Adjustment could be CPU speed. For
instance, a 2.2GHz Xeon is probably less sensitive to a certain length of linear traversals than an 850MHz
PowerPC. As a result, everything else being equal (including memory access latency, cache miss ratio, object allocation frequency, etc.) the PowerPC gains by using a low Adjustment so as to switch to the 4-D structure earlier. In general, Adjustment is tied to both the specific application and the specific architecture.

Though, it is important to notice that Adjustment does not really matter at large scales. Figure 3 shows the threshold relation to performance with hypothetical examples. The CPU consumption growth pattern shown in Figure 3 does not matter for the purpose of the example; the curves “Slow at scale” (SAS) and “Fast at scale” (FAS) could have been anything but linear. We define $S_c$, $T_a$ and $T_h$ respectively as the actual communicator size, the accurate threshold value and the heuristic-approximated threshold value. As a reminder, all the thresholds are expressed in communicator size; and $T_h$ is the aforementioned PointerTraversalLengthThreshold*Adjustment. In Figure 3, lower CPU consumption is better because it means faster queue operation. The aim is to:

- Choose SAS (or the linked list in our particular case) if $S_c < T_a$
- Choose FAS (or 4-D in our particular case) if $S_c \geq T_a$

Since $T_a$ is not known, the choice of the sub-queue type depends on $T_h$. The following scenarios can then happen:

1. $T_h < T_a$:
   Consequences:
   a. If $S_c < T_h$: The right sub-queue (SAS) is chosen because $S_c < T_a$ is true.
   b. If $T_h \leq S_c < T_a$: FAS is chosen while SAS should have been chosen.

   ![Figure 3: Showing threshold accuracy effect on performance](image-url)
c. If $T_a \leq S_c$: The right sub-queue (FAS) is chosen.

t. $T_b > T_a$:

Consequences:

a. If $S_c < T_a$: the right sub-queue (SAS) is chosen.
b. If $T_a \leq S_c < T_b$: SAS is chosen while FAS should have been chosen.
c. If $T_b < S_c$: The right sub-queue (FAS) is chosen.

One can make the following observations:

1. An inaccurate threshold leads to the wrong choice only when $S_c$ falls between $T_a$ and $T_b$.
2. $S_c$ falling between $T_a$ and $T_b$ usually implies that $S_c$ is still in the neighbourhood of $T_a$ in which case performance variations due to the choice of SAS or FAS can even be assimilated to or balanced out by noise.
3. The extent $|T_a - T_b|$ of the threshold inaccuracy would not matter for large communicators; because $S_c$ will be far beyond both $T_a$ and $T_b$; i.e., $S_c$ will not fall between the two threshold values. Threshold inaccuracies will therefore never have any impact for the very use-case for which the 4-D is being proposed.

5 Performance Analysis

This section presents a runtime complexity, memory consumption behaviour and pointer traversal intensiveness analyses for the new proposal. For comparison purposes, we establish the same analyses for an array-based message queue design that we consider as the reference in terms of speed of processing; and a linked list-based approach that we consider as the reference in terms of memory consumption behaviour.

We define a message queue item (MQI) as follows:

```c
typedef struct MQI_t
{
    struct MQI_t *next;
    int contextId;
    int tag;
    int rank;
    /* other MPI implementation-specific data */
    ...
} MQI_t;
```

The linked list design is simply made of a pair of linked lists; one for the PRQ and another one for the UMQ. Each list has a head and a tail pointer. Figure 4 shows the array-based design which, because it is built over rank numbers, is a per-communicator approach too. The array-based design resorts to `ContextIdLeads` as well. Each `ContextIdLead` associated with a communicator of size $n$ has an array of $n$ `RankHead` data structures. The `RankHead` at position $i$ in the array is associated with the MQIs coming from or meant for the process of rank $i$ in the communicator. 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. For instance, for a `ContextIdLead` whose contextId is $c$, all the MQIs hosted in the PRQ head of `RankHead` $i$ are such that MQI->contextId=$c$ and MQI->rank=$i$; but MQI->tag can vary. MQIs bearing `MPI_ANY_SOURCE` are hosted in a separate linked list-based sub-queue because their rank field cannot be used as array index. The MPI message ordering constraint is once again enforced with a sequence number approach. It is important to mention that Open MPI uses an array-based approach, but in a
context that mitigates its associated memory waste drawback. In each process and for each communicator, Open MPI allocates once for all an array of data structures to host a set of information about the other processes. These arrays, while not dedicated for queue processing, are nevertheless leveraged for that purpose. The linked list-based approach has been adopted by MPICH and many of its numerous derivatives.

![Diagram of a linked list-based queue design for a communicator of size n](Figure 4: Array-based queue design for a communicator of size n)

5.1 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 [31]. We implemented the array-based method in MVAPICH2 for comparison purposes.

In all the analysis, 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 queue item 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 contextIds in the considered process
- $n_j$ the number of distinct valid ranks in the contextId $j$.
- $t_i$ the number of queue items associated with the same rank $i$ in any queue of the same contextId.
  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 $i$.
- $a_j$ the number of MPI_ANY_SOURCE queue items associated with the contextId $j$.

5.1.1 Asymptotic Complexities

The Linked List-Based Queue

Because the scanning goes through all the $t_i$ queue items of all the $n_j$ ranks of all the $k$ contextIds, searching the linked list happens in

$$O(\sum_{j=0}^{k-1} \sum_{i=0}^{n_j-1} t_i)$$

(1)
This search complexity is essentially made of the Cartesian product of the involved parameters. If there are MPI_ANY_SOURCE, the complexity becomes

\[ O\left(\sum_{j=0}^{k-1}\left(\sum_{i=0}^{n_j-1} t_i + a_j\right)\right) \]  

Both Eq. (1) and Eq. (2) are strictly equivalent and express the exact same complexity. It is possible to consider \(a_j\) as the \(t_i\) associated with the special rank MPI_ANY_SOURCE. In that case, MPI_ANY_SOURCE is considered as one of the \(n_j\) ranks; forcing Eq. (2) to degenerate to Eq. (1).

The worst possible search scenario for the linked list approach happens when the sought queue item is at the tail of the queue. It also occurs when the queue item does not exist in the queue; the queue in this case must be traversed entirely just to realize that the search is unfruitful. Deletion always happens at the found position and is therefore of \(O(1)\)

Insertion is \(O(1)\) as well because it always happens at the end of the queue.

**The Array-Based Queue**

As shown in Figure 5, 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 queue items associated with the same rank \(i\) in \(O(t_i)\). The UMQ and PRQ search complexity is thus

\[ O(k + t_i) \]  

In the data structure shown in Figure 5, we will assume for each rank slot that the left linked list is the PRQ and the right one is the UMQ.

