An Overview of MPI Characteristics of Exascale Proxy Applications

Benjamin Klenk, Holger Fröning

Institute of Computer Engineering
Ruprecht Karls University, Heidelberg
Mannheim, Germany
{benjamin.klenk, holger.froening}@ziti.uni-heidelberg.de

Abstract. The scale of applications and computing systems is tremendously increasing and needs to increase even more to realize exascale systems. As the number of nodes keeps growing, communication has become key to high performance.
The Message Passing Interface (MPI) has evolved to the de facto standard for inter-node data transfers. Consequently, MPI is well suited to serve as proxy for an analysis of communication characteristics of exascale proxy applications.
This work presents characteristics like time spent in certain operations, point-to-point versus collective communication, and message sizes and rates, gathered from a comprehensive trace analysis. We provide an understanding of how applications use MPI to exploit node-level parallelism, always with respect to scalability, and also locate parts which require more optimization. We emphasize on the analysis of the message matching and report queue lengths and associated matching rates.
It is shown that most data is transferred via point-to-point operations, but the most time is spent in collectives. Message matching rates significantly depend on the length of message queues, which tend to increase with the number of processes. As messages are also become smaller, the matching is important to high message rates in large-scale applications.

1 Introduction

While there are many challenges on the road toward exascale computing, communication is key to both performance and energy efficiency. It is projected that an exascale computing system comprises 50 times more nodes than systems deployed in 2010 [1]. Additionally, the number of available processing elements increases even more as nodes become more parallel themselves, including massively parallel and heterogeneous processors.

Data movement within such highly parallel environments cannot rely on a single paradigm, but needs to be hierarchical and specialized. A single global address space is just as unpromising as solely relying on message passing. Computing has become heterogeneous and thus the processor's different execution models require different communication models [2].

Nonetheless, message passing has become the de facto standard for data movement between nodes as it abstracts communication to a well understood concept of messages being sent and received by a source and destination process. Besides high productivity, message passing allows messages to be sent asynchronously to overlap communication with computation, and provides collective operations, such as *barrier* and *broadcast*. In particular, the Message Passing Interface (MPI) is in wide use, especially in large scale applications. In spite of increasing heterogeneity, message passing is expected to remain the dominating communication model for data exchanges across operating system (OS) boundaries, even in future hierarchical communication systems.

With exascale computing ahead of us, application developers as well as system architects need to understand how data is exchanged. Applications have to be optimized to minimize communication overhead and systems have to provide an environment for the application to achieve best possible performance. Both cases require communication to be well understood in order to tweak applications and systems for performance.

Consequently, a set of MPI applications has been compiled by the U.S. Department of Energy (DOE), representing applications that are expected to run on exascale systems. Traces are provided that reflect the communication behavior on current systems with varying scale.

In this work, we analyze these trace files to provide an understanding of various aspects of message passing for such large-scale applications. Besides general statistics, such as overall communication time, message size, and data transfer volume, we provide a comprehensive analysis with regard to the message matching. The matching process significantly adds latency if long queues have to be searched in order to find a matching message. The matching is important, as it has been shown that solely speeding up the matching process can reduce an application's run time by a factor of 3.5x [3]. We report queue lengths as well as search depths and message rates of various exascale-like applications.

The results of this work can be used by application developers to understand consequences of various MPI aspects. Furthermore, systems architects learn about applications' demands, hence systems can be tailored to further accelerate common patterns. We also want to motivate programmers to consider similar analyses for their applications. In summary, we provide the following contributions:

- Comprehensive analysis of exascale proxy applications with respect to communication characteristics, such as message size and rate, number of communication partners, and time spent in particular MPI routines
- Analysis of queue lengths and search depths to further understand performance and implications of the matching process
- Based on our data, we discuss our observations and show limitations and challenges that arise at large scale

The remainder of this work is structured as follows: Section 2 provides the background on MPI and our methodology. Section 3 shows related work, followed by an overview of the applications we are analyzing in Section 4. Next, Section 5

reports general MPI statistics, while Section 6 particularly assesses the message matching process. We discuss our observations in Section 7 before we conclude in Section 8.

2 Background

In this section we want to introduce MPI as prominent and widely used message passing system. A brief overview of our methodology completes this section.

2.1 The Message Passing Interface

MPI has become the de facto standard for data transfers in High Performance Computing (HPC) systems, due to its productivity and abstract interface. Each data transfer is declared as a message that is sent and received by two processes. Messages are delivered according to their origin and destination, but also require to be annotated with a tag and communicator. The tag allows for selection of messages between the same process pair and the communicator is a subset of all available processes, but can also comprise all processes of an application.

A process receives messages by calling a *recv* routine. The receive request needs to be matched with the right message, based on origin, tag, and communicator. This is widely known as *tag matching* and can significantly contribute to latency [4].

An important aspect of the matching performance is the length of the Unexpected Messages Queue (UMQ) and Posted Receive Queue (PRQ). Any incoming message for which a receive request has not been posted yet is added to the UMQ. Similarly, any receive request is added to the PRQ for which no message has been received yet. With longer queues, the search time and thus latency is increased, particularly limiting the rate at which small messages are exchanged.

Apart from direct messages between two distinct processes, MPI allows multiple processes to participate in collective communication routines. Collectives are executed by all processes of the communicator that is passed to the MPI routine and allow for synchronization, such as the barrier. Others enable collective data processing, such as determining the maximum of data that is distributed across multiple processes, namely (all-)reduce operation. Collectives for plain data distribution are implemented by broadcast, gather, and scatter operations.

Other MPI extensions, such as one-sided semantics, are beyond of this work's scope as we did not encounter them in the traces we analyzed.

2.2 Methodology

The foundation of our work are the exascale proxy application traces, made available by the U.S. Department of Energy (DOE) [5]. The traces cover a wide range of applications with different communication patterns and characteristics. Traces of the *Design Forward* program comprise only a single iteration, whereas the other programs do not provide any information on this manner.

For our analyses, we developed a script-based framework in Python and R to parse and analyze trace files and verified results with a different approach using bash commands. Traces are available in the *dumpi* format [1], whose library intercepts and logs every MPI call with its entrance and exit timestamp. In addition to the calls themselves, routine-specific meta data is logged. While an *MPI_Send* contains the destination rank and message size, an *MPI_Allreduce* also comprises the operation that is executed on the data. Traces are available for each rank separately.

