Received: 10 October 2017 | Revised: 22 August 2018 | Accepted: 25 October 2018
DOI: 10.1002/cpe.5158

WILEY

SPECIAL ISSUE PAPER

Tail queues: A multi-threaded matching architecture

Matthew G.F. Dosanjh[1,2] | Ryan E. Grant[1,2] | Whit Schonbein[1,2] | Patrick G. Bridges[2]

[1]Center for Computing Research, Sandia National Laboratories,* Albuquerque, New Mexico
[2]Department of Computer Science, University of New Mexico, Albuquerque, New Mexico

Correspondence
Matthew G.F. Dosanjh, Center for Computing Research, Sandia National Laboratories, Albuquerque, NM; or Department of Computer Science, University of New Mexico, Albuquerque, NM.
Email: mdosanj@sandia.gov

Summary

As we approach exascale, computational parallelism will have to drastically increase in order to meet throughput targets. Many-core architectures have exacerbated this problem by trading reduced clock speeds, core complexity, and computation throughput for increasing parallelism. This presents two major challenges for communication libraries such as MPI: the library must leverage the performance advantages of thread level parallelism and avoid the scalability problems associated with increasing the number of processes to that scale. Hybrid programming models, such as MPI+X, have been proposed to address these challenges. MPI THREAD MULTIPLE is MPI's thread safe mode. While there has been work to optimize it, it largely remains non-performant in most implementations. While current applications avoid MPI multithreading due to performance concerns, it is expected to be utilized in future applications. One of the major synchronous data structures required by MPI is the matching engine. In this paper, we present a parallel matching algorithm that can improve MPI matching for multithreaded applications. We then perform a feasibility study to demonstrate the performance benefit of the technique.

KEYWORDS

high performance computing, many core, MPI, networks

1 | INTRODUCTION

Exascale class supercomputers will require orders of magnitude increases in parallelism. This requirement can be satisfied in two ways; increasing inter-node parallelism through higher node counts and increasing intra-node parallelism through core and hardware thread counts. When running classic MPI applications that utilize the MPI-everywhere programming paradigm, this increased parallelism increases rank space (the range of valid MPI addresses). This has the potential to expose a number of scalability issues in MPI initialization, MPI memory and cache usage, and MPI collective behavior.

Many-core systems can exacerbate these problems, not only by increasing process count but also by increasing the usage of shared intra-node resources such as RAM and High Bandwidth Memory (HBM). Because they feature a reduced clock speed and complexity, applications have to increase processes count to achieve the same computational throughput. For instance, the Intel Xeon Phi Knights Landing processors used on Los Alamos National Laboratories' Trinity[1] and NERSC's Cori[2] have 68 cores with 4 thread contexts each.

With these advancements in processor technology, careful adaptation of system software libraries is required to fully leverage the available performance. MPI message rate, for instance, has been shown to decrease by an order of magnitude when running on a many-core system.[3] One of the big factors in this performance degradation is matching MPI's data placement mechanism to identify incoming data with the target memory. Matching, in part, due to the ordering and wild-card constraints, creates a large critical section with two data-structures, with non-trivial dependencies. Most matching implementations use linked list data structures, which is problematic for many-core architectures, because the performance of these structures relies on pointer lookups and many-core architectures have reduced cache sizes and limited out-of-order processing.

Hybrid parallel programming models, such as MPI+X (MPI used in combination with a threading library such as OpenMP), have been proposed and utilized to address these concerns. These programming models reduce the number of MPI processes while maintaining the same level of

*Sandia National Laboratories is a multimission laboratory managed and operated by the National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

intra-node parallelism. This has the effect of reducing memory and cache requirements, reducing the number of messages required for collectives, and allowing multiple processing elements to exchange data using their shared memory space. Additionally, these models can take advantage of thread-level parallelism features that are unavailable to MPI-everywhere applications. For instance, most MPI-everywhere applications (where MPI acts as the sole layer of parallelism) often do not utilize hardware thread contexts to avoid resource contention during highly synchronous code paths, such as communication phases.

MPI's thread safe mode, MPI THREAD MULTIPLE,[4] is currently supported in MPI libraries, but often faces performance issues. Synchronous data structures, such as the matching engine[*] and serialized network hardware access, make MPI difficult to parallelize while maintaining performance. This is particularly difficult for the matching engine due to the ordering and wildcard constraints. In this paper, we explore a technique to improve performance in a thread safe matching engine. Previous work has explored levering multiple threads to perform a single matching request[5] by relaxing the matching constraints. Our approach allows multiple matching requests to progress concurrently while supporting ordering and wildcards.

In this paper, we present Tail Queues, our proposed algorithm, to reduce the critical section in matching. This technique allows for more performance in hybrid applications that utilize fine-grained messaging patterns. Tail Queues aims to separate matching into multiple critical sections by creating inboxes that isolate the potential race conditions of this data structure. This allows the lists to be parallelized independently. Most current hybrid applications avoid using MPI THREAD MULTIPLE by limiting multithreading to computations phases in Bulk Synchronous Parallel (BSP) applications. This is done as MPI THREAD MULTIPLE has been non-performant when compared to single-threaded implementations.

There are three major contributions of this paper:

- a novel parallel matching architecture;
- a proof-of-concept study of the parallel matching architecture; and
- a feasibility study using a prototype implementation in MPICH.[6]

The rest of this paper is structured as follows. Section 2 presents the background of MPI, Matching, and Thread Multiple. Section 3 describes the Tail Queues architecture and how it could be applied to different matching algorithms. Section 4 presents a study of a proof of concept benchmark and the preliminary Tail Queues implementation. Section 5 compares our architecture with the state-of-the-art in matching and MPI Thread Multiple techniques. Section 6 presents our conclusions and proposes future work.

2 | BACKGROUND

In this section, we present the background of this work. Section 2.1 discusses the details of matching. Section 2.2 presents the different threading modes of MPI. Section 2.3 presents the proposed application paradigms that have been proposed to better leverage many-core architectures. Section 2.4 discusses the implications of these programming models on matching.

2.1 | MPI matching

The message matching engine is one of the major synchronous data structures in MPI. It is used to match incoming data to a user buffer specified in a receive function call. This matching is done by comparing three identifiers, a user defined tag, the identifier of the sender (rank), and the communicator[†] context. This data structure is constrained by behavior guarantees in the MPI specification.