Whenever an MPI_ANY_SOURCE receive is issued, all the UMQ queue items associated with all the ranks must be probed (Figure 6). The UMQ search complexity is thus

\[ O(k + \sum_{i=0}^{n_j-1} t_i) \]
As mentioned earlier, **MPI\_ANY\_SOURCE** queue items build a separate queue of length $a_j$ attached to the ContextIdLead. In that case, for any incoming message, the **MPI\_ANY\_SOURCE** sub-queue must be searched on top of the linked list associated with the rank borne by the message envelope (Figure 7). The PRQ search complexity thus becomes

$$O(k+t_i+a_j)$$ (6)

![Figure 7: Array-based PRQ search when at least one MPI\_ANY\_SOURCE receive is pending](image)

Deletion always happens at the position where a sought queue item is found. Deletion is therefore $O(1)$. Insertion, no matter where it happens, is $O(1)$ as well because new queue items are simply appended to mere linked lists as shown in Figure 8.

![Figure 8: Possible insertion points in the array-based structure](image)

### The 4-D Data Structure Queue

The search procedure in absence of **MPI\_ANY\_SOURCE** is depicted in Figure 9. The right ContextIdLead is found in $O(k)$. Then the rank is decomposed in four slices of dimensionSpan each. As a reminder, $dimensionSpan = 2^{\lceil \log_2(n_j) \rceil / 4}$; with $n_j$ being the size of the communicator associated with the contextId $j$. The right Plane is found in $O(2^{\lceil \log_2(n_j) \rceil / 4})$ (Figure 9-a). Then, the right X coordinate is found in $O(1)$ because it is an array index in the Plane (Figure 9-b). The Y coordinate is found in in $O(2^{\lceil \log_2(n_j) \rceil / 4})$ (Figure 9-c) and the segment is reached. For both the PRQ and the UMQ, the Segment
contains a maximum of dimensionSpan distinct ranks as well; with the rank i having t_i queue items (Figure 9-d). The Segment traversal therefore happens in \(O(\sum_{i=0}^{\log_2(n_j)/4} t_i)\). The overall search complexity for both the UMQ and the PRQ is thus

\[
O(k + 2^{\log_2(n_j)/4} + 1 + 2^{\log_2(n_j)/4} + \sum_{i=0}^{\log_2(n_j)/4} t_i), \text{ or simply}
\]

\[
O(k + \sum_{i=0}^{\log_2(n_j)/4} t_i)
\]

(7)

---

**Figure 9: **4-D structure queue search in absence of MPI_ANY_SOURCE

When a posted receive bears MPI_ANY_SOURCE, once the ContextIdLead is found, the UMQ of all existing Segments must be probed. This corresponds to searching all the Y and X coordinates of all the Planes. While the complexity of that UMQ search for MPI_ANY_SOURCE can be derived from Figure 9 as well, one can notice that searching all the UMQs of the data structure comes down to searching all the \(t_i\) UMQ queue items of all the \(n_j\) ranks in the communicator. Such a search simply has a complexity of
The UMQ search complexity when an MPI\_ANY\_SOURCE queue item is posted is thus $O(\sum_{i=0}^{n_j-1} t_{i})$. The UMQ search complexity when an MPI\_ANY\_SOURCE queue item is posted is thus $O(k+\sum_{i=0}^{n_j-1} t_{i})$, as in Eq. (5); just like in the case of the array-based data structure.

Finally, when at least one MPI\_ANY\_SOURCE queue item is pending, the PRQ search performs all the operations depicted in Figure 9 and then searches the existing MPI\_ANY\_SOURCE sub-queue in $O(a_j)$ (Figure 10). The PRQ search complexity in that case thus becomes

$$O(k+\sum_{i=0}^{\lceil \log_2(n_j)/4 \rceil} t_{i} + a_j)$$

(8)

Figure 10: 4-D structure PRQ search in presence of MPI\_ANY\_SOURCE

Deletion happens at the point where the queue item is found and is therefore $O(1)$. Insertion of a new queue item is $O(1)$ as well and can be explained as follows (Figure 11). MPI\_ANY\_SOURCE queue items are appended to the end of the MPI\_ANY\_SOURCE sub-queue. If the queue item has a specific rank instead, its insertion always happens at the point where the previous search failed. The previous infructuous search failed because the rank decomposition showed that either 1) the Plane is inexistent; 2) the Plane exists but the Segment is inexistent because the X or Y coordinates yielded by the decompositions are currently unused; or 3) both the Plane and the Segment exists but the queue item does not. In the first case, a new Plane and Segment are created and inserted before the queue items. In the second, the Segment is inserted and then the queue item is. In the third case, the queue item is appended right away to the right PRQ or UMQ of the Segment where the search failed. Each of the aforementioned operations is $O(1)$.

Figure 11: Inserting a new queue item in the 4-D queue

**Runtime Complexity Analysis Summary**

The runtime complexities are summarized in Table 1. A few observations stem from the complexities:
1. When there are more than a single contextId (i.e., $k>1$), the runtime search complexity of the linked list design has a cubic behaviour in terms of the used parameters (Eq. (1), Eq. (2)). In comparison, the array-based approach and the 4-D proposal remain entirely quadratic without regard to the presence of MPI_ANY_SOURCE (Eq. (4), Eq. (5), Eq. (6), Eq. (7), Eq. (8)).

2. 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 (Eq. (4)) for both the UMQ and the PRQ. In the 4-D approach, this parameter has a logarithmic behaviour (Eq. (7)), meaning that its impact on the cost grows more slowly at large scales.

3. When there are MPI_ANY_SOURCE, the PRQ search in the linked list must potentially scan all the MPI_ANY_SOURCE queue items of all the existing contextIds (Eq. (2)). In comparison, the PRQ search of the array (Eq. (6)) and the 4-D structure (Eq. (8)) grows only by the number of MPI_ANY_SOURCE queue items pending for the sole contextId of interest.

4. More generally, as shown by the parameter $k$ for searches, each new communicator adds a single pointer visitation to the array (Eq. (4), Eq. (5), Eq. (6)) and the 4-D structure (Eq. (5), Eq. (7), Eq. (8)) while it adds the quadratic term $\sum_{i=0}^{n_j-1} t_i$ or $\sum_{i=0}^{n_j-1} t_i + a_j$ to the linked list (Eq. (1), Eq. (2)).

Finally, none of the complexities is impacted by the use of MPI_ANY_TAG.

