Measuring Multithreaded Message Matching Misery

Whit Schonbein[1,2(✉)], Matthew G. F. Dosanjh[1], Ryan E. Grant[1,2], and Patrick G. Bridges[2]

[1] Sandia National Laboratories, Center for Computing Research, Albuquerque, USA
{wwschon,mdosanj,regrant}@sandia.gov
[2] Department of Computer Science, University of New Mexico, Albuquerque, USA
bridges@cs.unm.edu

Abstract. MPI usage patterns are changing as applications move towards fully-multithreaded runtimes. However, the impact of these patterns on MPI message matching is not well-studied. In particular, MPI's mechanic for receiver-side data placement, message matching, can be impacted by increased message volume and nondeterminism incurred by multithreading. While there has been significant developer interest and work to provide an efficient MPI interface for multithreaded access, there has not been a study showing how these patterns affect messaging patterns and matching behavior. In this paper, we present a framework for studying the effects of multithreading on MPI message matching. This framework allows us to explore the implications of different common communication patterns and thread-level decompositions. We present a study of these impacts on the architecture of two of the Top 10 supercomputers (NERSC's Cori and LANL's Trinity). This data provides a baseline to evaluate reasonable matching engine queue lengths, search depths, and queue drain times under the multithreaded model. Furthermore, the study highlights surprising results on the challenge posed by message matching for multithreaded application performance.

1 Introduction

As the number of cores per node increase, scientific codes are moving toward hybrid model of parallelism combining an inter-process communication model,

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

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

Under the terms of Contract DE-NA0003525, there is a non-exclusive license for use of this work by or on behalf of the U.S. Government.

© National Technology & Engineering Solutions of Sandia, LLC 2018
M. Aldinucci et al. (Eds.): Euro-Par 2018, LNCS 11014, pp. 480–491, 2018.
https://doi.org/10.1007/978-3-319-96983-1_34

such as MPI, with a threading model. Due to performance concerns with MPI implementations, most contemporary codes leverage a hybrid BSP model where computation phases fan out to use multiple threads and communication phases filter down to a single thread. However, there is significant developer interest in leveraging MPI in a multithreaded manner to increase communication and computation overlap, decrease thread fan in/out overheads, and reduce development overheads.

While some studies address improvements to MPI's multithreaded codepaths, few assess how multithreaded communication affects the behavior of MPI message processing. Specifically, in single-threaded contexts, determinism in communication patterns allows users to ensure performance by ordering receive requests to match the corresponding sends. In contrast, non-determinism introduced by multithreaded communication may undermine this optimization, leading to increased message processing times. Furthermore, since the common strategy of packing data into a few large messages is likely to be discarded in favor of having each thread send smaller messages, the issue may be exacerbated by the increased number of messages. Since the acceptable performance of many current scientific codes is based on the assumption MPI message processing overhead is small in comparison to time spent in computation phases, it is of critical importance we grasp the implications of multithreaded communication so that appropriate steps can be taken in advance of the exascale timeframe.

In this paper, we explore the impact of increased messaging and decreased determinism in message ordering on MPI message processing. This study explores this impact on widely-used, simple, and highly-scalable stencil communication patterns that limit communication to a minimal number of peers. We introduce a model for the number of threads engaged in inter-process communication, and messages exchanged, when these stencil patterns are converted to multithreaded messaging in straightforward ways. We then empirically assess the effects of these patterns on average search depths and times. The results of these tests are surprising to us as MPI experts, as the MPI queue search depths are worse than expected. This means that MPI multi-threaded message processing overhead will be unacceptable when compared to the current performance of scientific codes.

The contributions of this paper are:

- A theoretical analysis of the characteristics of different thread-level decompositions for common stencil communication patterns;
- A testing structure enabling experiments of the effect of different thread-level decompositions on MPI message matching;
- An empirical study of the effects of threading on average search depths and queue drain times for MPI message matching.