First, the data structure must adhere to MPI's ordering constraint. Section 3.5 of the standard defines ordering of a point to point communication within a communicator.[7] The ordering requirements consists of two rules. If two receives both match the same message, the first must be fulfilled before the second can match said message. Similarly, if a process sends two messages to the same destination that match the same receive, the first message must be matched before the second message can match the said receive.

Second, receives must support wildcards. The MPI standard supports two wildcards, namely, any source and any tag. These allow a receive to match against a wider range of tags and can be useful for application behavior such as unexpected communication between two nodes. For example, AMG[8] uses this to establish communication outside of its regular neighborhood communication (halo exchange). There are current proposals to allow communicators to be created without the wildcard constraint, as this constraint limits optimizations for application that do not use these features.

Traditionally, MPI implementations have used a pair of linked lists to implement this: a posted receive queue (PRQ) and an unexpected message queue (UMQ). Figure 1 illustrates the process. To post a receive, the processes must search through the UMQ. If a matching message is found, the receive is fulfilled, the message is removed from the list, and the matching engine returns. If a matching message is not found, the process creates and appends a new element to the PRQ. An incoming message follows the same process but searches through the PRQ and appends to

[*]The matching engine refers to MPI's internal process for determining where to place incoming network data given a list of user specified buffers and message identifiers (a sending address or rank and a user defined identifier or tag.

[†]An MPI communicator defines an isolated communication context; communicators can include a subset of MPI processes in a job, remap the address space, and provide isolation to prevent matching collisions. The default communicator is MPI_COMM_WORLD, which allows for communication between all processes in a job.

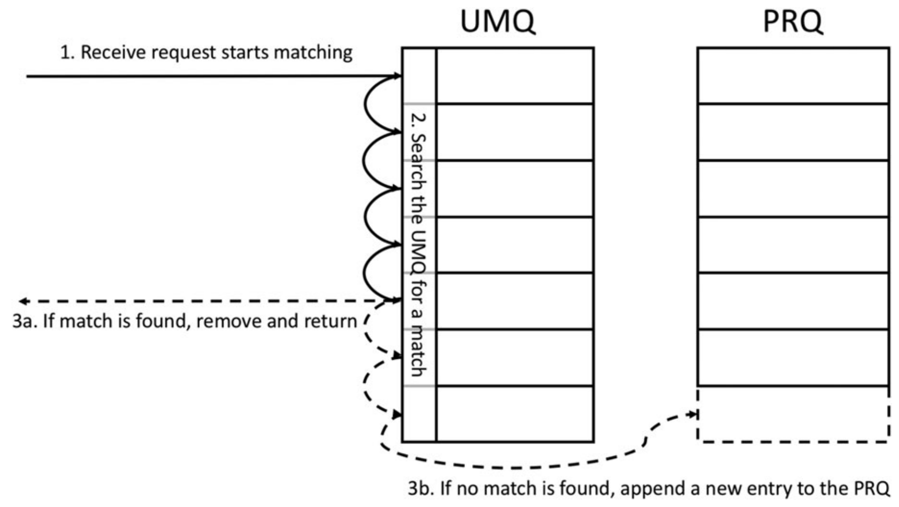
UMQ
PRQ
1. Receive request starts matching
2. Search the UMQ for a match
3a. If match is found, remove and return
3b. If no match is found, append a new entry to the PRQ

FIGURE 1 The Traditional Matching Process - to post a receive request, one searches the UMQ for a match, if it is not found it is appended to the PRQ. Incoming messages follow the same processes with the opposite lists

the UMQ. Due to the complexity of guaranteeing order and wildcard usage, most multithreaded implementations treat the matching engine as a singe large critical section.

This traditional matching engine appears in many open-source MPI implementation such at MPICH[9] and MVAPICH.[10] This is likely due to the fact that it is suited to and can be optimized for modern single-threaded applications, which leverage determinism to reduce search depths.[11] Implementing the leverage, other approaches include Open MPI,[12] which leverages a linked list per-peer approach, leveraging additional memory to reduce search depths.

2.2 | MPI thread levels

The MPI specification[4] has four defined thread modes that provide optimizations based on user guaranteed behavior. The most general is MPI THREAD MULTIPLE, which provides a thread-safe MPI interface. This mode has traditionally been implemented using coarse-grain synchronization methods, attempting to grab a lock upon entry to MPI. More recently, most implementations have been exploring fine-grain synchronization to minimize the performance impact of critical sections.

MPI THREAD SERIALIZED is used in most applications. This threading model allows for applications to call MPI from multiple threads but does not provide thread safety. MPI THREAD SINGLE and MPI THREAD FUNNELED allow for potential optimizations beyond MPI THREAD SERIALIZED. MPI THREAD SINGLE is for applications that will only use a single thread and MPI THREAD FUNNELED is for applications that will have multiple threads and only call MPI from a single thread. While these could theoretically be used to optimize memory operations (by taking advantage of thread-local storage), in practice, MPI implementations will simply run using the MPI THREAD SERIALIZED mode.

2.3 | Exascale applications models

The ratio of network injection bandwidth[‡] to CPU throughput, considered as a key metric for supercomputer performance,[13,14] has been decreasing.[15] Given this decrease, we expect systems to spend an increasing amount of time in the communication phase of a bulk synchronous program (BSP). To better leverage modern and future systems, applications must adopt new programming models.

Many programming models have been proposed to decrease network bottlenecks, especially as projections show parallel network access and data movement increasing. Many of these approaches are fine-grain messaging techniques that aim to alleviate the overheads of networking. These fine-grain messaging approaches leverage many, smaller, and non-blocking messages to distribute the messages without a dedicated communication phase. For example, communication and computation overlap proposes to use fine-grain messaging to distribute the communication to BSP computation phases where the network has traditionally been idle. This reduces the amount of data still awaiting transfer during the communication phase. This will create longer matching lists and more matching events as there are more messages in flight per iteration. Additionally, while this is easy to implement in single threaded MPI programs, threaded applications are more complex. Threaded applications leveraging these models will require a thread-safe way to access MPI, such as MPI THREAD MULTIPLE.