<table>
<thead>
<tr>
<th>Linked list</th>
<th>Linked list</th>
<th>Eq. (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRQ/UMQ Search Without MPI_ANY_SOURCE</td>
<td>$O(\sum_{i=0}^{k-1} \sum_{j=0}^{n_i-1} t_i)$</td>
<td></td>
</tr>
<tr>
<td>PRQ/UMQ Search with MPI_ANY_SOURCE</td>
<td>$O(\sum_{i=0}^{k-1} \sum_{j=0}^{n_i-1} (t_i + a_j))$</td>
<td>Eq. (2)</td>
</tr>
<tr>
<td>Insertion</td>
<td>$O(1)$</td>
<td>Eq. (3)</td>
</tr>
<tr>
<td>Deletion</td>
<td>$O(1)$</td>
<td>Eq. (3)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Array</th>
<th>Array</th>
<th>Eq. (4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRQ/UMQ Search Without MPI_ANY_SOURCE</td>
<td>$O(k + t_j)$</td>
<td></td>
</tr>
<tr>
<td>PRQ search with MPI_ANY_SOURCE</td>
<td>$O(k + \sum_{i=0}^{n_j-1} t_i)$</td>
<td>Eq. (5)</td>
</tr>
<tr>
<td>Insertion</td>
<td>$O(1)$</td>
<td>Eq. (3)</td>
</tr>
<tr>
<td>Deletion</td>
<td>$O(1)$</td>
<td>Eq. (3)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4-D</th>
<th>4-D</th>
<th>Eq. (7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRQ/UMQ Search Without MPI_ANY_SOURCE</td>
<td>$O(k + \sum_{i=0}^{\log_2(n_j)-1} t_j)$</td>
<td></td>
</tr>
<tr>
<td>UMQ search with MPI_ANY_SOURCE</td>
<td>$O(k + \sum_{i=0}^{n_j-1} t_i)$</td>
<td>Eq. (5)</td>
</tr>
<tr>
<td>PRQ search with MPI_ANY_SOURCE</td>
<td>$O(k + \sum_{i=0}^{\log_2(n_j)-1} (t_i + a_j))$</td>
<td>Eq. (8)</td>
</tr>
<tr>
<td>Insertion</td>
<td>$O(1)$</td>
<td>Eq. (3)</td>
</tr>
<tr>
<td>Deletion</td>
<td>$O(1)$</td>
<td>Eq. (3)</td>
</tr>
</tbody>
</table>

### 5.1.2 Intuitive Grasp of Runtime Costs

Asymptotic complexities do not always allow an intuitive grasp of comparative costs. We therefore propose to quickly pass the three approaches through a test performed in [3]. The original test, based on 4096 processes, searches for a queue item which is the last one in a queue of 4095 items. We use a variant of the test where each process has a single queue item in the queue. Assuming a single communicator, the number of pointer operations required for the search is shown in Table 2 for each of the message queue architectures.

In the linked list case the search would always scan through all the items. In the array-based case, the search always scans through three pointers; one for the ContextIdLead, one for the position of the rank by indexing the array, and the last one for the queue item being looked for. In the 4-D structure, the search scans through one pointer to find the ContextIdLead, dimensionSpan pointers to find the Plane,
one pointer to reach the X coordinate, \textit{dimensionSpan} pointers to find the Y coordinate and \textit{dimensionSpan} pointers to reach the queue item in the Segment. The goal of this analysis is to show intuitively how slowly the 4-D approach degrades compared to the linked list when scales grow. The array-based approach is communicator size-agnostic.

One can easily show that the same slow degradation is expected from the proposed approach even if each process (rank) has \( t \) pending messages. Table 3 shows the same test for 10 communicators, each having the same size. In this case, the number of pointer operations for both the array-based approach and the 4-D proposal grows exactly by the number of additional communicators. In comparison, the number of pointer operations for the linked list-based approach grows by the sum of all the items associated with each of these communicators.

<table>
<thead>
<tr>
<th>Communicator size</th>
<th>Linked list</th>
<th>Array</th>
<th>4-D</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>4095</td>
<td>3</td>
<td>26</td>
</tr>
<tr>
<td>65,536</td>
<td>65,535</td>
<td>3</td>
<td>50</td>
</tr>
<tr>
<td>1,048,576</td>
<td>1,048,575</td>
<td>3</td>
<td>98</td>
</tr>
</tbody>
</table>

Table 3: Number of pointer operations required to reach the last queue item for different communicator sizes (case of 10 communicators)

<table>
<thead>
<tr>
<th>Communicator size</th>
<th>Linked list</th>
<th>Array</th>
<th>4-D</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>( 9 \times 4095 + 4095 )</td>
<td>( 9 + 3 )</td>
<td>( 9 + 26 )</td>
</tr>
<tr>
<td>65,536</td>
<td>( 9 \times 65,535 + 65,535 )</td>
<td>( 9 + 3 )</td>
<td>( 9 + 50 )</td>
</tr>
<tr>
<td>1,048,576</td>
<td>( 9 \times 1,048,575 + 1,048,575 )</td>
<td>( 9 + 3 )</td>
<td>( 9 + 98 )</td>
</tr>
</tbody>
</table>

5.2 Memory Overhead Analysis

The memory overhead of the proposed 4-D approach is difficult to model concisely in analytic form because it increases by steps, according to \textit{dimensionSpan} and according to the distribution of the ranks in the queues. 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 to host the same number of queue items. 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. As a reminder, both the array and the 4-D are built over the rank; not the tag. As a result, the memory overhead behaviour is easier to compute only when each process has a single pending message; so as to avoid the same rank showing up more than once. In fact the same rank appearing twice does not increase the number of RankHeads, or Plane, X, Y coordinates or Segments. Consequently, when the same rank appears more than once, the computation has to account for it only once to be accurate.

We designate by \( n_{MQI} \) the number of MQIs. For a communicator of size \( S \), we proceed to compute the memory consumption \( M_{ll}, M_{ar} \) and \( M_{4D} \) of the linked list-based, array-based and 4-D designs respectively. The linked list design only uses a chains of MQIs; thus

\[ M_{ll} = n_{MQI} \times \text{sizeof}(MQI) \]  \hspace{1cm} (9)

The array design (See Figure 4 in Section 5) is made of the ContextIdLead object pointing to an array of \( S \) RankHead objects. Thus

\[ M_{ar} = \text{sizeof}(ContextIdLead) + S \times \text{sizeof}(RankHead) + n_{MQI} \times \text{sizeof}(MQI) \]  \hspace{1cm} (10)

The 4-D design (Figure 1 and Figure 2) is made of the ContextIdLead object, the number \( n_{seg} \) of Segment objects required to host all the MQIs, and the number \( n_{pl} \) of Plane objects required to host all the \( n_{seg} \) Segments. X and Y are not separate objects; they are tied to the Planes and Segments. Thus,
A Segment can contain up to `dimensionSpan` distinct MQIs; thus, for contiguously distributed ranks,

\[ n_{\text{seg}} = \left\lceil \frac{n_{\text{MQI}}}{\text{dimensionSpan}} \right\rceil \]  

Since the X dimension hosted in Planes is an array of `dimensionSpan` slots, Planes are polymorphic objects whose sizes depend on `dimensionSpan` (see Figure 1). We designate by Plane4, Plane8, Plane16, etc. the specific Plane object used for `dimensionSpan` equals 4, 8, 16, etc.