While some data can be gathered by simply parsing the files, more complex characteristics require additional processing. For example, the determination of MPI queue length and search depth requires the queues to be rebuilt and searched for every occurring *MPI_Send*, *MPI_Recv*, and *MPI_Wait(all)*.

Although plenty of insights can be gained by a trace-based analysis, there are limitations. For example, not all traces provide information on custom data types, thus the exact size of messages cannot be reported for all applications. Instead, we can only report the number of elements in given messages. Similarly, if a new mapping of ranks is generated by *MPI_Cart_create*, for example, the queues cannot be rebuilt easily. Furthermore, it remains unclear whether and how much communication is overlapped with computation as *dumpi* tracks MPI calls only.

Unfortunately not all applications provide information on the systems the traces were generated on. Applications from the *Design Forward* program also offer Integrated Performance Monitoring (IPM) data [2], which contains MPI time, message size distribution, and load balancing information. Nonetheless, we want to report these metrics for all applications and chose to report numbers from our own analyses. Note that these numbers can differ since metrics are collected by different methods.

3 Related Work

There are two fields that are related to this work: general MPI statistics and the analyses of matching and queues, respectively. A brief overview of existing work is presented in the following.

Early work on analyzing communication characteristics focused on the NAS Parallel Benchmark (NPB) suite [6] [7]. Their finding was that collectives are rather static, meaning that parameters can be determined at compile time and associated messages sizes are rather small. Furthermore, 5 out of 8 applications heavily use point-to-point communication with a share of more than 80%. Message sizes never exceed 64kB on 64 nodes in any NPB applications.

Vetter [8] and Kamil [9] looked at a small set of various applications and analyzed MPI metrics similar to our choice. They also found that in their set of applications the number of peer processes any rank communicates with and the message size of collective operations are rather small.

[1] `http://sst.sandia.gov/about_dumpi.html`
[2] `http://ipm-hpc.sourceforge.net/`

A similar analysis was done by Raponi et al. [10], in which MPI time and number of calls were studied. The set of applications differs from ours, except for *AMG*. However, they looked at small scale with only one application exceeding 512 ranks, while *AMG* was run with 128 ranks. The analysis showed similar results regarding the data transfer volume, which is strongly dominated by point-to-point communication. Traces were also analyzed by Lammel et al. [11], who proposes a trace analysis tool. Various MPI metrics are reported as well.

Other work also exists in the area of queue and matching analyses. UMQ and PRQ lengths were analyzed by Brightwell et al. [12], but only for the NPB applications again. They found that a significant amount of unexpected messages results in queue lengths of up to 200 entries with up to 140 processes. Furthermore, PRQ is always smaller than UMQ and average search lengths never exceed 30 entries. Note that they stated that it is necessary to analyze real applications, rather than benchmarks. Based on this, Underwood et al. [13] developed a list-acceleration unit in hardware. Benefits were shown as long as the queues fit in on-NIC memory. Keller et al. [14] analyzed large-scale applications, showing that the UMQ length scales linearly with the process count for a thermodynamics application on the Jaguar and JaguarPF systems. However, ranks other than 0 differ significantly with not exceeding a queue length of 200. Reported UMQ lengths for other applications are much smaller, ranging between 10 and 30 entries.

New matching algorithms have been proposed by Zounmevo et al. [15], aiming at reduced memory footprint and enhanced scalability. One algorithm uses multiple queues, statically assigned to ranks. Sequence numbers are used to comply with wildcards. Significant relative performance improvements were achieved for two applications (nbody and radix sort), but absolute numbers are missing. Flajslik et al. [3] also proposed a new matching algorithm, based on hash tables. Using this algorithm, the Fire Dynamics Simulator was run 3.5x faster by only replacing the matching algorithm and no further optimization of the application. The authors in [4] proposed a dynamic matching algorithm and reported matching times for several benchmarks. However, no queue lengths or search depths were analyzed.

In previous work [16], we analyzed the GPU's capability to perform message matching and proposed an appropriate algorithm. We found that the message passing protocol would need to be relaxed with regard to wildcards and ordering to suit the GPU's execution model. The conclusions can also be applied to the CPU's protocol to allow for more optimizations and faster matching, especially as the number of cores per CPU keeps increasing.

4 Application Overview

This section provides an overview of the applications we are analyzing in this work. Table 1 summarizes the applications' communication pattern and general statistics for each application. Note that numbers of our scalability analysis can differ from this table since we did not include small scale configurations.

Table 1. Exascale proxy applications and various MPI characteristics (Non.Bl. S/R = share of non-blocking Send/Recv operations ; Unxp.Msgs. = share of unexpected messages).