The rest of this paper is organized as follows: Sect. 2 explores the background of this work including MPI Matching and MPI thread multiple. Section 3 presents our analysis of thread decompositions of different stencil patterns that we explore in this paper. Section 4 presents our empirical study of multithreaded non-determinism on search depth, list length, and queue drain time. Section 5

presents the state-of-the-art related work to this study. Finally, Sect. 6 presents the conclusions and implications of this work.

2 Background

In this section we present relevant background on MPI message matching and multithreaded MPI.

2.1 Message Matching

Message matching is MPI's receiver-side data-placement mechanic, used primarily to support point-to-point communication. To send a message (e.g., MPI_Send or MPI_Isend), an MPI process specifies a buffer containing data to be sent, a destination ID ('rank'), and a placement identifier ('tag'). The receiving MPI process posts a corresponding receive (e.g., MPI_Recv or MPI_Irecv) specifying a buffer where data will be placed, the rank of the sender, and the tag of the expected message. The communication is completed when the receiver matches the sending rank and tag of an incoming message to that of a posted receive, and the payload delivered to the specified buffer.

The MPI specification imposes several constraints on receiver-side message matching. First, messages must be matched in the order their receives are posted. Second, the matching mechanism must allow wildcards for both rank and tag. To handle these requirements, traditional implementations use two linked lists: a list of outstanding receive requests in a posted receive queue (PRQ), and a list of unmatched messages in the unexpected message queue (UMQ). When an MPI process posts a receive, its UMQ is traversed to determine whether a message with the desired sending rank and tag has already arrived, and if not, the receive is appended to the PRQ. When a message arrives at that process, the PRQ is traversed to determine whether a receive with the required rank and tag has already been posted, and if not, the information is appended to the UMQ. MPI ordering and wildcard semantics are guaranteed by initiating searches from queue heads and appending to their tails.

For the purposes of this paper, we use a traditional model for message matching, based on the model used by MPICH [19] and its derivatives. Some other implementations have opted for different models. For example, Open MPI [20] utilizes an array of lists, indexed by sending rank, which can reduce average search depth at the cost of increased memory. The benchmark and results presented in this paper can provide a better understanding on how these optimized models will impact next-generation applications.

2.2 Multithreaded MPI

The MPI standard introduces four threading modes which can be chosen during initialization [15]. Three of these require the user to prevent simultaneous requests while the fourth provides thread-safety. This paper is concerned with

MPI's thread-safe mode, MPI_THREAD_MULTIPLE, which requires the underlying implementation be able to handle simultaneous requests from different threads.

While the prevalence of hybrid-parallel applications has risen, few have leveraged MPI_THREAD_MULTIPLE. This has been due to the community's perception of the performance implications of this mode [7]. These performance implications are not inherent in the MPI standard, but are often a result of the complexity of implementing that standard. Recently, there have been efforts to improve this performance through mechanisms such as fine grained locks [1,2], one sided communication [10], and software offloading [23]. These efforts have primarily looked at improving the mechanics of multithreaded MPI; there has been little work on the impact of multithreaded MPI access patterns on MPI processing such as message matching.

3 Analysis of Stencil Decomposition

In this section, we provide an analysis for several possible stencil communication patterns using thread-level decompositions. The analysis assumes, first, that the thread decomposition is uniform, and second, each thread is responsible for its own outgoing and incoming data. This has implications for number of messages received, but maintains memory management schemes used by current applications at the process level (Table 1).

Table 1. Notation

Symbol	Description
L_d	Length of decomposition along dimension d
T_r	Number of receiving threads
T_s	Number of sending threads
M_e	Number of messages across a 1d edge
M_s	Number of messages across a 2d surface
M_t	Total number of messages exchanged in BSP communication phase

Given these assumptions, the simplest pattern is a *naive* case, where each thread communicates with all of its neighbors. For example, if the problem domain allocated to an MPI process is decomposed into $L_x \times L_y$ threads, and the stencil is 9 point, then each thread posts 8 receives, and the MPI matching engine must handle $8L_xL_y$ total messages during each BSP communication phase, distributed across L_xL_y threads.