For the same `n_{\text{MQI}}`, the respective memory overheads \( O_{\text{ar}} \) and \( O_{4D} \) of the array and 4-D designs compared to the linked list approach are

\[ O_{\text{ar}} = M_{\text{ar}} - M_{\text{ll}} \]

\[ O_{4D} = M_{4D} - M_{\text{ll}} \]

We designate by Plane4, Plane8, Plane16, etc. the specific Plane object used for `dimensionSpan` equals 4, 8, 16, etc.

Table 4 shows the runtime size of the objects on a 64-bit Linux system. The ContextIdLead of the array approach is not the same C type as the ContextIdLead of 4-D design; though, they happen to have the exact same size. Table 5 shows the values of \( O_{\text{ar}} \) and \( O_{4D} \) for communicators of different sizes. For each size, the results are presented for a few different queue item numbers (\( n_{\text{MQI}} \)) in the queues.

<table>
<thead>
<tr>
<th>Communicator Size (S)</th>
<th>Number of Queue Items (n_{MQI})</th>
<th>Array Overhead (O_{ar})</th>
<th>4-D Overhead (O_{4D})</th>
</tr>
</thead>
<tbody>
<tr>
<td>4,096</td>
<td>1</td>
<td>128.05KB</td>
<td>184B</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>128.05KB</td>
<td>76B</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>128.05KB</td>
<td>6.07KB</td>
</tr>
<tr>
<td></td>
<td>4,096 (i.e., full)</td>
<td>128.05KB</td>
<td>24.68KB</td>
</tr>
<tr>
<td>65,536</td>
<td>1</td>
<td>2.00MB</td>
<td>248B</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>2.00MB</td>
<td>536B</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>2.00MB</td>
<td>3.15KB</td>
</tr>
<tr>
<td></td>
<td>65,536 (i.e., full)</td>
<td>2.00MB</td>
<td>194.3KB</td>
</tr>
<tr>
<td>1,048,576</td>
<td>1</td>
<td>32.00MB</td>
<td>376B</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>32.00MB</td>
<td>520B</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>32.00MB</td>
<td>1.82KB</td>
</tr>
<tr>
<td></td>
<td>1,048,576 (i.e., full)</td>
<td>32.00MB</td>
<td>1.51MB</td>
</tr>
</tbody>
</table>

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 MQIs in the queues. An example of such a situation occurs when there are only 1000 queue items 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 queue items. This behaviour can be observed at a fixed communicator size. Second, its increase rate tends to flatten considerably when the communicator size grows. The memory overhead growth rate is smaller for the communicator of size 65,536 compared to the communicator of size 4096. That growth is even smaller for the communicator of size 1,048,576. The two behaviours are very consistent with the scalable comportment we expect to provide. The case when all the queue items 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 5.19 times lighter on memory than the array-based design for a 4096-rank communicator. This ratio becomes 10.54 and 21.19 for 65,536 and 1,048,576 communicator sizes respectively.

We must point out that the previous test assumed contiguous ranks. The 4-D mechanism does not behave similarly memory consumption-wise when the ranks are sparsely distributed. In particular, two queue items whose ranks are separated by at least dimensionSpan are located in two distinct Planes; meaning that the maximum number of Planes of the container can be required for no more than dimensionSpan distinct ranks. Similarly, two ranks separated by at least dimensionSpan will require two distinct Segments. The maximum memory overhead is reached with dimensionSpan ranks each separated by dimensionSpan. When that happens, all the Segments are allocated in the 4-D container; this means implicitly that all the Planes are allocated as well to host those Segments. However, once that maximum is reached, the overhead totally levels up and stops increasing when the other (dimensionSpan-1)* dimensionSpan additional distinct ranks are added. For instance, with a communicator of size 1,048,576, the memory overhead of the 4-D container reaches 1.51MB for no more than $32^3 = 32,768$ distinct ranks if these ranks are distributed in the fashion 0, 32, 64, 96, ... , 1,048,544. However, once that maximum is reached, the memory overhead stops increasing for all the other $31*32^3 = 1,015,808$ distinct ranks.

6 Practical Justification of a New Message Queue Data Structure

We first remind that hash tables have not been envisioned because they have been proposed before [21][22] and deemed prohibitively slow for insertions [18]. There are several other existing data structures that allow fast searches. In particular, the large family of tree-based data structures were initially considered as candidates. We did a quite thorough analysis of several of them before considering a totally new container. Non self-balancing trees suffer severe degradations and are not considered in our analyses. While self-balancing trees offer very good performances, they all render insertion and deletion less trivial, for the sake of the balancing that yield the good search performances.

For the sake of justification, we compare how red-black trees perform compared to our 4-D data structure. The choice of red-black trees resides in their well-known performance that justifies their tremendous popularity. As an example, they are heavily used in the Linux kernel for schedulers, the high-resolution timer, the ext3 filesystem and the virtual memory areas tracking [35]; just to name a few examples. We implemented both the 4-D and the red-black tree-based message queues offline (i.e. not inside MPI). Then we tested both designs by reproducing the message queue activity patterns found in MPI (See Section 3.4). We did not reproduce pattern 2 because it is similar to pattern 1; and we did not reproduce pattern 4 due to its similarity with pattern 3. No MPI_ANY_SOURCE is used because it is not a differentiator between the two designs. In order to encompass all the situations where an approach might have a distinctive advantage over the other one, we generated the following rank insertion and deletion patterns:

- Insert in increasing rank order; remove in increasing rank order
- Insert in increasing rank order; remove in decreasing rank order
• Insert in decreasing rank order; remove in decreasing rank order
• Insert in decreasing rank order; remove in increasing rank order
• Insert iteratively from both ends and remove in the same order. For instance, for 256 ranks, we insert 0, 255, 1, 254, 2, 253, ..., 127, 128; and we remove in the same sequence.
• Insert iteratively from both ends and remove in the reverse order. For instance, for 256 ranks, we insert 0, 255, 1, 254, 2, 253…, 127,128; and we remove in the sequence 128, 127, 129, 126, …, 255, 0.