Application	Pattern	Ranks	MPI (Comm) time	Unxp.Msgs.	Non-Bl. S/R	Peers
MOCFE	Nearest	64	74 (8) %	n/a	100/100 %	2
(CESAR) *	Neighbor	256	86 (6) %	n/a	100/100 %	3
	(Near.N.)	1,024	92 (9) %	n/a	100/100 %	4
NEKBONE	Nearest	64	11 (7) %	40 %	99.9/99.9 %	18
(CESAR)	Neighbor	256	34 (11) %	35 %	99.9/99.9 %	8
		1,024	78 (23) %	45 %	99.9/99.9 %	29
CNS	Nearest	64	3 (2) %	28 %	0/92.5 %	26
(EXACT)	Neighbor	256	24 (20) %	40 %	0/98.5 %	44
CNS Large	Nearest	64	3 (3)%	30 %	0/60.8 %	26
(EXACT)	Neighbor	256	11 (11) %	27 %	0/85.7 %	20
		1,024	43 (39) %	34 %	0/98.4 %	72
MultiGrid	Nearest	64	6 (3) %	27 %	0/100 %	14
(EXACT)	Neighbor	256	16 (12) %	47 %	0/100 %	37
MultiGrid	Nearest	64	3 (1) %	40 %	0/100 %	14
Large	Neighbor	256	5 (3) %	31 %	0/100 %	17
(EXACT)		1,024	22 (18) %	33 %	0/100 %	20
LULESH	Nearest	64	1 (1) %	21 %	100/100 %	14
(EXMATEX)	Neighbor	512	8 (8) %	29 %	100/100 %	19
CMC 2D	Nearest	64	76 (76) %	n/a	n/a	n/a
(EXMATEX)	Neighbor	256	78 (78) %	n/a	n/a	n/a
		1,024	84 (84) %	n/a	n/a	n/a
AMG	Nearest	216	3 (3) %	44 %	100/100 %	57
(DF)	Neighbor	1,728	1 (1) %	46 %	100/100 %	79
		13,824	0 (0) %	48 %	100/100 %	92
AMR	Irregular	64	9 (5) %	27 %	0/99.9 %	18
Boxlib (DF)		1,728	12 (10) %	37 %	0/99.9 %	35
BigFFT	Many-	100	99 (3) %	n/a	n/a	n/a
(DF)	to-Many	1,024	99 (3) %	n/a	n/a	n/a
		10,000	99 (0) %	n/a	n/a	n/a
BigFFT	Many-	100	72 (29) %	n/a	n/a	n/a
Medium	to-Man	1,024	81 (19) %	n/a	n/a	n/a
(DF)		10,000	99 (1) %	n/a	n/a	n/a
Crystal	Staged	10	23 (23) %	46 %	0/100 %	3
Router (DF)	All-to-All	100	63 (63) %	31 %	0/100 %	6
Fill Boundary	Nearest	125	40 (27) %	34 %	0/100 %	16
(DF)	Neighbor	1,000	52 (44) %	30 %	0/100 %	20
		10,648	72 (70) %	32 %	0/100 %	23
MultiGrid	Nearest	125	40 (17) %	41 %	0/100 %	14
(DF)	Neighbor	1,000	66 (58) %	39 %	0/100 %	10
		10,648	70 (69) %	38 %	0/100 %	8
MiniDFT	Many-	125	15 (15) %	n/a	32/3.4 %	19
(DF) *	to-Many	424	11 (11) %	n/a	31.3/2.2 %	30
MiniFE	Staged	144	7 (6) %	n/a	0/100 %	12
(Mantevo) *	All-to-All	1,152	7 (6) %	n/a	0/100 %	15
PARTISN (DF)	*Near.N.	168	51 (50) %	n/a	0/0 %	1
Average	n/a	n/a	41 (21) %	36 %	n/a	23

* The queue analysis of this application was not possible since rank numbers are renamed, resulting from MPI's cart create.

The second column of the table shows the communication pattern of the applications. Although we analyze a wide range of applications, it seems that nearest neighbor communication is by far the most prominent one, whereas no application relies on pure all-to-all communication. Only *Crystal Router* and *MiniFE*, both from the *Design Forward* program, implement a staged form of all-to-all. *Crystal Router* and *AMG* use send/receive operations only and refrain from using any collective data transfer operation. *BigFFT (Design Forward)* and *EXMATEX's CMC 2D*, on the other hand, completely rely on collective communication.

We determine the number of peer ranks a rank is communicating with by counting how many different ranks are addressed with all send and receive operations together. On average across all applications, only a mean of 23 ranks participate in point-to-point communication with any given rank. It suggests that point-to-point communication is rather local, which allows for optimizations regarding process mapping and topology.

We also want to constitute that except for two applications, namely *MiniDFT* and *MiniFE*, we did not see any wildcard for the source specifier in any *MPI_Recv* operation. Wildcards introduce additional complexity in the message matching process, which seems quite unnecessary for the vast majority of applications. Additionally, we could not find tag wildcards in any trace file either, questioning whether MPI needs to support wildcards at the cost of complex matching algorithms. However, it is possible that trace files omit MPI's initialization phases, in which wildcards may be used more often. It would still be desirably if MPI allows the user to refrain from using wildcards during compute phases to allow for optimized message matching algorithms [3] [15].

Although messages within different communicators can be matched in parallel by replicating the associated data structures, applications do not seem to use multiple communicators. We observe that only *MiniDFT* groups ranks in 7 different communicators for point-to-point messages and *Nekbone* in 2, respectively, while all other applications rely on a single communicator. Given that communicators can be matched independently, we advocate to use multiple communicators to reduce matching overhead, allowing for higher message rates to be achieved.

The fifth column of the table shows the share of all messages that are unexpected. On average across all applications, 36% percent of all messages do not find a matching receive upon arrival and need to be placed in the UMQ. This is distributed as follows: applications and configurations with less than 100 ranks send 30% unexpected messages (15 samples), less than 500 ranks 34% (28 samples), and more than 1,000 ranks 39% (8 samples). Although the number of unexpected messages seems to increase with the scale of the application, the increment is not significant. Nonetheless, we observe a significant increase of unexpected messages with the number of ranks in the *AMG (Design Forward)* application, from 12% with 8 processes to 46% with 216 processes. However, increasing the scale to 1,728 processes has no further impact. Another example is *Crystal Router*, for which we observe that the number of unexpected messages

increases with scale, from 31% at 10 processes to 46% at 100 processes. However, more samples would be needed to allow for more profound statements.

A mechanism to avoid unexpected messages is to post non-blocking *MPI_Irecv* operations in advance to provide MPI with the appropriate user-space buffer for the expected message. We count occurrences of blocking and non-blocking send and receive operations for each application and found that no general statement for all applications can be made and refer to the results shown in the table. Nonetheless, it seems to be a common pattern to send messages in a blocking way and receive them by non-blocking receive operations.

We also want to state that *Design Forward's MiniDFT* is the only application that uses *MPI_Rsend* and *MPI_Sendrecv_replace* in addition to the standard blocking and non-blocking send routines. We have not observed any synchronous or buffered send operations in any other application.

Another important metric of any large-scale application is how much time is actually spent for data transfers versus computation. The accumulated time of all MPI calls is divided by the total application time, which is determined here by the first and last MPI operation that appears in the traces. Although this approach does not represent the exact application time as it does not account for non-MPI operations, it still allows for a good estimate. Second, the communication time is the accumulated time for all data transferring or synchronizing MPI calls, such send/recv, collectives, and *MPI_Wait(all)*. Comparing both times provides insights on how much MPI overhead an application contains. For example, overhead is increased by creating new datatypes, communicators, or groups. Both MPI and communication time are determined for rank 0.

