Impact of the Entering Time on the Performance of MPI Collective Operations

Christoph Niethammer, Dmitry Khabi, Huan Zhou, Vladimir Marjanovic, and José Gracia
High Performance Computing Center Stuttgart (HLRS), University of Stuttgart
70569 Stuttgart, Germany
Email: {niethammer,khabi,zhou,marjanovic,gracia}@hlrs.de

***Abstract*—Collective operations strongly affect the performance of many MPI applications, as they involve large numbers, or frequently all, of the processes communicating with each other. One critical issue for the performance of collective operations is load imbalance, which causes processes to enter collective operations at different times. The influence of such late-arrivals is not well understood at the moment. Earlier work showed that even small system noise can have a tremendous effect on the collective performance. Thus, although algorithms are optimized for large process counts, they do not seem to tolerate noise or consider delay of involved processes and even a small perturbation from a single process can already have a negative effect on the overall collective execution. In this work, we show a first detailed study about the effect of late arrivals onto the collective performance in MPI. For the evaluation a new, specialized benchmark was designed and a new metric, which we call delay overlap benefit, was used. Our results show that there is already some potential tolerance to late arrivals for the most common collective operations - namely barrier, broadcast, allreduce and alltoall - but there is also a lot of room for future optimizations.**

Keywords*–collectives; late-arrivals; benchmarking; MPI collective operations

I. Introduction

Collective operations strongly affect the performance of many Message Passing Interface (MPI) applications, as they involve large number, usually all, of processes communicating with each other. One critical issue for the performance of collective operations is load imbalance, which causes processes to enter collective operations at different times. The influence of such delayed processes is not well understood at the moment. The results in this paper extend our initial work [1] on this topic. Earlier studies showed that even small system noise can have a tremendous effect on the collective performance [2] [3]. So, though algorithms are optimized for large process counts [4], they do not seem to tolerate noise or consider delay of involved processes and thus even a small perturbation from a single process can already have a negative effect on the overall collective execution.

The MPI 3.0 standard introduced non-blocking collective operations, which give the opportunity to speed up applications by allowing overlap of communication with computation [5], reducing the synchronisation costs of delayed processes as well as the effects of system noise. Many MPI programs are written using non-blocking point-to-point communication operations and application developers are familiar with managing this process using request and status objects. Extending this to include collectives allows programmers to straightforwardly improve application scalability.

In contrast to the already existing blocking collectives, the non-blocking counterparts require the MPI implementations to progress the communication task in parallel to computations. This is a non-trivial task, even if the network hardware provides support for offloading network operations from the CPU, e.g., message buffers may have to be refilled for large messages or more complex collective operations need multiple communication steps. The Cray XE6 and XC30 platforms feature a special “asynchronous process engine” for this, which uses spare hyperthreads (XC30) or dedicated CPU cores (XE6) for the required operations [6].

This work analyses and emphasizes the effect of late arrivals on collective operation in MPI for large number of processes. Therefore, a benchmark and metric for evaluation and detection of effects caused by late arrivals are introduced. The obtained results show the tolerance of state of the arty MPI collective operations used in the latests Cray XC30 and XC40 systems and may point to potential for improving performance by solving the issue of late arrivals in the future.

This work is structured as follows. Section III describes the testing methodology and the micro benchmark suite, which we designed specifically to study the impact of late arrivals, i.e., delay on collective performance. At the begin of Section IV, we define a metric to quantify the amount of tolerated delay. Then, the results for different application relevant collective operations are presented and evaluated on basis of absolute times as well as the delay overlap metric.

II. Background

Since long time, message passing is the standard when it comes to the programming of high performance computing (HPC) applications on distributed memory systems. Since its beginnings many different benchmarks have been developed to measure the performance of the underlying hardware, or to test the efficiency of the MPI library implementation. The most well known of them are the OSU Micro-Benchmarks, Ohio MicroBenchmark suite, Intel MPI Benchmarks and the Effective Bandwidth (b_eff) Benchmark [7]. All of these benchmarks test bandwidth and latency of the various MPI communication functions for different number of processes and data size. These benchmarks assume that communication happens in a perfect environment and all operations are well synchronized. While this allows to figure out the best performance reachable with the various MPI calls for the used hardware and MPI library, it does not represent the majority of real world applications.

At first, real world applications are influenced by system noise. So, compute nodes run services in parallel to the application, as for instance, time synchronization or node health checker, which interrupt the execution of the application. Also, there are shared resources like I/O or the network itself, which are used by other applications running at the same time. This noise can have a tremendous effect on the collective performance [2].