We performed our tests on a node having two quad-core 2GHz AMD Opteron 2350 processors and 8GB of RAM. The binaries are created with GNU GCC 4.4.4 and run on an x86_64 Linux kernel 2.6.31. We realized that for both designs, the performance is absolutely not impacted by the data insertion and deletion sequence. Consequently, we report only the outcome of the first sequence in Table 6. We generated a different binary for memory tests to avoid instrumentation impacts on the latency results. Each test is averaged over 3 iterations. For each latency test, we perform n insertions, followed by n deletions or searches depending on the pattern; with n being the number of ranks. The tests are performed for one and five messages per rank to simulate the cases of single and multiple tags respectively. All the tests are performed for a single contextId because contextIds are not a differentiator between the two approaches. In fact, contextIds are searched before any of the queues (4-D or tree) is reached. We can notice that the 4-D design always has an edge over the tree design. More importantly, Table 7 shows that the 4-D design is far more memory-efficient than the tree. The memory overhead of the 4-D is computed runtime with Eq. (15). The memory overhead of the tree is \( O_{\text{rb}} = \text{sizeof}(\text{empty\_tree\_object}) + n_{\text{MQI}} \times \text{sizeof(tree\_node)} \); with \( n_{\text{MQI}} \) being the number of MQIs in the queue. One can refer to Section 5.2 for the reasoning that would lead to \( O_{\text{rb}} \). For reasons already mentioned in Section 5.2 as well, the memory overhead tests are performed for a single message per rank. Trees are better than arrays because they allocate memory only on demand; meaning that they would not waste large amounts of fixed memory for large communicators which are sparsely used with respect to message queue. However, they do consume an amount of memory proportional to the number of distinct ranks in the queue. Furthermore, no matter the tree design, simply for the existence of left, right and possibly parent pointers, tree nodes are more expensive memory-wise than each RankHead object used for the array. Consequently, when all the ranks are represented in the message queue, trees, without regards to their particular design are even more expensive than the array-based approach.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>5 msgs</th>
<th></th>
<th>5 msgs</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 msg</td>
<td></td>
<td>1 msg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>256 ranks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Patern 1</td>
<td>4.95E-04</td>
<td>1.28E-02</td>
<td>2.78E-02</td>
<td>4.00E-02</td>
<td></td>
</tr>
<tr>
<td>Patern 3</td>
<td>5.15E-04</td>
<td>1.20E-02</td>
<td>2.51E-02</td>
<td>3.85E-02</td>
<td></td>
</tr>
<tr>
<td>Patern 5</td>
<td>5.39E-04</td>
<td>1.22E-02</td>
<td>1.45E-03</td>
<td>1.33E-02</td>
<td></td>
</tr>
<tr>
<td>16,384 ranks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Patern 1</td>
<td>1.64E+00</td>
<td>8.71E+01</td>
<td>4.85E+00</td>
<td>9.05E+01</td>
<td></td>
</tr>
<tr>
<td>Patern 3</td>
<td>1.62E+00</td>
<td>8.73E+01</td>
<td>4.70E+00</td>
<td>9.05E+01</td>
<td></td>
</tr>
<tr>
<td>Patern 5</td>
<td>1.62E+00</td>
<td>8.70E+01</td>
<td>3.07E+00</td>
<td>8.90E+01</td>
<td></td>
</tr>
<tr>
<td>131,072 ranks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Patern 1</td>
<td>2.33E+02</td>
<td>5.99E+03</td>
<td>4.62E+02</td>
<td>6.21E+03</td>
<td></td>
</tr>
<tr>
<td>Patern 3</td>
<td>2.33E+02</td>
<td>5.98E+03</td>
<td>3.40E+02</td>
<td>6.09E+03</td>
<td></td>
</tr>
<tr>
<td>Patern 5</td>
<td>2.33E+02</td>
<td>5.99E+03</td>
<td>4.66E+02</td>
<td>6.22E+03</td>
<td></td>
</tr>
</tbody>
</table>

It is important to mention that we did make a design choice for the tree design. Each rank is represented with a tree node; yielding the fastest possible tree-based representation of the message queue. We could have decomposed the rank; and used only a certain number of bits as the tree key. The other ignored bits
would lead to linear traversals; as we do with Segment objects in the 4-D design. This design would have resulted in less memory consumption but would have slowed down the tree. The usual strength of binary search trees resides in their speed; and we wanted to compare the 4-D design against that strength at its best. In any case, we claim the superiority of the 4-D design in terms of memory consumption degradation even if only part of the rank bits is used as tree key.

Table 7: Memory overhead (in KB) of 4-D and red-black tree compared to linked list (1 message per rank)

<table>
<thead>
<tr>
<th></th>
<th>4-D</th>
<th>Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Peak</td>
<td>Average</td>
</tr>
<tr>
<td>256 ranks</td>
<td>3.242188</td>
<td>1.695312</td>
</tr>
<tr>
<td>16,384 ranks</td>
<td>48.61719</td>
<td>24.42969</td>
</tr>
<tr>
<td>131,072 ranks</td>
<td>193.117188</td>
<td>96.742188</td>
</tr>
</tbody>
</table>

7 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. Therefore, the cluster has a total of 64*11 = 704 CPU cores and 11*128GB = 1.375TB of RAM. The CPUs have 48KB of L1 cache per core and 16MB L3 shared cache. The nodes are equipped with Mellanox ConnexxtX-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 over OFED 1.5.3.3 [36]. In all the tests, the linked list design, which is the default MVAPICH2 implementation, is used as the reference for speedup, memory overhead and cache performance results.

7.1 Note on Measurements

The latency results report how long MPI spends processing message queue operations as a whole. We take a start_time_stamp right at the entrance of every topmost message queue routine, and we take an end_time_stamp right before the exit. This approach guarantees that every latency and overhead inherent to all the operations of each message queue design is taken into account. Thus, the latency of a message queue design for a process is the sum of all (end_time_stamp - start_time_stamp).

In [11], we computed the memory overhead via a difference between the working-set sizes. That approach was non-intrusive but is a bit noise-sensitive because some other aspects of the application can increase or decrease the virtual memory size of each process. In the current paper, we measure the memory consumption of the actual queue operations inside each process for each queue architecture. The MVAPICH2 code implementation of each queue design has been instrumented for that purpose. Each allocation of objects such as RankHead, ContextIdLead, Plane, Segments and MQI updates the average and maximum memory consumption associated with message queue activities. For each message queue design, we use a separate MVAPICH2 build for memory tests; in order to avoid any instrumentation-induced impact on the latency results.

Finally, we gather cache statistics with the Linux “perf stat” utility. The results are gathered for the whole execution of each process. The cache miss improvement for the array for instance is computed as (cache_miss_linked_list - cache_miss_array)*100/cache_miss_linked_list. The same formula is used for the 4-D design. A 75% cache miss ratio improvement for the array for instance means that cache_miss_array is only 25% of cache_miss_linked_list for the same execution.
7.2 Microbenchmark Results

The microbenchmark 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. All the microbenchmark metrics are gathered over the receiver process only; that is, the one that builds the long queues.