A more nuanced approach assumes threads need only communicate along a process' domain boundaries; intra-process communication is handled outside of the MPI message matching engine. This maps well to real-world applications, where shared memory is typically used for intra-process communication. Even if intra-process MPI calls are used, they often bypass internal data structures and processing.

In this scenario, the number of sending and receiving threads differ because of corners and edges of the decomposed domain, as well as the type of stencil. Here we provide analyses for the case of 2d, 9pt and 3d, 27pt stencil communication. The analyses of 5pt and 7pt stencils are omitted for space. Note these analyses make the additional assumption that the length of each dimension is $\geq$2 subdomains.

3.1 9 Point Stencil

$$M_e = 3L_e \tag{1}$$
$$M_t = 6L_x + 6L_y - 4 \tag{2}$$
$$T_r = 2L_x + 2L_y - 4 \tag{3}$$
$$T_s = 2L_x + 2L_y + 4 \tag{4}$$

A 9-point stencil is a communication pattern for a 2 dimensional split of the problem space based on a 2d, radius-1 Moore neighborhood. The pattern requires communication to each neighbor process, including the corners. Equation 1 shows the number of messages sent across a single edge of the domain. Under our assumptions, the number of messages crossing an edge is three times the number of subdomains along that edge. Equation 2 extends the previous equation, by summing the number messages across all four edges and removing overlap. Equation 3 counts the communicating internal threads by calculating the sum of all the subdomains touching an edge of the process's domain and subtracting the overlap. This formula is subject to the $L_d \geq 2$ limitation as the overlap at $L_d = 1$. Finally, Eq. 4 counts the external threads by calculating the sum of all the subdomains that touch an edge of the process's domain and accounting for the four corners that weren't previously counted.

3.2 27 Point Stencil

$$M_s = 9L_mL_n \tag{5}$$
$$M_t = 2(\sum_{m<n|m,n\in\{x,y,z\}} 9L_mL_n) - 4(\sum_{m\in x,y,z} 3L_m) + 8 \tag{6}$$
$$T_r = 2(\sum_{m<n|m,n\in\{x,y,z\}} L_mL_n) - 4(\sum_{m\in x,y,z} L_m) + 8 \tag{7}$$
$$T_s = 2(\sum_{m<n|m,n\in\{x,y,z\}} L_mL_n) + 4(\sum_{m\in x,y,z} L_m) + 8 \tag{8}$$

A 27-point stencil is a communication pattern for a 3 dimensional split of the the problem space, based on a 3d, radius-1 Moore neighborhood. The pattern requires communication to each neighbor process across edges and corners. Equation 5 shows the number of messages sent across a single surface of the domain. Under our assumptions, the number of messages crossing a surface is

the number of subdomains on that surface times 9. Equation 6 extends the previous equation, by summing the number messages across all six surfaces and removing overlap. Note in these equations, the notation $m < n | m, n \in \{x, y, z\}$ can be thought of as nested loops generating the products xy, xz, and yz; an alternative notation is $N \in \{x, y, z\}^{(2)}$, where N is a metavariable ranging over the *set* formed by the 'n choose k' operator. Messages going diagonally from an edge are counted twice and are thus removed. Corner communication is counted three times but removed three times by accounting for the diagonal edge, and so are re-included. Equation 7 counts the internal threads by calculating the sum of all the subdomains touching an edge of the process' domain, subtracting the overlapped edges, and re-including corners. This formula is subject to the $L_d \geq 2$ limitation. Finally, Eq. 8 counts the external threads by calculating the sum of all the subdomains that touch an surface of the process's domain and accounting for the twelve edges and eight corners that weren't previously counted.