Asynchronous algorithms and task-based models such as charm++[16] are a series of more disruptive techniques that try to extend the concept of communication-computation overlap. The idea is to eliminate the synchronous phases entirely by decomposing computation into tasks that

[‡]Network injection bandwidth is the rate at which the network interface hardware can move data on to the network fabric

can be executed independently. Generally, applications of this nature can be represented by a directed acyclic graph of dependencies that do not necessarily have to be temporally co-scheduled. Because of the asynchronous nature of these algorithms, their communication orderings are significantly less deterministic, which will increase the average search depth.

2.4 | Implications for matching at exascale

Many-core processors make MPI matching non-performant.[3] Because of the limited cache size and out-of-order processing, cache misses are more frequent and expensive. This also applies to TLB misses. This is problematic for BSPs as the communication phase performance is degraded by these caching effects. While other application models may be less latency-sensitive, they require a performant MPI THREAD MULTIPLE implementation. Even with fine-grain locks in MPI, large critical sections, like the matching engine, would still negatively impact performance through blocking and lock contention. To adapt MPI to these new architecture paradigms, MPI must be able to handle concurrent requests while providing sufficient performance. To support this level of threading, MPI must enable as much parallel message processing progress as possible by parallelizing these large critical sections.

3 | THE TAIL-QUEUES ARCHITECTURE

The data structures that the MPI matching engine uses have a number of challenges that make matching difficult to parallelize. The wildcard and ordering constraints in addition to a race condition caused by the list dependencies make it difficult to utilize any list parallelism techniques. Tail Queues aims to provide more parallelism by isolating the shared critical section to the smallest portion of the race condition. The lists can then be parallelized in any manner that respects wildcards and ordering. Table 1 presents terminology that will be used throughout the rest of this section.

Figure 2 is a visual representation of the concept behind Tail Queues. This technique is achieved by creating inbox queues, shown at the bottom of the figure, that semantically represent the end of each respective list. This creates two queues, the PRQT and the UMQT, which are created to allow for lazy appends to the PRQ and UMQ. These inboxes segment the list in a way that allows us to append new entries without locking the other list. This allows us to encapsulate inter-list dependancies into the TL serialized region.

TABLE 1 Matching engine implementation terminology used in this paper

UMQ	Unexpected Message Queue
PRQ	Posted Receive Queue
UMQT	Unexpected Message Queue Tail
PRQT	Posted Receive Queue Tail
ML	Matching Lock
PRQL	Posted Receive Queue Lock
UMQL	Unexpected Message Queue Lock
TL	Tail Lock

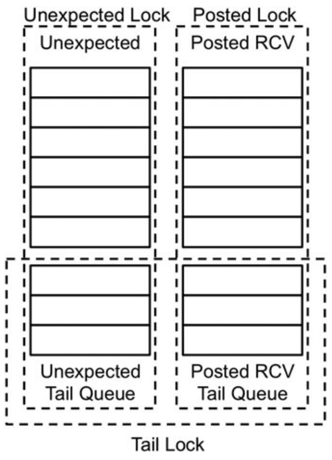

FIGURE 2 The Critical Sections of Tail-Queues. Each solid line box represents a matching element, which are arranged into the Posted RCV (PRQ) and Unexpected (UMQ). There is a separation between the main queue and the inboxes, the Unexpected Tail Queue (UMQT) and the Posted RCV Tail Queue (PRQT). The dotted boxes represent the data each of the three locks protect; the Unexpected Lock (UMQL) on the left, the Posted Lock (PRQL) on the right, and Tail Lock (TL) on the bottom. Note the PRQL and UMQL have separate locks and only overlap with the TL

Algorithm 1 Baseline MPICH CH3[9] - Post Receive

```
Input: Request with Tag, Source, and Context ID
Output: Matched Element (null if not found)
lock ML
for all element e in UMQ do
    if element matches Tag, Source, and Context ID then
        remove element from UMQ
        unlock ML
        return element
    end if
end for
append new element to PRQ
unlock ML
return null
```

Algorithm 1 shows one of the most common matching engine implementations. It is based on MPICH CH3's matching engine and is often found in many derivatives of that codebase, such as MVAPICH, Cray MPI, and Intel MPI. Because it uses one big list, it has three matching parameters: a user given tag, an expected source, and the communicator's context_id. With fine grained locks, the matching lock (ML) is locked upon entry and released on exit. The function then searches the unexpected message queue (UMQ), removes the first match and returns it or if no match is found in the UMQ, it appends a new entry to the posted receive queue (PRQ). For incoming messages, the process is similar, but the queues are switched.

Algorithm 2 Tail Queues - Post Receive

```
Input: Request with Tag, Source, and Context ID
Output: Matched Element (null if not found)
lock UMQL
for all element in UMQ do
    if element matches Tag, Source, and Context ID then
        remove element from UMQ
        unlock UMQL
        return element
    end if
end for
lock TL
for all element in UMQT do
    if element matches Tag, Source, and Context ID do
        remove element from UMQT
        append UMQT to UMQ
        UMQT = empty list
        unlock TL
        unlock UMQL
        return element
    end if
    remove element from UMQT
    append element to UMQ
end for
append new element to PRQT
unlock TL
unlock UMQ
reutrn null
```

Algorithm 2 shows the same function implemented for Tail Queues. The algorithm showcases the basic concept of isolating the inter-list dependencies. This can be extended to provide intra-list parallelism as we discuss below. It now acquires the unexpected message queue lock

(UMQL) upon entry and unlocks it upon exit. It searches the UMQ as usual, however, if a match is not found, it acquires the tail lock (TL) and continues the search through the unexpected message queue tail (UMQT). It removes all elements in said inbox and appends them to the UMQ itself. If it still has not found a match, it then appends a new entry to the posted receive queue tail (PRQT). If the TL is locked, upon exit, the processes unlock it.

This algorithm has a number of advantages. The basic implementation allows two threads to search the PRQ and UMQ simultaneously and exposes the potential for further parallelization of the PRQ and UMQ. For some of the results in this paper, we implemented a simple list parallelization in the form of binning. In this case, our approach creates a bin around the first 50 elements.[§] This allows for two threads to be working on the same list simultaneously. It also has the advantage of being compatible with the current state-of-the-art matching implementations. Because the PRQ and UMQ are fundamentally two separate structures, the technique of isolating the critical section using inboxes (the PRQT and UMQT) can be used with many different implementations including Open MPI's array of linked lists[17] and proposed hashing approaches.[5,18] This makes Tail Queues an interesting solution as any implementation could adopt it and provide parallel progress for message matching.