7.2.1 Single Communicator Tests

We first perform the tests with a single active communicator (i.e., one contextId) in the job. Figure 12 shows the queue operation time speedup yielded by the array-based and our 4-D approaches for both the PRQ and the UMQ. When the queue items are consumed from the bottom of the queue (reverse search) as in the test described in [3], the speedups are always above 1. For 10 pending messages per sender, the PRQ search is accelerated by up to more than 40 times by the array-based design and more than 30 times by our 4-D design (Figure 12-a). The UMQ is accelerated even more by both the array and the 4-D design (Figure 12-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 (Figure 12-d and Figure 12-h). 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 might not be compensated for.

Those two forward and reverse search tests are the extremes. There are two kinds of in-between situations. The first one, which is not presented because it behaves like certain cases of the second one, is when a fraction of the matches occurs toward the bottom and another fraction occurs towards the top. The second situation occurs when a fraction of the receives are MPI_ANY_SOURCE (Figure 12-b and Figure 12-c for PRQ; Figure 12-f and Figure 12-g 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 queue items in Figure 12. 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 reaches 100%; in which case the speedups disappear. Forward search results with fractions of MPI_ANY_SOURCE are omitted because they add little information to the set of tests already presented in Figure 12. All forward searches tend to behave like Figure 12-d for the PRQ and Figure 12-h for the UMQ.

Figure 13 shows the average memory overhead generated by each of the array and 4-D approaches compared to the linked list. We omit the peak memory tests because they present the same trends as in Figure 13. We can observe in Figure 13 that the 4-D approach is always the one that adds the smaller memory overhead to the linked list memory requirements. Even in the worst case, the 4-D approach is approximately as twice more memory efficient than the array design. We expect that ratio to keep improving with job sizes.
Figure 12: Queue operation speedup over the linked list, with 1 communicator.
We gathered cache miss improvements ratios as well. However, for a single communicator, no specific trend is observed. The linked list behaves better or worse than the array and the 4-D approaches.
in a totally random fashion. Based on the cache results that we show later for the multiple communicator tests, we can conclude that a clear trend builds up after a threshold that the single communicator tests has not reached in our microbenchmarks. In theory, every other thing kept constant, the cache behaviour is strongly impacted by the search depths. Consequently, even for a single communicator, we would expect a clear trend for job sizes larger than the one used in our tests.

7.2.2 Multiple Communicator Tests

We repeated the tests for 10 and 50 active communicators. The behaviour of the message queues in presence of MPI_ANY_SOURCE can be generalized from the single communicator tests. As a consequence, for conciseness sake, we do not repeat the tests with various percentages of MPI_ANY_SOURCE. Then, in the complete absence of MPI_ANY_SOURCE, both PRQ and UMQ behave very similarly. We therefore stick to only one of these queues, namely the PRQ. All the observations made for the PRQ with 10 and 50 communicators in absence of MPI_ANY_SOURCE can be generalized to the UMQ for the same number of communicators. All the tests are done for 5 pending messages per process. For each of the tests, we provide the results for forward and reverse traversal of the communicators. The forward traversal uses the same communicator traversal order for both senders and receivers. The reverse traversal does the opposite. These communicator traversal orders are different from the forward and reverse searches that happen inside each communicator. Figure 14 depicts the various matching sequences yielded by the combinations of those two distinct order parameters. As shown in Figure 14, the matches happen entirely for one rank before moving on to the next one.

We first notice that the speedup increases with the number of active communicators (Figure 15). The higher the number of communicators, the larger the speedups. The PRQ reverse search speedup with one communicator (Figure 12) varies between 1.7 and 40. With 10 communicators, it varies between 42 and 175.8 (Figure 15-a). With 50 communicators, it varies between 100 and 556 (Figure 15-c). The huge speedups observed for multiple communicator tests can be explained as follows. The array and 4-D approaches will always skip all the queue items in any communicator that is not of interest for the search in progress. In comparison, the linked list approach will always scan the queue linearly and go over all the queue items of the irrelevant communicators if the communicator of interest happens to occur later in the queue. As shown in Figure 14, the linked list will always begin with the queue item (1,0) of communicator_0, go all the way to the end of that communicator and then jumps on queue item (1,0) of communicator_1 and then progress the same way until the match occurs. The magnitude of the speedups observed in these tests is an embodiment of the observation number 4 at the end of Section 5.1.1. The same information is conveyed by Table 3 in Section 5.1.2.

The average memory overhead tests (Figure 16) show the expected trend of the array being more memory-greedy than the 4-D approach. Once again, we omit the peak memory overhead tests because they present the exact same trend as the ones shown in Figure 16.

We notice huge cache miss ratio improvements (Figure 17, Figure 18, and Figure 19). As expected, the L1 data cache miss improvement is higher with 50 communicators (Figure 17-c, Figure 17-d) than with 10 communicators (Figure 17-a, Figure 17-b). For the L1 instruction cache miss improvements, the tests with 10 communicators (Figure 18-a, Figure 18-b) show a slight absence of consistent trend even though the general observation is an improvement. With 50 communicators (Figure 18-c, Figure 18-d), the trend for L1 instruction cache miss improvement is clear and the improvement ratios are higher as well. Finally the last level cache ratios (Figure 19) follow the same trend of increase from 10 to 50 communicators. The tests in Figure 19-c and Figure 19-d correspond to high rates of back-and-forth
between the L3 cache and physical memory for the linked list-based queue; a very detrimental situation that both the array and the 4-D approaches fix.

Figure 14: Different match orders for multi-communicator tests: example with 10 communicators of 256 distinct ranks and 5 pending messages per process
Figure 15: PRQ operation speedup over the linked list, with multiple communicators

Figure 16: PRQ average memory overhead compared to the linked list, with multiple communicators
Figure 17: PRQ L1 data cache improvement over the linked list, with multiple communicators

Figure 18: PRQ L1 instruction cache improvement over the linked list, with multiple communicators
7.3 Application Results

The tested applications are Radix [37] and Nbody [37] [38]. 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 have very shallow search depths and present no room for search optimization. We experimented with NAS LU [39] and AMG2006 [40] as they build reasonably long queues. Unfortunately, AMG2006 over 512 processes has average search depths of 4.29 and 3.56 for UMQ and PRQ, respectively. These metrics are 4.04 and 0.67 for NAS LU class E over 512 processes.

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 8. 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 8 shows the job sizes appended to the names of Nbody. We provide in Table 8 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 8 are the averages computed over three 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 Search Length (MASL) column shows the largest of the average search lengths 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 single communicator.
reverse search scenario presented in the microbenchmarks (Section 7.2); and the smaller they are, the closer it gets to the forward search scenario. Table 8 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.