4 Experimental Results

4.1 Methods

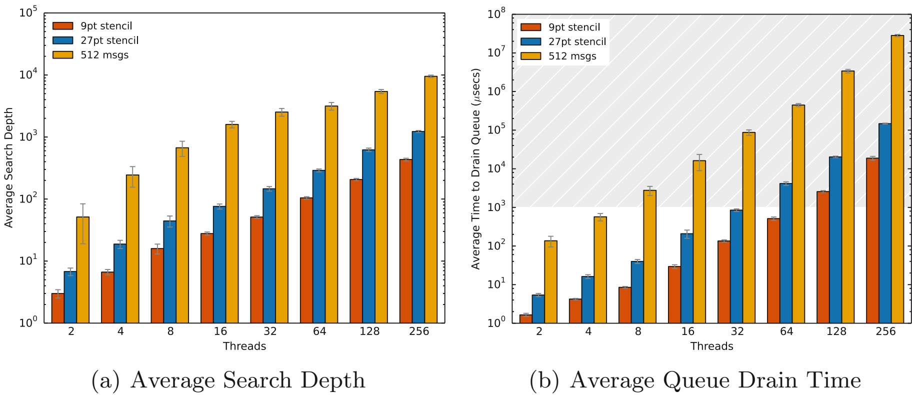
To investigate the effects of multithreading on MPI matching, we (i) instrumented MPI to report average PRQ search depths and time spent searching, and (ii) designed a benchmark to utilize MPI point-to-point communication in thread-multiple mode, while varying the thread count and total messages exchanged. For the former, an Open MPI development branch[1] was modified to use a matching engine mimicking that of MPICH. Open source MPICH does not provide support for our high-speed network, but is the basis for the vendor optimized MPI library on our system, therefore we used an open-source instrumented Open MPI with a MPICH style match list to best represent a fully-optimized vendor MPI. Since all messages originate from the same sending process, the list length under Open MPI's native matching engine is the same, although lengths for Open MPI matching can be roughly estimated from the results given below, by dividing by the anticipated number of sending MPI processes.

The benchmark emulates the behavior of an MPI process participating in bulk synchronous parallel (BSP) application with multi-threaded communication. Two nodes are allocated, each hosting a single MPI process. One is designated the receiving process, while the other serves as a proxy for the sending processes in the communication pattern. In an openMP region utilizing T_r threads, the receiving process pre-posts M_t receives; each message is given a unique tag. The order in which receives are posted is thus determined by thread scheduling and lock contention. Both processes barrier to ensure that all receives are pre-posted. The sending process then issues M_t sends, distributed across T_s threads, also in a free-for-all ordering incurred by a multithreaded region. To ensure fairness, tags are strided across sending and receiving threads. This provides a tag-ordering to the messages as m_i will have higher priority than m_j given $i < j$.

[1] Open MPI git hash f56847542eace89512aa482b186012d43fed7d4d.

As recent work has shown that some applications have queue lengths in excess of 1000 messages [11], the naive results include the case where each thread posts 512 receives, in addition to 9 and 27 point stencils. For the two- and three-dimensional decompositions, two stencils are considered for each: 5 and 9 point for 2d, and 7 and 27 point for 3d.

Experiments were run on a Cray XC40 using KNL nodes with 68 cores and four hardware threads per core, for a total of 272 possible threads. This system uses the Aries Interconnect. In all experiments, the receiving process is never oversubscribed. Since we only model threads at the boundaries of the decomposition, in some cases we are able to present data that goes beyond the expected number of total receiving threads for the system. We allow for oversubscription of sending threads. For the data points where this occurs the oversubscription is noted in the figure captions. To avoid overhead incurred by thread start up costs, no data is collected during initial trials. Runs are distributed across different nodes as determined by the resource manager (SLURM), and all values given are averaged across ten runs.