The time spent in MPI routines averages about 36% of the application time across all applications and configurations (67 samples). If we consider only applications with less than 100 ranks, the MPI time averages 20% (24 samples), whereas applications with less than 500 ranks spent 27% (48 samples) of their time in MPI functions. The larger the scale the more time is spent in MPI, as applications with more than 500 ranks show an average MPI time of 57% (19 samples) and more than 1,000 ranks result in 60% (16 samples). This is not surprising as most traces are generated with the same input and problem size and the impact of communication usually increases with strong scaling. The actual communication time, however, is lower with an average of 20% across all applications and only 12% for applications with less than 100 ranks. Again, increasing the scale also increases the communication time as applications with more than 500 ranks show an average of 29%.

On average, 73% of the MPI time is spent for communication routines like send/recv or collectives. However, there are a few applications with significantly higher MPI than communication time. *Mocfe* and *BigFFT* both contain the most overhead with spending only 10-20% of their MPI time for communication and synchronization. For example, *Mocfe* (1,024) spends 75% of its application time a single MPI_Cart_create call. Thus, actual communication times may be higher during the application after initialization is complete.

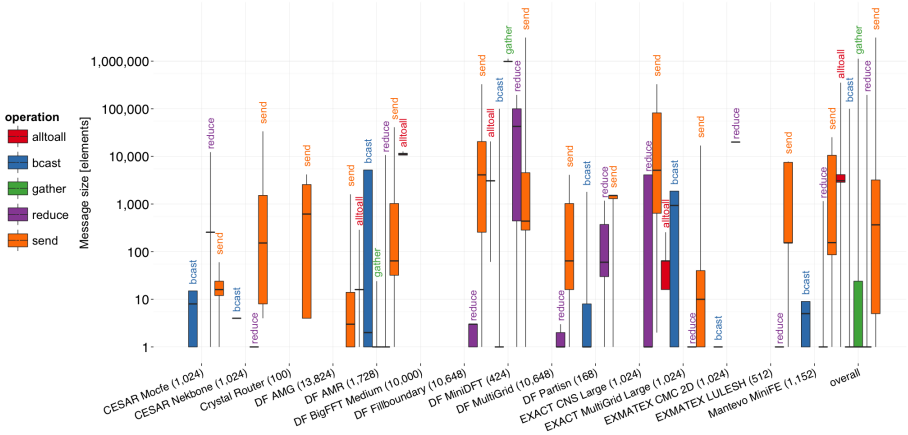


Fig. 1. Message sizes for various MPI operations and applications.

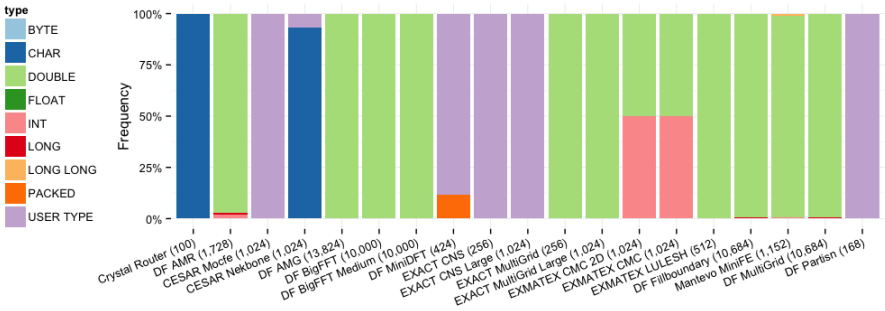


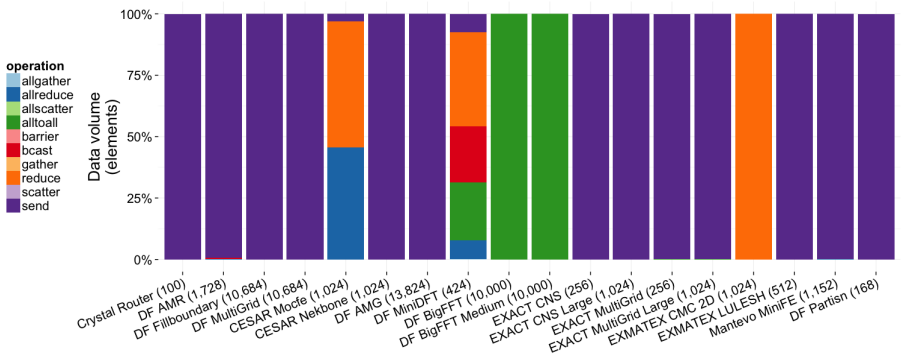
Fig. 2. Datatype distribution for each application.

Traces for different problem sizes are also available for some applications, namely *BigFFT (Design Forward)*, and *EXACT*'s *CNS* and *MultiGrid*. In all cases the MPI time is lower for larger problem sizes due to an increased amount of computation.

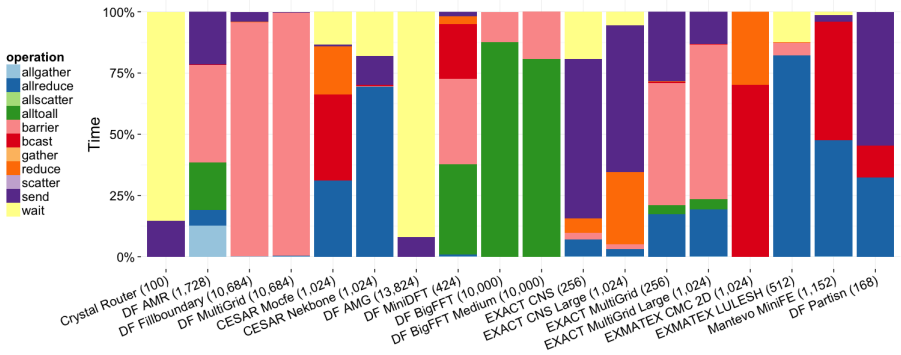
5 General MPI Statistics

This section presents our findings regarding general MPI characteristics, such as data volume, message size, and usage of various MPI operations and features.