Table 8: Queue behaviour data for Radix and Nbody

<table>
<thead>
<tr>
<th>Application and parameters</th>
<th>PRQ</th>
<th>UMO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Max Queue Length (MQL)</td>
<td>Average Queue Length (AQL)</td>
</tr>
<tr>
<td>Radix.n28</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Radix.n29</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Radix.n30</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Nbody.256</td>
<td>255</td>
<td>254.953</td>
</tr>
<tr>
<td>Nbody.512</td>
<td>511</td>
<td>508.775</td>
</tr>
<tr>
<td>Nbody.704</td>
<td>703</td>
<td>682.646</td>
</tr>
</tbody>
</table>

Application results are gathered per process; then averaged over all the processes of the job. For instance, peak memory represents the average of the peak memory outputted by each individual process; and average memory is averaged over the average memory outputted by each process. Figure 20 and Figure 21 present the performance results for Radix and Nbody respectively. For Radix, the queue operation speedup reaches 8.46 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 (Figure 20-b, Figure 20-c). For 2\(^{30}\) integers in particular, the array-based and the 4-D approaches tremendously improve the overall execution time of Radix. 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 (Figure 21), the 4-D approach accelerates the queue processing time by close 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. Figure 21-b and Figure 21-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 8.
Figure 20: Radix speedups, memory overhead and cache statistics for the array-based and 4-D designs over the linked-list approach.
As for the memory overheads, there is a clear trend that the array-based approach is always more memory-intensive than the 4-D design (Figure 20-d, Figure 20-e and Figure 21-d, Figure 21-e). The memory overheads, both average and peak values, are very consistent with the expectations. For Nbody
(Figure 21-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 (Figure 20-d), the 4-D approach shows a decreasing average memory overhead when the number of sorted integers grows. There is 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 (Figure 20-d) when executed with the 4-D approach. The array approach can yield the negative memory overhead to a lesser extent than the 4-D approach because of its fixed allocation scheme that depends on the communicator size. We must emphasize a key difference between the applications and the microbenchmarks for the memory behaviours. With the microbenchmarks, the queues of the linked list, the array and the 4-D container all have the same number of items at each step of the execution. Actually, the microbenchmarks do just back-to-back communications for the purpose of message queue processing. The applications on the other hand mix computations and communications in a totally uncontrolled fashion as far as message queue operations are concerned. In particular, the three kinds of message queue do not necessarily have the same number of queue items at the same steps of the application execution. The negative memory overhead and the difference of trends between the peak and average memory overheads are a consequence of the uncontrolled environment shown by the applications. The observation being made and explained here is that the memory behaviour of the 4-D approach is more interesting in non-controlled environments; the very environment that characterize applications.

Finally, Radix generally improves a lot on the L1 data cache misses. The instruction cache miss rate improvement is substantial as well; especially when sorting $2^{30}$ values. As expected, the Last level gets noticeably improved for radix.30. The comparative cache miss variations are not pronounced for Nbody. There is generally a slight improvement for L1 data cache and a generally slight degradation of less than 0.8% in the worst case for the last level cache. These small improvement and degradations are due to the relatively shallow search depth shown by Nbody. The L1 instruction cache for Nbody, however, shows serious degradations for both the array and the 4-D containers. Nbody is known to be very computation-intensive [38]. As a result, its instruction cache is usually filled with non-message queue-related code. The linked list-based message queue instructions are simpler and less diversified than the array and 4-D-based queue operation instructions. Consequently, when the queue operation instructions must get room in the L1 instruction cache or leave it for computation instructions to occupy the space, less eviction happens with the linked list than the other two message queue types.

### 7.3.1 Message Queue Memory behaviours at Extreme Scales

While the speed differences between the 4-D and the linked list-based queues are obvious even at 512 and 704 processes, the memory values obtained convince only by their trends (See Table 9 for the average memory values). In fact, the cluster that we used for these tests has 2GB of memory per CPU core; and the array average memory overhead for Nbody at 704 processes represent only 0.0008% of the memory available for the process.

However, the scales targeted by this work go far beyond 704 CPU cores. Let's re-examine the percentage of process memory consumed by message queues on nowadays top supercomputers. Radix was run at the fixed size of 512 processes and its results are difficult to extrapolate with larger system sizes. Nbody was run over increasing job sizes and is therefore a good candidate for extrapolation. As of November 2012, each of the 1,572,864 CPU cores of Blue Gene Sequoia has no more than 1GB of RAM available while each of the 560,640 CPU cores of the Titan - Cray XK7 has less than 1.27GB [2]. Let's
also consider a large HPC application $A_H$ which reuses about 2 libraries built out of MPI. That application would have a total of 3 job-size communicators [34], leading to a total of 6 job-size contextIds in an MPI implementation such as MVAPICH or MPICH where the same communicator has two contextIds [41]; one for collectives and another for non-collective communications. We are interested in seeing the impact of running Nbody or the hypothetical job $A_H$ on the Titan - Cray XK7 and the Blue Gene Sequoia.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>App</td>
<td>Linked list</td>
<td>Array</td>
<td>4-D</td>
<td>Overhead</td>
<td>Overhead 4-D</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>radix.28</td>
<td>97.15702733</td>
<td>111.6615767</td>
<td>97.51474067</td>
<td>14.50454933</td>
<td>0.357713333</td>
</tr>
<tr>
<td>2</td>
<td>radix.29</td>
<td>108.466118</td>
<td>114.992498</td>
<td>107.435325</td>
<td>6.52638</td>
<td>-1.030793</td>
</tr>
<tr>
<td>3</td>
<td>radix.30</td>
<td>141.070079</td>
<td>138.628713</td>
<td>137.0056497</td>
<td>-2.441366</td>
<td>-3.814858</td>
</tr>
<tr>
<td>4</td>
<td>nbody.256</td>
<td>79.712919</td>
<td>86.06264833</td>
<td>81.053974</td>
<td>6.349729333</td>
<td>1.341055</td>
</tr>
<tr>
<td>5</td>
<td>nbody.512</td>
<td>82.11410633</td>
<td>93.55728967</td>
<td>82.87720267</td>
<td>11.44318333</td>
<td>0.763096333</td>
</tr>
<tr>
<td>6</td>
<td>nbody.704</td>
<td>107.045541</td>
<td>123.9089133</td>
<td>108.5719277</td>
<td>16.86337233</td>
<td>1.526386667</td>
</tr>
</tbody>
</table>