4 | EXPERIMENTAL RESULTS

This section presents an experimental study of two implementations of Tail Queues. Section 4.1 presents the details of the experimental setup. Section 4.2 presents the results of the experiments. Section 4.3 discuses the implications of the results.

4.1 | Experimental methodology

To measure the potential impact of the Tail-Queues algorithm, we created two implementations of the algorithm. First, we built a simple prototype of the algorithm external to MPI. Section 4.1.1 presents the details of this implementation and the details of how we evaluated this implementation. Section 4.1.2 presents details of the MPICH-based implementation and experiments.

4.1.1 | Prototype

To examine the performance characteristics in detail, we built a self-contained prototype implementation. This prototype implements a Tail Queues matching engine and a traditional matching engine, protected by a single lock, to serve as a baseline. To ensure that the prototype would fully exercise the data structures, we used MCS scalable locks,[19] which leverage list-based locking techniques to help avoid traditional problems that could impact the results like live-lock and thread starvation. These two implementations use the same linked-list and lock implementations to make a fair comparison. This prototype allows us to study how the match list can perform with varying amounts of parallelism without the need to manage coarse grained locks in sections of existing MPI libraries. In addition, this provides a high search rate that tests the locking mechanisms at high intensity, potentially highlighting any concurrency issues that could arise from the Tail Queues method.

We built a benchmark to compare the two implementations. The benchmark starts by posting a given number of posted receive entries and unexpected message entries to evaluate the algorithms at different queue lengths. The driver then starts a given number of threads. Each thread alternates simulating posting of a receive and the arrival of the corresponding incoming message. This experimental setup allowed us to explore the performance characteristics of each implementation by isolating the matching rate.

The platform for these tests was a small scale testbed cluster at Sandia National Laboratories. It is a Dual-Socket Intel Xeon Sandy Bridge, 2.6 GHz eight-core cluster with two Xeon Phi Knights Corner accelerator cards in each node. The Xeon tests were run on the node and the Xeon Phi tests were run in a shell on the accelerator. The data associated with these tests are the mean and standard deviation of 10 runs to ensure the numbers are representative on our experimental testbed system. Unless stated otherwise, the Queue Length independent variable represents the size of both queues.

4.1.2 | MPICH

To evaluate this technique further, we implemented Tail Queues in MPICH 3.1.4. This was done using the internal linked-list data structures and locking mechanism to be consistent with the MPICH baseline. Additionally, these tests show the results of independent linked-list parallelization in the form of binning. To test this, we modified the Sandia Microbenchmarks (SMBs)[20] to use multithreading and have different queue lengths. The SMBs aim to capture real world message rate. They accomplish this in a number of ways including making parallel transfers, clearing cache between iterations, and testing different communication patterns. Through injecting unmatched receives and messages, we can control the search depth of the queue during the benchmark. This allows us to test the effect of Tail Queues at different search depths, allowing us to reason about the potential impact on different applications. These tests represent the message rate of the single direction test for eight processes. The

[§]A 50 element bin is an arbitrary setup for the preliminary implementation, tuning and more complex approaches are left to future work.

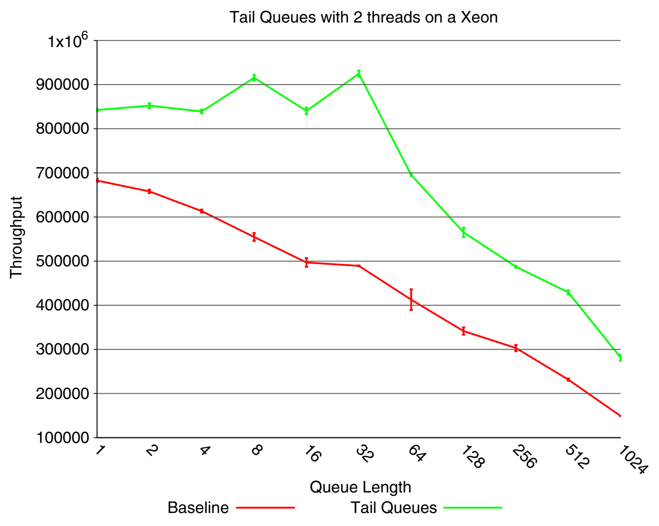

FIGURE 3 Tail queues prototype using two threads on a Xeon Sandy Bridge

SMBs' single direction is similar to a multi-bandwidth test, where half of the processes are designated as senders and send multiple successive messages to an assigned receiving peer.

The tests shown in this paper were run using the shared-memory netmod.[¶] Running in shared memory allowed us to capture the performance characteristics of each approach by removing the oudated InfiniBand network available on the system as a bottleneck. Additionally, the shared memory netmod currently supports fine grained locks. This provides the hardest experimental conditions for the Tail Queues locks, as they are exercised to the highest possible level of intensity on our test system. The MPICH tests were run in a single node with an I7 quad core processor. This limits the amount of parallelism available below the number of threads required by the SMBs (eight processors with two threads each) and thus these results should be a worst case for these techniques. These results are presented mainly as proof that the techniques are feasible and reasonable to implement in an MPI implementation that supports fine grained locking. The results presented represent the mean of 10 runs.

4.2 | Results

Figure 3 presents the results of running the shared memory test on a Sandy Bridge processor. There are a several interesting things to note here. First, the throughput shows a significant difference even at a small queue length. At a queue length of size 1, we see a 20% improvement. This is unexpected because the algorithm adds additional instructions to the code path and cannot do much work in parallel. However, Tail Queues results in reduced lock contention, since the contention has been spread out over many locks, which is a factor in improving performance. The difference peaks at 32 entries, where it gets close to a 95% improvement. This is where the threads are spending the majority of the benchmark working in parallel. Once both queues have 32 elements, both sides of the data structure require an equal amount of work, allowing for near maximum parallel work.