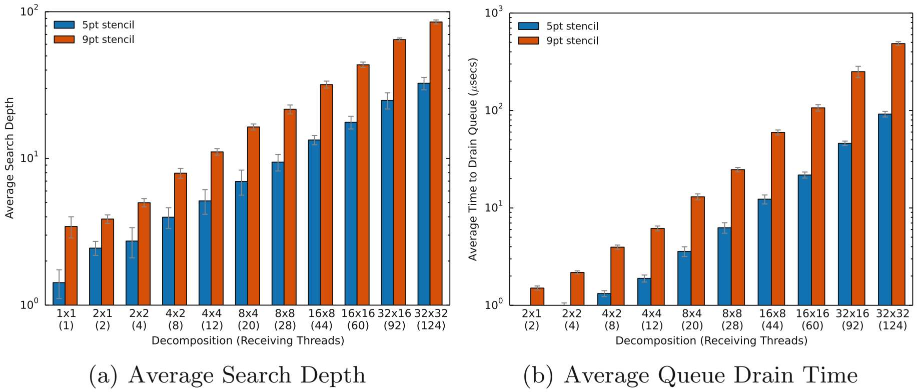
(a) Average Search Depth (b) Average Queue Drain Time

Fig. 1. Naive decomposition. Oversubscription does not occur. Grey region highlights drain times ≥1 ms.

4.2 Results

Figure 1(a) shows the average search depths observed for the naive decomposition using 9 and 27 point stencils (8 and 26 messages per thread, respectively), and 512 messages per thread. Average search depths increase rapidly as the number of threads grow. For instance, at 64 threads the average search depth for 512 messages-per-thread is over 3000 list elements, and the 27 point stencil exceeds 1000 at 256 threads.

Unsurprisingly, these inflated search depths translate into onerous search times (Fig. 1(b)). In this and subsequent graphs, the grey region highlights the range where drain times extend beyond 1 ms, which is problematic for many

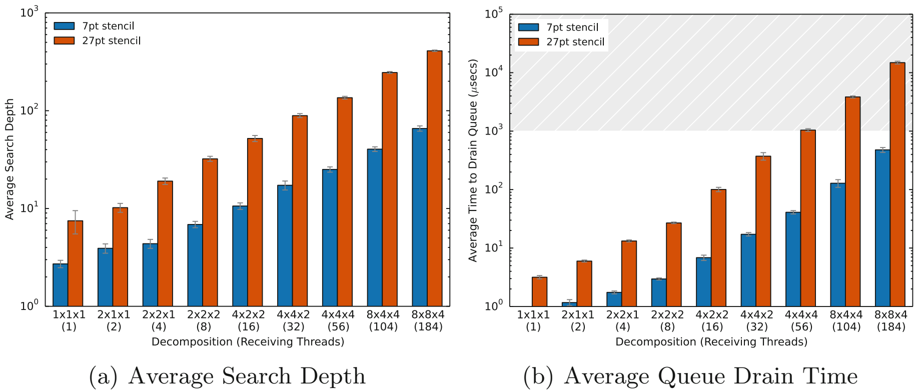


(a) Average Search Depth (b) Average Queue Drain Time

Fig. 2. 2D square domain decomposition. Oversubscription does not occur. Queue drain times for both stencils in the 1×1 condition are under one μsec, so are not shown.

codes (see discussion in Sect. 4.3). For instance, at 64 threads, the 27 point case requires, on average, more than four milliseconds to drain the queue, and at 256 threads requires 147147 ms.

More reasonable decompositions reduce search depths and times, but these remain surprisingly large (Fig. 2(a) and (b)). For instance, a 32-by-32 decomposition using a 5 point stencil has 124 receiving threads and 128 total messages, yielding an average search depth of 35.512 items and an average queue drain time of 91.78 µs; the 9 point stencil increases these to 85.18 items searched and 486.54 µs to drain the queue.