We show in Table 10 the absolute message queue related memory consumption and the percentage of the physical memory represented by that consumption for each process. Our 704 CPU cores test system is also represented in Table 10 for comparison purposes. Anticipating the same linear growth observed for the average memory overhead of Nbody for the array (Table 9), we extrapolated that same amount of overhead to Titan and Sequoia. For the 4-D, the extrapolation is a more difficult task because the memory overhead growth is not linear (both in theory and experimentally). For instance, the 4-D queue consumes less memory for 1,048576+1 than for 1,048,576 because of the change in dimensionSpan (See Table 5 for more such examples). Nevertheless, we show for the application $A_H$ what would happen. The results shown for $A_H$ are computed from the data in Table 4, along with Eq. (14) and Eq. (15). For Nbody, the **fixed** memory overhead of the array would represent more than 3% of the available RAM for each process on Sequoia. That overhead exists even when the message queues host only 1 MQI. However, the process must fit, in that 1GB, its binary and all the other MPI internal data; on top of hosting its useful application data; without hopefully swapping. It helps to mention that MPI itself already needs a substantial amount of system buffers to function. Those buffers are crucial for the unavoidable eager and control messages. The requirement of these buffers is even higher for RDMA-enabled network technologies such as InfiniBand and iWARP. For those particular network technologies, the lesser system buffer MPI can access, the tougher the flow control mechanism it must use to recycle them. The effects of this flow control mechanism are wait and delay propagation. It also helps to realize that the fear of memory consumption at large scale in MPI is not necessarily viewed from the point of view of a single MPI aspect such as message queues; it is the aggregated effect of many aspects that is worrisome. The way of fixing the effect of MPI eating away 30% of the process memory is to fix separately each of the 5, 10 or 20 aspects responsible for mere 3, 4 or 5%. This approach, already being fulfilled in the community, justifies ideas like replacing comprehensive enumeration of process ranks by ranges to save on memory consumed by, not array-based message queue RankHeads, but mere 8-byte integers [30]. All those considerations show that the 28% fixed memory overhead of the array approach for $A_H$, compared to less than 1% for the 4-D, would be unsustainable for Sequoia, let alone for future supercomputers. We emphasize that the description of $A_H$ is completely realistic. Actually, we even omitted in its definition the many virtual topology communicators that might be required in large networks; worsening even further
than described in Table 10 the impact of array-based message queues on memory per CPU core at large scales.

<table>
<thead>
<tr>
<th>Application $A_H$</th>
<th>Nbody</th>
<th>4-D</th>
<th>Array</th>
<th>4-D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead in GB (% of RAM per process)</td>
<td>Overhead in GB (% of RAM per process)</td>
<td>Overhead in GB (% of RAM per process)</td>
<td>Overhead in GB (% of RAM per process)</td>
<td></td>
</tr>
<tr>
<td>Current</td>
<td>1.608E-05(8.04E-4%)</td>
<td>1.456E-6(7.278E-5%)</td>
<td>1.26E-4(6.30E-3%)</td>
<td>2.5E-5(1.20E-3%)</td>
</tr>
<tr>
<td>Titan</td>
<td>0.0128(1.01%)</td>
<td>-</td>
<td>0.100251(7.89%)</td>
<td>4.727E-3(0.37%)</td>
</tr>
<tr>
<td>Sequoia</td>
<td>0.036(3.59%)</td>
<td>-</td>
<td>0.281250(28.10%)</td>
<td>6.610E-3(0.66%)</td>
</tr>
</tbody>
</table>

8 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.

From a speed of operation point of view only, 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. As physical memory is a shrinking resource per CPU core, the array-based approach is a solution that will bear another scalability issue at large scales because it will waste a lot of memory for large communicators. In this work, we propose a novel message queue mechanism that is scalable with respect to both speed and memory consumption. After separating the queue items by communicator identifiers, our design decomposes the rank into four slices and use the three most significant slices to build a 3-dimensional jump coordinate that is used to substantially accelerate searches in situations of large queues. A fourth dimension made of the least significant slice is exploited to build small-length PRQs and UMQs. Our 4-dimensional design not only limits considerably linear traversals but it optimizes unfruitful searches by detecting them early in the 3-dimensional jump process. We justify the need for a novel approach by comparing it against well-known data structures such as binary search trees which prove slower and substantially more memory-greedy.

We implemented both our 4-dimensional design and the array-based approach and compared them with the linked list-based message queue. Both our design and the array approach considerably improve over the linked list-based message queue. We show speedup, memory overhead and cache miss improvement tests. As expected, the array-based approach shows the higher speedups but it requires more memory than our 4-dimensional design. We tested Radix and Nbody and observed up to 4 time execution time improvement for our approach. While the array-based method improves even better, it has an order of magnitude more memory overhead than our 4-dimensional design. With Nbody in particular, the array-based approach consumes more memory when the job size increases while our proposed design shows a generally stable memory overhead. We show in particular that the fixed allocation scheme of the array-based message queue becomes unsustainable at petascale because it consumes an unacceptably large percentage of the physical memory available to each process. With Nbody as well, the large memory allocation scheme of the array-based approach proves detrimental even for the speedup. This behaviour makes the speedup of the array-based approach worse than our 4-dimensional mechanism for small queue
search depths. Compared to the linked list, we also show that both the array and the 4-dimensional container can substantially improve cache miss ratio when the search depths are substantial. The tests show in particular that, for sufficiently large search depth, our design is on par with the array approach when it comes to cache miss rate improvement.

For future work, we first intend to observe the behaviour of our proposed approach on larger-scale systems. We also intend to investigate message queue dedication for certain communication types between intensively communicating peers. The dedication approach is meant to specifically target certain communications which impact the most the overall execution time due to their frequencies or data sizes.

9 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.

10 References

[31] MVAPICH/MVAPICH2, http://mvapich.cse.ohio-state.edu, last accessed 2013/01/17


Judicael Zounmevo is a High Performance Computing Ph.D. candidate in the Electrical and Computer Engineering department of Queen’s University in Kingston, ON, Canada. He is particularly interested in optimizing the MPI one-sided communication model, the MPI progress-engine in general and its message queues for the petascale and upcomingexascale systems. He is supervised by Dr. Ahmad Afsahi. Judicael has a thesis-based Master of Engineering from McGill University (Montreal, QC, Canada) and a Bachelor of Engineering from Ecole Polytechnique of Montreal, QC, Canada

Dr. Ahmad Afsahi is currently an Associate Professor in the Department of Electrical and Computer Engineering at Queen’s University in Kingston, ON, Canada. He received his Ph.D. degree in Electrical Engineering from the University of Victoria, BC, in 2000. His research activities are in the areas of parallel and distributed processing, network-based high-performance computing, high-speed networks, and power-aware high-performance computing. He has been mainly focused on improving the performance and scalability of communication subsystems, messaging layers and runtime systems for high-performance computing and data center clusters. His research has earned him a Canada Foundation for Innovation Award and an Ontario Innovation Trust Award. Dr. Afsahi is a Senior Member of IEEE.