A second point, which is not taken into account by existing benchmarks are load imbalances inside the application. These load imbalances lead to different entering times of different processes at communication points. So called late arrivals will cause other processes to wait on them. The more processes are involved in the communication the more this becomes a problem, as equal load distribution becomes more complicated and the number of processes, which may have to wait on a single late arrival, increases.

Algorithms implementing collective operations use different strategies to optimize the network use and achieve the best possible performance [4]. Common technique here are tree based communication structures and ring sends to match the underlying network hardware on the one hand, as well as to reduce the number of send messages or reduce the bandwidth requirements on the other hand. Late arrivals in these communication schemes will affect the performance badly at this point due to the internal communication dependencies. Though there is potential for optimizations, e.g., deferring the dependence to communicate with a late arrival to the end of the communication scheme inside the collective, can hide some of the delay from this process. However, not much is known about the effects of the entering time on MPI collective performance at the moment; to our knowledge there is no benchmark specific on this topic so far.

III. Methodology and Benchmark description

To study MPI collective operations with respect to late arrivals, a micro benchmark suite was designed. The central point for the analysis therein is a global clock. The global clock is chosen to be the one of process with MPI rank 0. To obtain this global clock the micro benchmark suite determines the clock offsets between process zero and all other processes. Based on the global time, the benchmark performs then the following tasks for a collective benchmark:

- Measures start and end times of all involved MPI processes.
- Determines earliest start and latest end time over all involved MPI processes.

Each benchmark is run with different number of processes and if the collective exchanges data, different data sizes. Initially, a warm-up for the network, CPUs, etc. is performed running the benchmark several times before the real measurement is started. Then, the times for the real benchmark runs are recorded.

The design of the benchmark suite allows for easy extendibility and addition of new benchmarks. Table III lists all currently implemented MPI collective benchmarks. Within this work, results for blocking and non-blocking barrier, allreduce and alltoall operations are reported.

Table I. List of currently implemented MPI collective benchmarks.

benchmark	blocking	non-blocking
barrier	x	x
bcast	x	x
reduce	x	x
allreduce	x	x
alltoall	x	x

A. Clock offset determination

The local clocks of different processors across a distributed system report different times as they are not perfectly synchronized. They may even run at slightly different speeds [8] [9], which is not taken into account by the benchmark suite at the moment. This simplification is acceptable because the benchmarks run only for a relatively short time. Nevertheless, a verification step validating this assumption is performed at the end of the benchmark. It shows that there is no significant change in the time differences over the benchmark runs.

Hence, for the comparison of the times from the different clocks in the benchmark, the error between the clocks has to be taken into account. For this purpose, a simple linear approximation model is used [10]. Because of the short runtime only the clock offset σ—defined as the constant difference between the locally measured reference time t and the remote time t'—is of interest:

$$t' = t + \sigma \ . \tag{1}$$

The offset is determined at the start of the benchmark and checked at its end.

A modified ping pong experiment is used to determine the clock offset following Cristian's algorithm [11]: A root process sends a request to another process after determining his local clock value t_0. The other process answers with his current local time t'_r and the root process recognizes the time t_1 after receiving the response. We improve the accuracy by adding a second timer t_2 directly after taking t_1 allowing to determine the timer delay Δ, which is the time required to read out the clock itself. From this experiment the ping pong latency λ_p and timer delay Δ are obtained, see Figure 1.

To obtain the clock offset σ between local clock t and remote clock t' defined in (1), both messages in the ping-pong are assumed to have the same message latency. In this case, the remote time t'_r is at the mid of the ping pong. The clock offset is then given by

$$\sigma = t'_r - t_0 - (\lambda_p + \Delta)/2 \ , \tag{2}$$

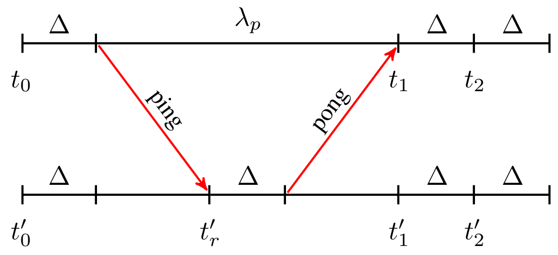

Figure 1. Modified ping pong experiment to determine ping pong latency λ_p, timer delay Δ and clock offset on the basis of the remote time t'_r, which is assumed to be taken at the mid of the ping pong.

where the timer delay Δ is obtained via

$$\Delta = t_2 - t_1 \ . \tag{3}$$

The accuracy of the obtained clock offset is increased by using the statistical value over 100 measurements.