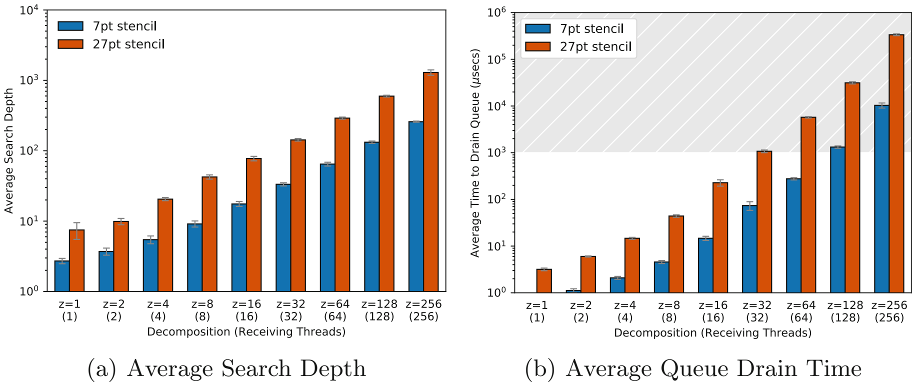


(a) Average Search Depth (b) Average Queue Drain Time

Fig. 3. 3D domain decomposition. Sending threads for 7pt never oversubscribe; those for 27pt oversubscribe at $8 \times 8 \times 4$ ($T_s = 344$). Grey region highlights drain times $\geq$1 ms.

Figure 3(a) and (b) show search depths and times for a 3d cube decomposition using 7 and 27 point stencils. For an $8 \times 8 \times 4$ decomposition where $T_r = 184$, a 7 point stencil results in $M_t = 256$ giving an average search depth of 65.85 items and an average queue drain time of 479.15 µs, while the 27 point stencil ($M_t = 2072$) results in 410.02 items and drain time of 14.86 ms. A less-ambitious decomposition to $4 \times 4 \times 4$ yields 56 communicating boundary threads. With a 7 point stencil ($M_t = 96$), we observe an average search depth of 25.1 items and drain time of 41.02 µs. Under the same conditions, the 27 point stencil ($M_t = 728$) has an average depth of 135.86 items and a time of 1044.17 µs.

Finally, Fig. 4(a) and (b) show results for another common 3d decomposition strategy, where the problem is decomposed only along the z axis. Because this decomposition has no internal cells, every thread communicates across the boundaries to neighboring MPI processes, putting additional stress on matching. For example, a $1 \times 1 \times 256$ decomposition ($T_r = 256$) using a 7 point stencil ($M_t = 576$) has an average search depth of 114.81 and a drain time of 3.29 ms, while the 27 point counterpart has a depth of 967.27 and time of 163.05 ms.



(a) Average Search Depth

(b) Average Queue Drain Time

Fig. 4. Linear 3D domain decomposition; x and y dimensions are both 1, while z varies. 7pt sending threads oversubscribe at $z = 128$ ($t_s = 514$), 27pt at $z = 32$ ($T_s = 274$). Grey region highlights drain times ≥1 ms.

4.3 Discussion

Single-threaded MPI codes often leverage deterministic communication patterns to optimize search so that search depths can be kept shallow (typically less than ten elements), even when matching lists grow long (a few thousand elements, total) [11]. Furthermore, contemporary hybrid MPI+X codes typically do not take advantage of thread multiple mode due to inefficiencies in current implementations. However, not only are these implementation issues being addressed for the exascale time frame (2020s), recent surveys show send/recv will remain the dominant programming model for exascale applications, and developers anticipate taking advantage of multi-threaded MPI communication [7]).