Figures 4, 5, and 6 show the results of Tail Queues with different thread levels on the Xeon Phi Knights Corner architecture. There are several interesting trends here. First, while the general trend lines are similar to the Xeon numbers, there are a couple of major differences: the maximum throughput is significantly lower, which is unsurprising since the cores are slower and less complex, and the maximum difference point has moved to be closer to 128 entries, where it appears to exceed 100% improvement, in some instances reaching close to 150% improvement. This further indicates that, in addition to increasing parallelism, we see an effect from reducing lock contention, like we observed for the Xeon processor results. Second is the general trend as thread counts increase. As this implementation only allows two threads to operate simultaneously, the lack of performance improvement as the number of threads increase is unsurprising. However, the decrease in throughput from two to four threads is surprising. This may be due to limited cache sharing on the Phis, increasing the cost of obtaining the lock. As we increase from four to eight threads, we observe the throughput growing to roughly the equivalent throughput of two threads.

Figure 7 shows the results of an experiment where the queue lengths only increase in the PRQ. This is done to show that, even with an imbalanced workload, Tail Queues can significantly increase performance. While the improvement is smaller than the other Xeon Phi experiments, the improvement is significant. While there is less work that can be done in parallel with two threads in this case, there is still an impact of the extra parallelism and reduced lock contention.

Figure 8 shows the preliminary results in an MPI implementation. These show roughly a 7.3% improvement to MPI message rate for Tail Queues plus binning over the baseline. At these queue lengths, MPI matching is significant enough of a bottleneck that we can see improvement on an

[¶]A netmod is the MPICH backend to support different data transport implementations. This term is implementation specific. OpenMPI, for example, has byte transport layers (BTLs) and matching transport layers (MTLs)