Message size: The message size for common MPI operations and various applications is depicted in Figure 1. The graph shows the message size as a boxplot (1^{st}, 2^{nd}, and 3^{rd} quartile, minimum, maximum) and considers all messages from all ranks of a given application and configuration. At last, an overall message size distribution is shown across all applications and configurations, however, applications are not equally represented as some applications exchange much more messages. Generally it can be said that point-to-point messages contain



(a) Transferred data



(b) Communication time

Fig. 3. Transferred data and communication time, broken into various MPI operations

more elements than collective messages. In fact, collectives are often called with a single data element.

Taking the scale of applications into account, point-to-point messages tend to become smaller with an increasing number of ranks. This is observed in half of the applications (6 out of 12), whereas *Crystal Router*, *DF MiniDFT*, and *DF MultiGrid* show an increase in message size at larger scale. Contrary, messages remain roughly constant in *Mocfe*, *Fillboundary*, and *LULESH*.

Broadcasts are mostly unaffected by scale as only *MiniDFT* shows an increase in message size while other applications' messages remain constant in size. The message size of *alltoall* operations always decreases with an increasing number of ranks. This is observed in *BigFFT*, *EXACT MultiGrid*, and *AMR*.

The most prominently used collective operation is *(all-)reduce*. Here, the message size is constant over scale in two-thirds of the applications (6 out of 9). However, *MiniDFT* and *AMR* show an increase in message size. *Nekbone*, however, first uses larger messages when the scale is increased from 64 to 256 processes, but messages become smaller again for 1,024 processes.

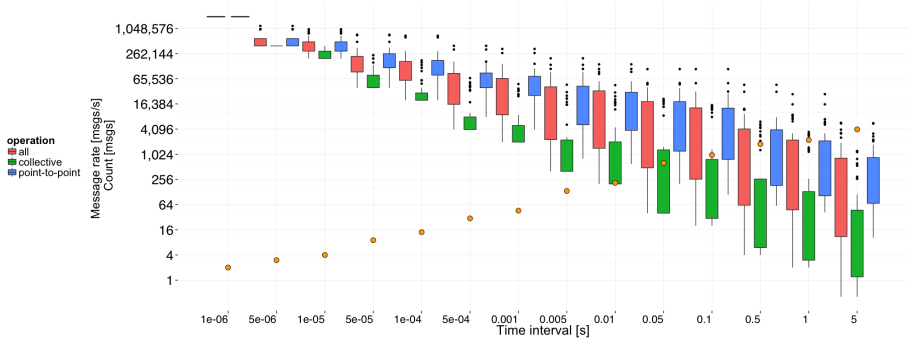


Fig. 4. Message rates for various time intervals, measured across all applications. The orange points indicate the number of messages (mean) that fall into a given time interval.

Summarizing it can be said that messages tend to become smaller or remain constant in size at larger scale. Nonetheless, a few applications show an increased message size, for example *MiniDFT*'s messages become larger for both point-to-point and collective messages.

Note that we cannot report the exact size of messages in terms of bytes since some traces lack information on the composition of user-defined types. However, Figure 2 shows the datatype distribution for each application. While most applications represent their data as *double*, user defined data structures are also prominent. Most applications use the same datatypes for point-to-point and collectives, however, there are exceptions. *Fillboundary*, for example, mainly uses *double* for point-to-point and *long* for collective operations. *Nekbone* uses *char* for point-to-point and user-defined types for collectives.

Transferred data volume: Figure 3(a) breaks down the total volume of transferred data into various MPI operations. As can be seen, most data is sent via point-to-point communication, except for a few workloads that primarily rely on collective operations for data movement. For example, *CESAR's Mocfe* and *EXMATEX's CMC* heavily rely on reduce and allreduce, respectively. *Design Forward's MiniDFT* almost entirely exchanges the data via all-to-all. Nonetheless, it is surprising how many applications rely on send/receive communication for data movement.

An interesting aspect is the data volume transferred during an application's run time. However, this is relative since we can only determine how many elements are sent, rather than the exact number of bytes. The most communication-intensive applications are *Crystal Router* (100 ranks) with 5.7G elements per application time, and *BigFFT* (1,024 ranks) and *Fillboundary* (10,648 ranks) with 3.3G elements/s each. Substituting application time with communication time yields different results. Here, *BigFFT* (10,000 ranks) is by far the most communication-intensive application with 112G elements/s, followed by *AMG* (13,824) and *LULESH* (512) with 37G and 24G elements/s, respectively. The lowest rate is achieved by *CMC* with 1.2K (64 ranks) to 24K (1,024) elements

per communication time. *DF MultiGrid* shows also low rates with an average of about 100K elements/s across 125, 1000, and 10648 ranks.

The transfer rate can also be determined with regard to the number of ranks. Here, *MiniFe* (18 ranks) shows the highest rate with 900M elements per communication time and rank, followed by *LULESH* (16) and *Crystal Router* (10) with each yielding about 400M elements/s per rank. Again, *CMC* also shows the lowest rate in this analysis.

The vast majority of applications responds to an increase in scale with a decrease in transferred elements per communication time. As we have shown earlier, messages tend to become smaller at larger scale and thus communication and synchronization overhead predominates at some point as well. Contrary, the total data volume tends to increase with scale in most applications.

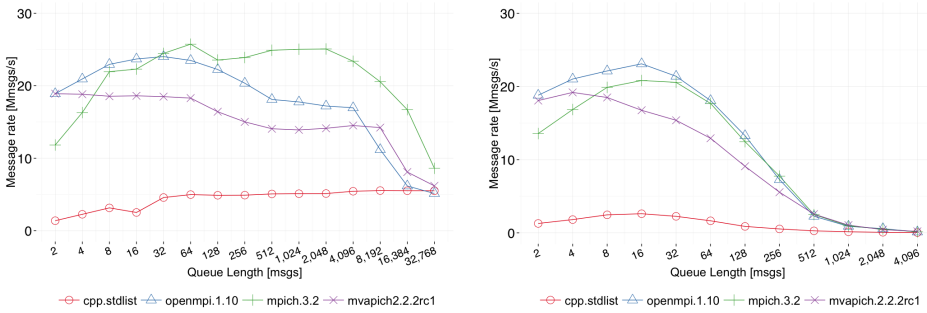
Communication time: Besides transferred data volume, the time spent in certain MPI operations is used to break down the MPI time. Results are shown in Figure 3(b). Although each application behaves differently, collective operations tend to contribute most to the application's MPI time. This is due their implicit synchronization and dependency on all ranks of the collective's communicator, whereas point-to-point communication just depends on two ranks, thus imbalances are less impactful regarding the operation's latency. However, only small amounts of data are moved by collective operations.