The results reported here suggest the matching overhead introduced by multi-threaded point-to-point MPI communication may be unacceptable for the future performance of some scientific codes. For example, molecular dynamics codes commonly use halo exchanges of the sorts investigated here. It is important to note that these halo-exchanges represent the highest scalability and lowest complexity of the communication patterns observed in scientific computing. From discussions with the developers of leading MD codes and comparative benchmarks [14,16], we observe the number of timesteps performed per second – where each step includes a halo exchange – ranges from tens to thousands (where each timestep simulates 1 femtosecond of time). This creates a total time budget per timestep of 100 ms to less than 1 ms. In the preceding timing graphs, this budget is highlighted in grey. This is the region where message matching overhead alone can exceed the iteration's time budget. Our results confirm this budget can be met for low thread-counts across all common communication patterns. However, as thread counts grow, and non-determinism increases, these same communication patterns can introduce more overhead than the entire current budget for completing a timestep at a competitive application speed. MPI matching overheads can take up to 30x to 300x the target iteration time for highly-scalable stencil communication patterns.

5 Related Work

Understanding MPI message matching has been a topic of interest that has been explored for single threaded MPI in the past. Initial work by Underwood and Brightwell [21] explored the performance impact of long lists. Further studies by Barrett et al. [3] showed the impact of match list length on a variety of system architectures.

A significant body of contemporary work exploring how to enhance the performance of MPI Message Matching exists, with some approaches looking to alter the matching list themselves to hash tables [12] or modifying the fundamental match list structures [24]. Other approaches have used a hybrid hash table approach, to accelerate common cases while providing long list performance [6]. Work using unique hardware features [17] and GPUs [13] has also been performed. Alternative solutions accelerate matching by not providing support for some MPI features [8]. MPI message matching hardware has also been explored [22] and specified/developed [4,9]. While hardware mitigates the long list matching performance concerns, it is limited in how many elements can be supported in the hardware match unit (typically 1K–4K). Despite this, recent work has shown that modern applications don't need these solutions: by leveraging programmer knowledge and sequential execution determinism, search depths can be kept low, even for long lists [11]. However, as noted above, many application developers expect to leverage communication libraries in ways that don't provide the same levels of ordering determinism that exist today [7]. To the best of the authors' knowledge there is no publicly available empirical data showing the effect of the lack of determinism on processes such as MPI message matching. While some new approaches with hybrid fine-grained over-decomposition of

computation has been done that would create large amounts of non-determinism in some cases [5,18], this work did not introduce the effect, as it serialized the threaded access to MPI in order to avoid the issues we explore in this paper (with the penalty of not having concurrent network accesses). The goal of this paper is to explore the effects of multithreaded non-determinism on message matching to enable new techniques as well as support traditional multi-threaded MPI access.

6 Conclusions

As we move towards exascale, we expect developers to both retain common stencil communication patterns under a send/receive model, and to take advantage of improvements in fully multithreaded MPI runtimes [7]. However, the potential impact of the nondeterminism introduced by multithreading on MPI's mechanic for receiver-side data placement – message matching – is not well-understood.

In this paper, we addressed this gap by characterizing the number of threads engaged in inter-process communication, and the number of messages exchanged, when common stencil patterns are converted to multithreaded messaging. On this basis, we conducted an empirical study of the consequences of multithreading for average message matching search depths and queue drain times, assuming a BSP model. Results indicate that under some decompositions and stencils, search depths and times may become unacceptable given current performance expectations.

References

1. Amer, A., Lu, H., Wei, Y., Balaji, P., Matsuoka, S.: MPI+ threads: runtime contention and remedies. ACM SIGPLAN Not. **50**(8), 239–248 (2015)
2. Balaji, P., Buntinas, D., Goodell, D., Gropp, W.D., Thakur, R.: Fine-grained multithreading support for hybrid threaded MPI programming. Int. J. High Perform. Comput. Appl. **24**(1), 49–57 (2010)
3. Barrett, B.W., Brightwell, R., Grant, R.E., Hammond, S.D., Hemmert, K.S.: An evaluation of MPI message rate on hybrid-core processors. Int. J. High Perform. Comput. Appl. **28**(4), 415–424 (2014)
4. Barrett, B.W., et al.: The Portals 4.0.2 networking programming interface (2014)
5. Barrett, R.F., Stark, D.T., Vaughan, C.T., Grant, R.E., Olivier, S.L., Pedretti, K.T.: Toward an evolutionary task parallel integrated MPI+X programming model. In: Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores, pp. 30–39. ACM (2015)
6. Bayatpour, M., Subramoni, H., Chakraborty, S., Panda, D.K.: Adaptive and dynamic design for MPI tag matching. In: 2016 IEEE International Conference on Cluster Computing (CLUSTER), pp. 1–10. IEEE (2016)
7. Bernholdt, D.E., et al.: A survey of MPI usage in the U.S. exascale computing project. Concurrency and Computation: Practice and Experience (2017, in Press)
8. Dang, H.-V., Snir, M., Gropp, W.: Towards millions of communicating threads. In: Proceedings of the 23rd European MPI Users' Group Meeting, pp. 1–14. ACM (2016)