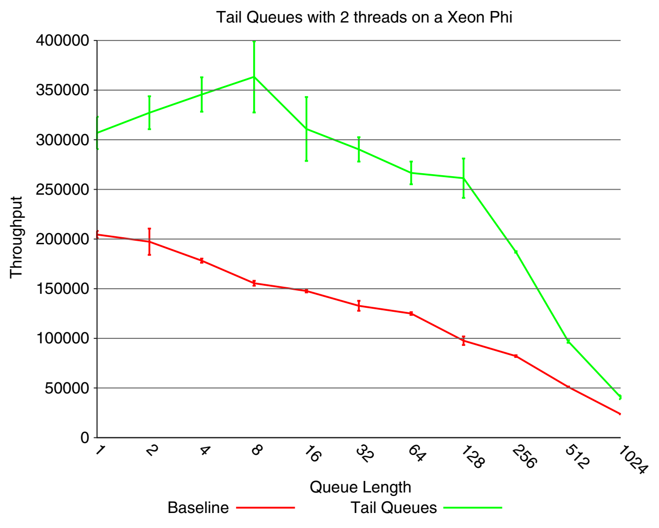

FIGURE 4 Tail queues prototype using two threads on a KNC

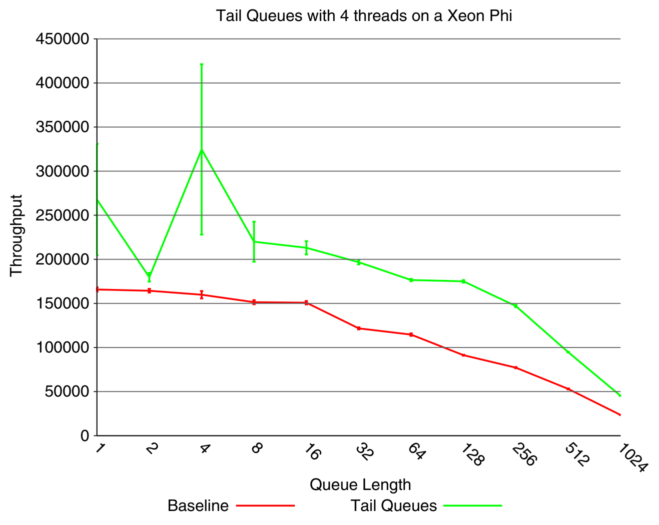

FIGURE 5 Tail queues prototype using four threads on a KNC

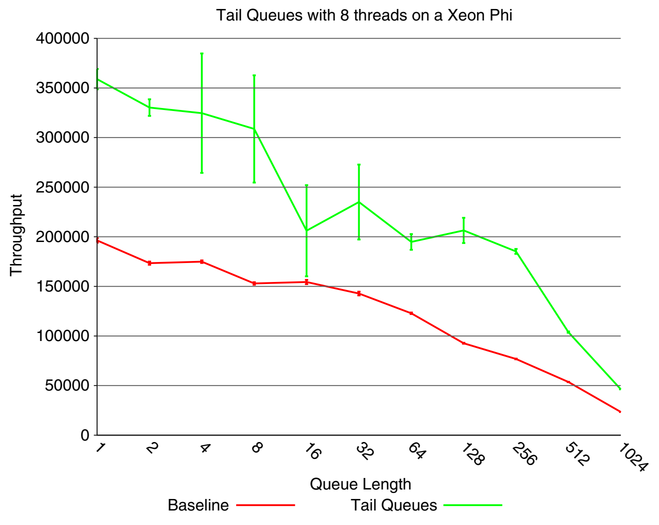

FIGURE 6 Tail queues prototype using eight threads on a KNC

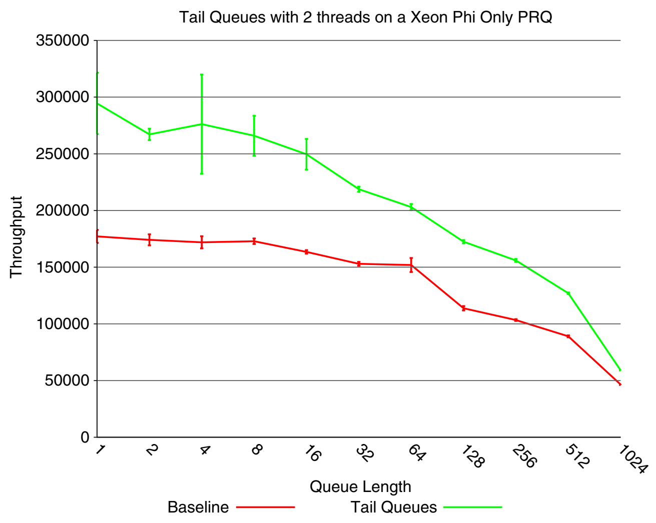

FIGURE 7 Tail queues prototype using two threads on a KNC - no UMQ entries

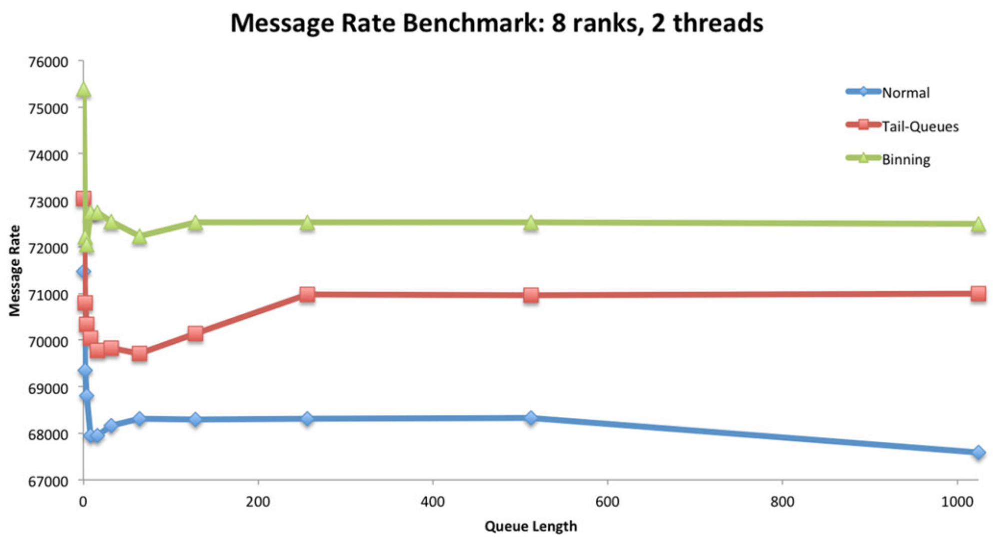

FIGURE 8 Tail queues MPICH implementation

oversubscribed CPU where the number of active threads is higher than the number of hardware thread contexts. These techniques would likely show improvement with more threads as this experiment only uses two threads when the binning implementation supports four-way–thread parallelism.

4.3 | Discussion

These results show that, by reducing the matching critical section and allowing for independent parallelization of the PRQ and UMQ, Tail Queues can improve MPI THREAD MULTIPLE performance. While matching is currently a small part of applications, as fine grained messaging and multi-threading lead to a large increase in requirement for message rate per MPI process, Tail Queues can help alleviate performance problems associated with the MPI Matching critical section.

There are also a number of challenges for these techniques to be effective. First, a fine-grained locking MPI implementation is required for this technique to be effective. Secondly, performant locking techniques are essential to identify for performance and understanding why MCS locks on the Xeon Phi become less performant at four threads is key to understanding how to properly optimize a Tail Queues–based matching engine. Finally, a tuned linked-list parallelization structure will allow for more performance as lists get long, which is non-trivial as the performance may depend on application behaviors such as average and distribution of matching search depth.

The goal of Tail Queues is to enable applications to leverage MPI_THREAD_MULTIPLE. It is difficult to evaluate impact on applications due to the lack of applications that utilize MPI_THREAD_MULTIPLE. While MPI_THREAD_MULTIPLE is desirable to support fine grained and parallel communication models, the lack of a performant implementation has prevented application developers from utilizing these code paths.

TABLE 2 Queue statistics from three applications run with 4096 processes

Application	PRQ Max Queue Length	PRQ Max Search Depth	UMQ Max Queue Length	UMQ Max Search Depth
MiniMD	1	1	11	8
AMG 2013	543	543	418	260
FDS	24	24	8147	8073

To explore the potential for matching parallelism in applications, we examined the queue statistics for three applications. Table 2 shows the results for three different applications at 4096 processes. These represent current codes and show a wide range of queue lengths and search depths. The evaluation in this paper is limited to list lengths of 1024 elements or less to represent the behavior of today's applications.[11] However, future multithreaded codes that use MPI parallel access through endpoints[21] or MPI THREAD MULTIPLE, queue lengths, and search depths are expected to increase in relation to the number of threads.[22] To utilize all of the hardware threads on a Knights Landing processor, one must utilize over 256 threads. Even if threads on a KNL were partitioned into a process per quadrant,[#] this could increase list lengths and search depths by 64x. This will make the matching engine even more of a bottleneck and will require leveraging the parallelism enabled by Tail Queues to remain performant.

5 | RELATED WORK

In this section, we discuss the current state-of-the-art research as it relates to this paper. Section 5.1 presents the current work on MPI_THREAD_MULTIPLE and Section 5.2 presents the current work on MPI message matching.

5.1 | Thread multiple

Recently, MPI_THREAD_MULTIPLE has been explored in depth. Gropp et al[24] discuss the implications and requirements of thread multiple to MPI implementations. There are a number of non-trival thread safety details of MPI. For example, many current applications, such as AMG,[8] require support for receiving messages of an unknown size. This is problematic as those applications use MPI_Probe to get the size of a message before posting a receive, thereby creating a critical section that encompasses the two calls. This is further illustrated by Hoefler et al,[25] who present a thread-safe mechanism to support receiving messages of an unknown size.

There has been work to improve thread-safe MPI implementations. While most current implementations have used a global synchronous lock, Balaji et al[26] apply fine grained locking to MPI, showing a performance improvement over global locking. By reducing the critical sections in MPI, they reduce the time spent waiting on locks. Additionally, previous models, such as MPI/Pro[27] and IBM MPI,[28] had developed methods of better thread support, but currently available solutions have difficulty replicating these approaches due to lock architecture. To further mitigate the cost of locking, Amer et al[29] propose a new synchronization arbitration to provide better fairness guarantees. This avoids potential starvation issues where one thread can be constantly blocked by other threads to the point if it falls behind in computation. Vaidyanathan et al[30] propose implementing a software offload mechanism to support thread multiple. They dedicate a core to do MPI processing and intercept the MPI calls to have a dedicated progress thread execute them. This is particularly useful for non-blocking calls as the calling thread can queue up a request and do other computation, whereas the offload thread progresses the communication.

There has been work to explore how to change the MPI programming model to better suit hybrid applications on highly parallel architectures. Stark et al[31] and Barrett et al[32] focus on exploring MPI+X task models that allows overdecomposition of a problem into a number of tasks that can then can be run by different MPI processes. Similarly, FG-MPI[33] is an execution model that extends MPI to enable task-based parallelism at a process level. Endpoints[34] is a proposal to allow MPI processes to separate some of the MPI state and processing by turning threads into addressable endpoints with a sub-rank. This could allow threads to access MPI in parallel for the parts of the processing that have been isolated. Finepoints[35] is another proposal that creates a new method of messaging that allows threads to specify a portion of a message in an MPI call. The underlying implementation can choose to aggregate and send this data to the receiver or can split the data transfers. This allows a process to complete a subset of the transfer while waiting on other threads to complete.