Looking at the graph suggests that point-to-point operations take less time than collectives, however, due to non-blocking send and receive operations the time spent in *MPI_Wait(all)* routines needs to be factored in as well. For example, *Design Forward's Crystal Router and AMG* spent most of their communication time on waiting for non-blocking operations to be completed, as it only uses non-blocking receive operations (see Table 1). Barriers, on the other hand, are especially time consuming in large-scale applications, such as *Fillboundary* and *MultiGrid*, both comprising 10,648 ranks.

Optimization of MPI communication needs to focus on collective operations and load balancing at large scale. While the data volume is lower than for point-to-point communication, the time spent in collectives is substantially higher.

Message rate: The message rate of an application can be determined by counting all messages that are sent during an application and divide the result by the application's run time. However, this does not reflect the application's requirements regarding the network's performance. A better approach is to define a time interval and count all messages that fall into it. If a bulk of messages is sent before a long period of computation, the message rate during the actual communication phase can demand high message rates from the network while the network could idle during computation.

Figure 4 shows the message rate observed across all applications for a given time interval and the number of messages that were counted during the time slot (orange points). As expected, high message rates are measured for short intervals, but the actual number of messages is fairly low. For example, 2 messages are exchanged within $1\mu s$, resulting in a message rate of 2M messages/s. The sample



(a) Best case (b) Average case (randomized)

Fig. 5. Matching rate of different MPI implementations for best and average cases.

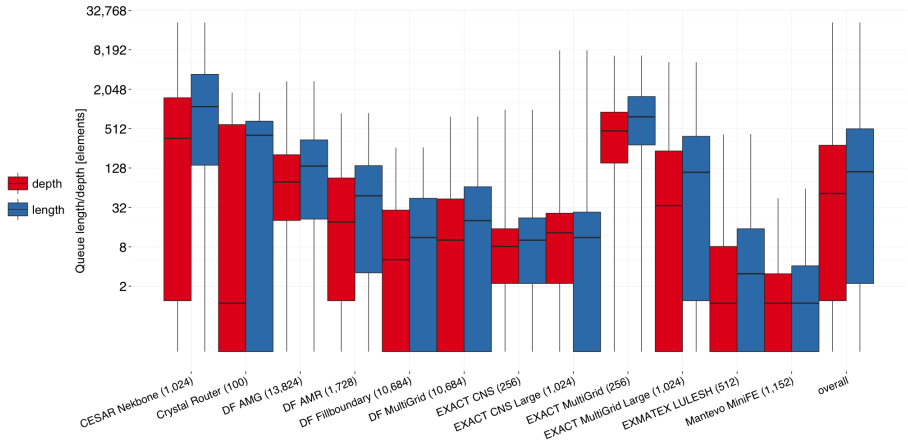


Fig. 6. Length and depth of the UMQ.

size is also low with 4 applications out of 48. We believe $100\mu s$ to be more representative as at least the median of the message count amounts to about 10 messages. The associated message rate is above 100k messages/s for half of the applications with the maximum being 500k messages/s. It can also be seen that the time between subsequent collective operations is larger than between point-to-point operations, also an effect that is caused by collective's implicit synchronization.

6 MPI Message Matching

Two-sided communication requires messages to be matched with the target's receive requests. The matching is complex as MPI guarantees in-order delivery of messages and allows for source and tag wildcards. Also, messages can arrive unexpectedly.

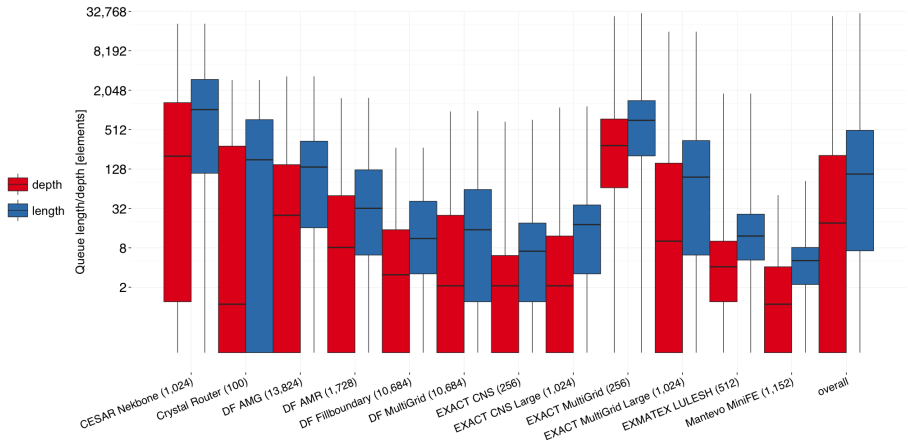


Fig. 7. Length and depth of the PRQ.

Matching performance: Messages and receive requests that cannot be matched need to be stored in queues, although all major MPI implementations implement lists. The length of these queues, or lists respectively, contributes to the memory footprint and increases latency if matching elements are found toward the queues' tail. Figure 5 shows the matching performance for two synthetic scenarios: best case, in which each receive request matches the head of the UMQ and average case with a randomized match position within the queue. The experiment has two MPI processes running on the same node, whereas one process sends a certain number of messages to the other process, followed by a barrier. The second process receives all messages after the barrier. The match position is determined by the tag used in the send and receive functions. The tag is ascending linearly for the best case and randomized for the average case. Our test system is a single node with an Intel Xeon CPU E5-2630 (Ivy Bridge) processor at 2.60 GHz with 1600 MHz DDR3 memory. We evaluate *OpenMPI 1.10*, *MPICH 3.2*, *MVAPICH 2.2.2rc1*, and a list implementation based on C++'s Standard Template Library (STL). Results are reported as average of thousands of iterations.

Results indicate that *OpenMPI* is optimized for small queues, while *MPICH* becomes superior for queues longer than 64 elements. The STL implementation is outperformed by far, demonstrating how MPI's lists are optimized for the matching purpose. Note that STL's queue container performs even worse due to its costly remove operation of elements at arbitrary positions.