9. Derradji, S., Palfer-Sollier, T., Panziera, J.-P., Poudes, A., Atos, F.W.: The BXI interconnect architecture. In: 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects (HOTI), pp. 18–25. IEEE (2015)
10. Dosanjh, M.G., Groves, T., Grant, R.E., Brightwell, R., Bridges, P.G.: RMA-MT: a benchmark suite for assessing MPI multi-threaded RMA performance. In: 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 550–559. IEEE (2016)
11. Ferreira, K.B., Levy, S., Pedretti, K., Grant, R.E.: Characterizing MPI matching via trace-based simulation. In: Proceedings of the 24th European MPI Users' Group Meeting, p. 8. ACM (2017)
12. Flajslik, M., Dinan, J., Underwood, K.D.: Mitigating MPI message matching misery. In: Kunkel, J.M., Balaji, P., Dongarra, J. (eds.) ISC High Performance 2016. LNCS, vol. 9697, pp. 281–299. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41321-1_15
13. Klenk, B., Froning, H., Eberle, H., Dennison, L.: Relaxations for high-performance message passing on massively parallel SIMT processors. In: 31st International Parallel and Distributed Processing Symposium (IPDPS). IEEE (2017)
14. Lindahl, E., Hess, B., Páll, S., Metere, A.: GROMACS 5.0 benchmarks (2017)
15. MPI Forum: MPI: a message-passing interface standard version 3.0. Technical report, University of Tennessee, Knoxville (2012)
16. Plimpton, S., Crozier, P., Thompson, A.: LAMMPS-large-scale atomic/molecular massively parallel simulator, vol. 18. Sandia National Laboratories (2007)
17. Rodrigues, A., Murphy, R., Brightwell, R., Underwood, K.D.: Enhancing NIC performance for MPI using processing-in-memory. In: 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS), p. 8–pp. IEEE (2005)
18. Stark, D.T., Barrett, R.F., Grant, R.E., Olivier, S.L., Pedretti, K.T., Vaughan, C.T.: Early experiences co-scheduling work and communication tasks for hybrid MPI+X applications. In: Proceedings of the 2014 Workshop on Exascale MPI, pp. 9–19. IEEE Press (2014)
19. MPICH Development Team: MPICH (2017). Accessed 30 Mar 2017
20. Open MPI Development Team: Open MPI (2017). Accessed 28 Mar 2017
21. Underwood, K.D., Brightwell, R.: The impact of MPI queue usage on message latency. In: International Conference on Parallel Processing (ICPP), pp. 152–160. IEEE (2004)
22. Underwood, K.D., Hemmert, K.S., Rodrigues, A., Murphy, R., Brightwell, R.: A hardware acceleration unit for MPI queue processing. In: 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS), p. 10–pp. IEEE (2005)
23. Vaidyanathan, K., et al.: Improving concurrency and asynchrony in multithreaded MPI applications using software offloading. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, p. 30. ACM (2015)
24. Zounmevo, J.A., Afsahi, A.: A fast and resource-conscious MPI message queue mechanism for large-scale jobs. Future Gener. Comput. Syst. **30**, 265–290 (2014)