5.2 | Matching architecture optimizations

With the approach of exascale, there has been new effort to improve MPI matching. Newer approaches, like latest implementation of MPICH,[9] use more than one list, separating matching entries by communicator. This takes advantage of the fact that matching is encapsulated within a communicator. Open MPI[17] separates the matching engine further by creating a separate list for each peer in the communicator and a wildcard

[#] The Knights Landing processor partitions its core into quadrants that share some resources, such as memory busses. There are potential performance implications for some applications that run threads across multiple quadrants.[23]

list. Each communicator has an array of linked lists that can be indexed by the communication peer. A search must check both elements in both the specific list for specified peer and the wildcard list, compare the order of any matches, and return the first match.

There have been several more complex proposals to accelerate the matching engine without incurring a memory penalty. Zounmevo and Afsahi[36] proposed a 4-dimensional machining engine that scales in both performance and memory. This approach attempts to avoid entries in the match list by turning the search into a lookup in a 4-dimensional table. Rodrigues et al[37] explored the use of wide-width ALU operations in a Processing in Memory (PIM) architecture to process multiple MPI match operations in a single ALU operation. Wide-width ALU operations are similar to vector operations in modern CPU architectures but work on streams of data as they pass through the PIM device.

Flajslik et al[18] proposes a hash-bucket approach. In this approach, the match lists are fixed-size hash buckets that use matching data as a key to separate the linked lists. The number of linked lists and the hash function are configurable parameters. The evaluation done in this paper compares this data structure with the linked list approach and shows that the proposed design with 256 bins significantly reduces the number of match attempts per message. Bayatpour et al[38] extend the hash-table approach by creating a dynamic runtime approach to swap between hashing and traditional matching when appropriate. The selection of each approach is done at runtime. This approach shows up to a 2x speedup.

Klenk et al[5] introduce GPU-based message matching algorithm that is optimized by relaxing the constraints of MPI in terms of ordering and wildcard support. This algorithm is designed with two phases, scan and reduce, to leverage the available threads on a GPU. Their evaluation shows that this algorithm could provide a 10x to 80x increase in matching rate. This approach has three major caveats: it relies on a simplified set of MPI requirements, the existence of a GPU-style accelerator card, and having a sufficiently long matching lists to make use of all the threads in a GPU warp.[39]

6 | CONCLUSIONS AND FUTURE WORK

In this paper, we presented the novel parallel matching algorithm, Tail Queues. To the authors knowledge, Tail Queues is the first work to explore fine grained locking within a matching engine. This technique can also be leveraged to parallelize many of the other proposed matching structures. We discussed the preliminary implementations, both the independent proof of concept and an initial attempt at implementing the algorithm inside an MPI implementation. We then presented an evaluation that shows the improvement between this algorithm and the MPICH CH3 baseline. The technique shows promise in that it both improves performance and exposes more potential for parallelism.

There are a number of future steps to build on this work. First, an expanded evaluation would allow us to evaluate the algorithm in different contexts. Other matching papers evaluate their results to over 32 000 list elements. While 32 000 is unrealistic according to matching characterization studies[11,40] and our measurements shown in Table 2, match lists sizes over 1024 are possible, although uncommon. Expanding the evaluation beyond 1024 would explore the effects of these techniques on said applications. Including other MPI tests such as bandwidth, latency, and other forms of message rate as well as application proxies would also help expand the evaluation.

One of the major limitations of this study was the lack of MPI results on large-scale systems. This limitation was caused by the lack of support for fine grained locking. There are currently several efforts in progress to enable fine-grained locking for MPI_THREAD_MULTIPLE in major open source implementations such as Open MPI and MPICH's CH4 have both been working to improve multithreading. As new implementations that support fine grained locking become available, implementing the Tail Queues algorithm in those implementations and expanding the MPI-level testing with a wider variety of benchmarks would provide insight on the impact of this algorithm on full implementations.

The secondary parallelization method, binning, was a first attempt at further parallelizing the matching engine while maintaining strict ordering. Doing a wider exploration of possible techniques, including ones based on alternative matching structure such as Open MPI's array of linked lists and Intel's hash table structure,[18] would allow us to quantify the extent of the exposed parallelism.

Finally, there is a lack of applications and application proxies that exhibit the fine grained messaging behavior that we expect to see in the future. Develop such proxy application(s) to quantify the impact of forward looking matching techniques and better identify the need of these proposed programming paradigms.

ORCID

Matthew G.F. Dosanjh https://orcid.org/0000-0001-5141-9176

REFERENCES

1. Los Alamos National Laboratory. The Trinity Advanced Technology System. http://www.lanl.gov/projects/trinity/. Accessed March 19, 2015.
2. NERSC. NERSC-8 System: Cori. https://www.nersc.gov/users/computational-systems/cori/. Accessed March 19, 2015.
3. Barrett BW, Brightwell R, Grant RE, Hammond SD, Hemmert KS. An evaluation of MPI message rate on hybrid-core processors. *Int J High Perform Comput Appl.* 2014;28(4):415-424.
4. MPI Forum. MPI: A Message-Passing Interface Standard. Version 3.1. June 4th, 2015.
5. Klenk B, Fröning H, Eberle H, Dennison L. Relaxations for high-performance message passing on massively parallel SIMT processors. Paper presented at: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS); 2017; Orlando, FL.