Regarding the average case, matching rates drop significantly for queues longer than 32 elements, reaching half of the peak matching rate at queue lengths of about 128 elements. This can be observed for all MPI implementations. Note that we also determined the worst case matching rate with receive requests always matching the tail of the UMQ, however, the course of performance is similar to the average case with slightly lower absolute matching rates.

Queue lengths and search depth: As shown, the length of the queues has a significant impact on the matching time, thus also on latency and message rate. On

the other side, if receive requests always match messages at the queue's head, the actual length is unconcerned, though memory footprint is still affected. Consequently, the search depth is another important aspect. Together with the length, the search depth is shown in Figures 6 (UMQ) and 7 (PRQ).

Generally, both UMQ and PRQ show similar lengths and search depths, although the UMQ tends to be slightly larger in most applications. The longest queues are observed in Nekbone and MultiGrid with the median length being about 1,024. Overall, the length is smaller than 128 elements in half and smaller than 512 in 75 % of the measured moments. Note that we determine the queue length and search depth in any event of send/recv or wait operation and especially toward the end of the application the queues become often zero in length. Thus, queues may be larger during the application's most active communication periods.

Furthermore, it is also interesting to compare search depth and length. If both are similar, matches tend to be found rather at the end of the queue. However, if the search depth is smaller than the length matches are often found near the head of the queue. Regarding the UMQ, search depth is never significantly lower than the measured length. Significantly lower would mean that the depth's median is below the length's 1^{st} quartile. This is different from the PRQ, where at least 6 applications show significant lower depth than length.

While lengths of UMQ and PRQ are similar, the PRQ's search depth tends to be lower. If a receive is posted, the UMQ needs to be searched for a matching message and according to our results, the message tends to be found somewhere near the end of the queue. However, if a message arrives the matching receive tends to be found near the head of the PRQ. That suggests that the order of which receive requests are posted likely matches the order of which messages arrive, or receive requests match with multiple messages.

The median of all applications' search depth ranges below 100 elements, both for UMQ and PRQ. Combining this with the average case matching performance in Figure 5, the effective matching rate in most cases amounts to less than 18M matches/s. This is about 30% lower than the peak rate of 25M matches/s. Again, communication intensive periods might show even longer queues and thus the matching rate drops even further.

Strong scaling effects: The overall mean search depth of the mean across all of the application's ranks with less than 100 processes amounts to 29 elements, while the median is 6. *CESAR's Nekbone*'s search depth is 140 for 64 ranks, significantly adding to the mean. Considering all applications with less than 500 ranks, the mean search depth increases to 68, while the median increases to 14. On the other hand, the mean search depths for applications with more than 500 ranks amounts to 143 with a median of 38. The UMQ shows the same trends with higher absolute values. With applications sending more messages at larger scale, it is not surprising that queue lengths increase accordingly.

7 Discussion

This section summarizes the results and discusses our findings. While some insights are already widely assumed to be true in the community, this work aims to quantify these believes.

MPI time: We observed that a significant amount of time is spent in MPI routines, averaging about 36% across all studied applications. Directing research towards optimizing communication, especially MPI, is therefore important. Especially at large scale, MPI can easily contribute to more than half of the total application time. On the other hand, strong scaling applications show an increased amount of messages and data volume, but also a decreased amount of data per message. Consequently, large scale applications tend to send a significant amount of small messages. This emphasizes the need for low latency communication, thus rendering the matching of messages and receive requests even more important.

Breaking down the MPI time into various operations, we have shown that collective operations consume a significant amount of time compared to plain send/receive communication. This is mainly due to the implicit synchronization and the large number of processes that are involved. Contrary, the amount of data that is transferred collectively is rather small since the bulk of data is transferred via send/receive operations. We advise to use MPI's non-blocking collective operations to hide synchronization time through overlap with computation. Non-blocking collectives were introduced in MPI 3 [17].

Similar recommendations apply to send/receive communications as we suggest using non-blocking operations whenever possible. Note that instead of *MPI_Wait*, a non-blocking scheme using *MPI_Test* can be implemented to avoid busy waiting on requests. On average across all applications, almost 40% of messages are unexpected and all applications heavily use non-blocking receive operations. Nonetheless, *Design Forward's Crystal Router* and *AMG* spend still more than 80% of their MPI time in *MPI_Wait(all)*, possibly allowing for further optimization.

The last important factor that contributes to MPI time is the barrier. We observed that almost every application uses only a single communicator, even at large scale. Hence, a large number of processes participate in barriers, penalizing imbalances. Barriers need to be used carefully and programmers need to consider using more communicators and groups to avoid synchronization at large scale, as also advocated in [18]. However, we understand that this might not be applicable for all applications. Another solution could be the non-blocking barrier, introduced with MPI 3. Instead of busy waiting on the arrival of all processes, useful work could be done in the meantime.

It would be interesting to analyze the MPI time of certain operations even further to obtain a detailed understanding of limiting aspects. For example, the actual time spent for message matching can be assessed for point-to-point operations. However, traces in the present form do not allow for such an analysis and a more detailed profiling framework is required.

Message matching: The most dominant process within the whole matching of messages and receive requests is to search through unexpected messages and posted receives, respectively. We presented queue lengths and search depths to assess the matching performance. As we have seen, an optimistic 70% of the peak matching performance is achieved in most cases.

With an increased amount of messages and them becoming smaller at large scale, the importance of the matching increases as well. While the choice of algorithms is limited by MPI's in-order delivery, source and tag wildcards, and support for unexpected messages, not all of these features are required [16]. Although we understand that out-of-order delivery could require to restructure applications, we do not see a strong need for wildcards. Regarding the applications, none of them uses the tag wildcard and only two apply wildcards to the source. Alternative matching algorithms [3] [4] [15] steer in the right direction, but are still limited by wildcards. We suggest to support a mechanism that allows users to disable wildcards and select a more performant messaging mode. As for out-of-order delivery, tags can be used to re-establish ordering on user level. This allows to replace queue structures with hash tables, for example, enabling better performance.