6. Gropp W, Lusk E, Doss N, Skjellum A. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Comput*. 1996;22(6):789-828.
7. MPI Forum. *MPI: A message-passing interface standard version 3.0*. Knoxville, TN: University of Tennessee; 2012.
8. Baker AH, Falgout RD, Kolev TV, Yang UM. Multigrid smoothers for ultraparallel computing. *SIAM J Sci Comput*. 2011;33(5):2864-2887.
9. Team MPICH Development. CH4. Last accessed January 4, 2017. 2016.
10. Panda DK, Tomko K, Schulz K, Majumdar A. The MVAPICH project: evolution and sustainability of an open source production quality MPI library for HPC. Paper presented at: Workshop on Sustainable Software for Science: Practice and Experiences, held in conjunction with International Conference on Supercomputing (WSSPE); 2013; Columbus, Canada.
11. Ferreira KB, Levy S, Pedretti K, Grant RE. Characterizing MPI matching via trace-based simulation. In: EuroMPI'17 Proceedings of the 24th European MPI Users' Group Meeting; 2017; Chicago, IL.
12. Graham R, Woodall T, Squyres J. Open MPI: a flexible high performance MPI. In: PPAM'05 Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics; 2006; Poznań, Poland.
13. Alvin K, Barrett B, Brightwell R, et al. On the path to exascale. *Int J Distributed Syst Technol*. 2012;1(2):1-22.
14. Pedretti KT, Brightwell R, Doerfler D, Hemmert KS, Laros JH III. The impact of injection bandwidth performance on application scalability. In: EuroMPI'11 Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface; 2011; Santorini, Greece.
15. Roweth D, Atkins M, McMahon K. The Cray® XCTM Series Scalability Advantage.
16. Kale LV, Krishnan S. CHARM++: a portable concurrent object oriented system based on C++. In: OOPSLA'93 Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications; 1993; Washington, DC.
17. Team OPENMPI Development. OPENMPI. Last accessed March 28, 2017. 2017.
18. Flajslik M, Dinan J, Underwood KD. Mitigating MPI message matching misery. Paper presented at: International Conference on High Performance Computing; 2016; Sydney, Australia.
19. Mellor-Crummey JM, Scott ML. Synchronization without contention. *ACM SIGPLAN Notices*. 1991;26(4):269-278.
20. Doefler D, Barrett BW. *Sandia MPI MicroBenchmark Suite (SMB)*. Technical Report. Albuquerque, NM: Sandia National Laboratories; 2009.
21. Sridharan S, Dinan J, Kalamkar DD. Enabling efficient multithreaded MPI communication through a library-based implementation of MPI endpoints. In: SC'14 Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis; 2014; New Orleans, LA.
22. Schonbein W, Dosanjh MG, Grant RE, Bridges PG. Measuring multithreaded message matching misery. Paper presented at: European Conference on Parallel Processing; 2018; Turin, Italy.
23. Sodani A. Knights Landing (KNL): 2nd generation Intel® Xeon Phi processor. Paper presented at: 2015 IEEE Hot Chips 27 Symposium (HCS); 2015; Cupertino, CA.
24. Gropp W, Thakur R. Thread-safety in an MPI implementation: requirements and analysis. *Parallel Comput*. 2007;33(9):595-604.
25. Hoefler T, Bronevetsky G, Barrett B, De Supinski BR, Lumsdaine A. Efficient MPI support for advanced hybrid programming models. In: EuroMPI'10 Proceedings of the 17th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface; 2010; Stuttgart, Germany.
26. Balaji P, Buntinas D, Goodell D, Gropp W, Thakur R. Fine-grained multithreading support for hybrid threaded MPI programming. *Int J High Perform Comput Appl*. 2010;24(1):49-57.
27. Dimitrov R, Skjellum A. Software architecture and performance comparison of MPI/Pro and MPICH. In: ICCS'03 Proceedings of the 2003 International Conference on Computational Science: Part III; 2003; Melbourne, Australia.
28. Adiga NR, Almási G, Almasi GS, et al. An overview of the BlueGene/L supercomputer. In: SC'02 Proceedings of the 2002 ACM/IEEE Conference on Supercomputing; 2002; Baltimore, MD.
29. Amer A, Lu H, Wei Y, Balaji P, Matsuoka S. MPI+ threads: runtime contention and remedies. *ACM SIGPLAN Notices*. 2015;50(8):239-248.
30. Vaidyanathan K, Kalamkar DD, Pamnany K, et al. Improving concurrency and asynchrony in multithreaded MPI applications using software offloading. In: SC'15 Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis; 2015; Austin, TX.
31. Stark DT, Barrett RF, Grant RE, Olivier SL, Pedretti KT, Vaughan CT. Early experiences co-scheduling work and communication tasks for hybrid MPI+X applications. Paper presented at: 2014 Workshop on Exascale MPI at Supercomputing Conference; 2014; New Orleans, LA.
32. Barrett RF, Stark DT, Vaughan CT, Grant RE, Olivier SL, Pedretti KT. Toward an evolutionary task parallel integrated MPI+X programming model. In: PMAM'15 Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores; 2015; San Francisco, CA.
33. Kamal H, Wagner A. FG-MPI: fine-grain MPI for multicore and clusters. Paper presented at: 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW); 2010; Atlanta, GA.
34. Dinan J, Grant RE, Balaji P, et al. Enabling communication concurrency through flexible MPI endpoints. *Int J High Perform Comput Appl*. 2014;28(4):390-405.
35. Grant R, Skjellum A, Bangalore PV. *Lightweight threading With MPI Using Persistent Communications Semantics*. Albuquerque, NM: Sandia National Laboratories (SNL-NM); 2015.
36. Zounmevo JA, Afsahi A. A fast and resource-conscious MPI message queue mechanism for large-scale jobs. *Future Gener Comput Syst*. 2014;30:265-290.
37. Rodrigues A, Murphy R, Brightwell R, Underwood KD. Enhancing NIC performance for MPI using processing-in-memory. Paper presented at: 19th IEEE International Parallel and Distributed Processing Symposium; 2005; Denver, CO.
38. Bayatpour M, Subramoni H, Chakraborty S, Panda DK. Adaptive and dynamic design for MPI tag matching. Paper presented at: 2016 IEEE International Conference on Cluster Computing (CLUSTER); 2016; Taipei, Taiwan.
39. Nickolls J, Buck I, Garland M, Skadron K. Scalable parallel programming with CUDA. Paper presented at: 2008 IEEE Hot Chips 20 Symposium (HCS); 2008; Stanford, CA.

40. Klenk B, Fröning H. An overview of MPI characteristics of exascale proxy applications. Paper presented at: International Supercomputing Conference; 2017; Chicago, IL.

How to cite this article: Dosanjh MGF, Grant RE, Schonbein W, Bridges PG. Tail queues: A multi-threaded matching architecture. *Concurrency Computat Pract Exper*. 2019;e5158. https://doi.org/10.1002/cpe.5158