Message rate and throughput: Surprisingly, we observed that message rates are rather low in all applications we have studied. Within 100μs, which we found to be a reasonable time interval, a median message rate of 100k messages/s is achieved with a maximum of 500k messages/s. Together with most applications' mean message size of about 1K elements/s for point-to-point and collective operations, an effective message rate of 100M elements/s is achieved. This translates to a throughput of about 400 MB/s for single precision and 800MB/s for double precision data, respectively. Since messages are most likely not aggregated, this is in the order of PCIe 2.0's bandwidth [19]. While it can also suggests that the message rate is limited by PCIe's bandwidth, our trace-based methodology is not sufficient to answer this question. Nonetheless, extremely fast interconnects are still limited by PCIe at end-point level. Tighter coupling of networking hardware and processors certainly steer into the right direction and will help to increase the network injection bandwidth.

8 Conclusion

We have presented and discussed several MPI characteristics of exascale proxy applications like time spent in certain operations, message size and rate, and queue lengths. Taking all applications with various scale into account, an application spends 36% of its time in MPI routines. Strong scaling applications with more than 1,000 ranks average an MPI time of even 60%. Most of this time is spent in collective operations, while the majority of data is transferred by point-to-point operations. We showed that messages become smaller at larger scale, emphasizing the importance of message matching.

We showed that search depth and queue length for both UMQ and PRQ are similar in most cases, suggesting that matching messages are rather found toward the end of the queues. Across all applications again, the median queue

length amounts to about 100 elements. This translates to a matching rate of 70% of peak performance. Again, scaling applications to a large number of processes renders queues longer and reduces matching rates even further.

Another important aspect we have shown is the effective message rate. We observed that message rates are rather low, so that within a time interval of 100μs only a median of 10 messages was counted, resulting in a message rate of 100k messages/s. Message sizes, on the other hand, show a median of about 512 elements/s for point-to-point and around 10 for collective operations.

While we gained valuable insights from the trace-based analysis, there are limitations and not all questions can be answered. The MPI time, for example, needs to be analyzed in more detail to understand what exactly is causing overhead. We assume that load imbalances lead to significant overhead for collective operations, but this needs to be verified using more detailed profiling frameworks, which is not trivial at large scale.

We also encourage operators of large computing facilities to provide more traces of their applications as access to these systems is often restricted. This allows for more applications to be analyzed and understood.

Acknowledgments

We would like to thank Hans Eberle and Larry Dennison from NVIDIA Corp. for insightful discussions that shaped the idea of this work. We also appreciate the U.S. DOE's effort of making the traces available to the public and thank all contributors for generating and providing the trace data.

References

[1] "The opportunities and challenges of exascale computing," summary report of the advanced scientific computing advisory committee (ASCAC) subcommittee at the US DOE Office of Science, Tech. Rep., 2010.

[2] B. Klenk, L. Oden, and H. Fröning, "Analyzing communication models for distributed thread-collaborative processors in terms of energy and time," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Philadelphia, PA, 2015.

[3] M. Flajslik, J. Dinan, and K. D. Underwood, "Mitigating MPI message matching misery," in *International Conference on High Performance Computing (ISC)*, Frankfurt, Germany, 2016.

[4] M. Bayatpour, H. Subramoni, S. Chakraborty, and D. K. Panda, "Adaptive and dynamic design for MPI tag matching," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, 2016.

[5] U.S. DOE, *Characterization of the DOE Mini-apps*, Retrieved 10/25/2016 from `https://portal.nersc.gov/project/CAL/doe-miniapps.htm`.

[6] A. Faraj and X. Yuan, "Communication characteristics in the NAS parallel benchmarks," in *IASTED International Conference on Parallel and Distributed Computing Systems (PDCS)*, Cambridge, MA, 2002.

[7] R. Riesen, "Communication patterns," in *Workshop on Communication Architecture for Clusters (CAC)*, Rhodes Island, Greece, 2006.

[8] J. S. Vetter and F. Mueller, "Communication characteristics of large-scale scientific applications for contemporary cluster architectures," *Journal of Parallel Distributed Compututing*, vol. 63, no. 9, 2003.

[9] S. Kamil, L. Oliker, A. Pinar, and J. Shalf, "Communication requirements and interconnect optimization for high-end scientific applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 2, 2010.

[10] P. G. Raponi, F. Petrini, R. Walkup, and F. Checconi, "Characterization of the communication patterns of scientific applications on Blue Gene/P," in *IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW) and PhD Forum*, Washington, DC, USA, 2011.

[11] S. Lammel, F. Zahn, and H. Fröning, "SONAR: Automated communication characterization for HPC applications," in *International Conference on High Performance Computing (ISC)*, Frankfurt, Germany, 2016.

[12] R. Brightwell and K. D. Underwood, "An analysis of NIC resource usage for offloading MPI," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Santa Fe, NM, 2004.

[13] K. D. Underwood, K. S. Hemmert, A. Rodrigues, R. Murphy, and R. Brightwell, "A hardware acceleration unit for MPI queue processing," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Denver, CO, 2005.

[14] R. Keller and R. L. Graham, "Characteristics of the unexpected message queue of MPI applications," in *European MPI Users' Group Meeting Conference (EuroMPI)*, Stuttgart, Germany, 2010.

[15] J. A. Zounmevo and A. Afsahi, "An efficient MPI message queue mechanism for large-scale jobs," in *IEEE Conference on Parallel and Distributed Systems (ICPADS)*, Singapore, 2012.

[16] B. Klenk, H. Fröning, H. Eberle, and L. Dennison, "Relaxations for high-performance message passing on massively parallel SIMT processors," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Orlando, FL, 2017.

[17] T. Hoefler, P. Kambadur, R. L. Graham, G. Shipman, and A. Lumsdaine, "A case for standard non-blocking collective operations," in *European Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface (PVM/MPI)*, Paris, France, 2007.

[18] P. Balaji, A. Chan, R. Thakur, W. Gropp, and E. Lusk, "Toward message passing for a million processes: Characterizing MPI on a massive scale Blue Gene/P," *Computer Science - Research and Development (CSRD)*, vol. 24, no. 1-2, 2009.

[19] M. J. Koop, W. Huang, K. Gopalakrishnan, and D. K. Panda, "Performance analysis and evaluation of PCIe 2.0 and quad-data rate InfiniBand," in *IEEE Symposium on High Performance Interconnects*, 2008.