To verify the correctness and to obtain an estimate for the error in the obtained clock offsets intra node times can be compared, which should not vary much. As can be seen in Table II, the clock offsets between rank 0 and all processes residing on one node are the same with a standard deviation of not more than $\pm 2\,\mu$s. In contrast, the clocks of different nodes vary by more than 10 ms between each other.

Table II. DETERMINED AVERAGE CLOCK OFFSET $\bar{\sigma}$ AND STANDARD DEVIATION σ_σ FOR A BENCHMARK RUN WITH 12 PROCESSES AND 4 PROCESSES PER NODE BASED ON A SET OF 100 MEASUREMENTS. (RESULTS OBTAINED ON HERMIT SYSTEM AT HLRS, SEE SECTION IV)

rank	$\bar{\sigma}$ [s]	σ_σ [s]
0	+0.000000	0.000000
1	+0.000000	0.000000
2	+0.000000	0.000000
3	+0.000000	0.000000
4	−0.017258	0.000002
5	−0.017258	0.000001
6	−0.017258	0.000001
7	−0.017258	0.000002
8	−0.011140	0.000002
9	−0.011140	0.000002
10	−0.011140	0.000002
11	−0.011140	0.000002

B. *Initial synchronization*

A synchronization of all processes is done at the beginning of each benchmark run. Two different synchronization methods are available: One using MPI barrier, and another using clock based synchronization [12]. The interface and implementation of the synchronization function already includes the application of a delay time, which we will describe in more detail in Section III-D.

a) Barrier based synchronization: The barrier based synchronization makes use of the `MPI_Barrier` to synchronize processes as shown in listing 1. The barrier based synchronization may not be perfect as can be seen from the trace in Figure 2 where the processes finish the barrier at slightly different times. The time difference between the processes at the exit of the barrier is there in the order of $4\,\mu$s for 32 processes over two nodes of hermit. The observed exit time pattern there shows the behaviour of a tree algorithm [13].

One idea to improve the barrier based synchronization is to measuring the time differences at its exit and to improve the sync using delays to compensate them afterwards. This succeeds only if the barrier algorithm works always in the same way, producing the same exit time pattern. But, testing the compensation approach with a delay granularity of $1\,\mu$s, resulted in even worse synchronization.

```
double synchronizeViaBarrier(MPI_Comm comm,
                                  double delaytime) {
        MPI_Barrier(comm);
        double r = delay(delaytime);
        return r;
}
```

Listing 1: Implementation of barrier based synchronisation.

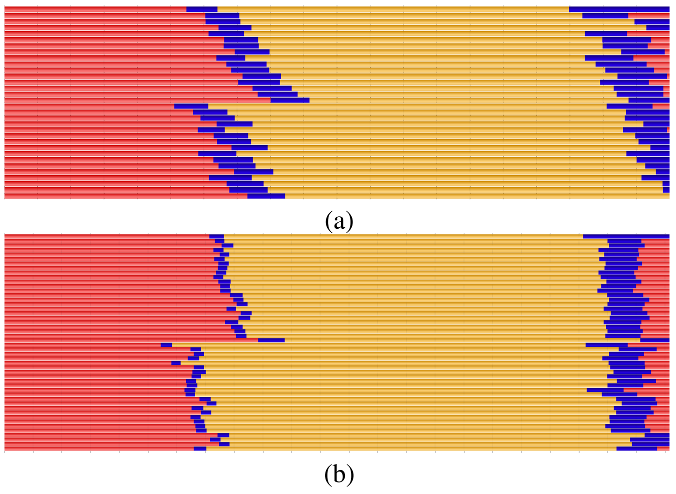

Figure 2. Time line trace images of synchronization barrier (red) before actual benchmark (orange) separated by timer calls (blue). Traces were obtained with Score-P and Vampir for (a) 32 PEs on 2 nodes of Hornet (Cray XE6) and (b) 48 PEs on 2 nodes of Hazel Hen (Cray XC40).

b) Clock based synchronisation: The clock based synchronisation allows a very precise time synchronization of events across processes [12]. It uses a global clock and local busy waiting until a defined start time point. The start time point is exchanged beforehand between all processes removing the dependence on sending messages over the network for the actual synchronisation step. Therefore, it is not affected by interconnect latencies, which may vary due to network contention. The accuracy of this method is limited by two things:

- the frequency and duration of clock read outs, which are required to monitor the current time
- the accuracy of the clock synchronisation, which is essential to define the global synchronisation time point.

With the current implementation of the clock based synchronisation in listing 2, using internally the POSIX `gettimeofday` for the `timer()`, the quality of the synchronization is already much better as the latest Barrier based synchronization front on Cray XC40 as can be seen in Figure 3.

```
double synchronizeViaClock(MPI_Comm comm,
                              double delaytime) {
        int syncRoot = 0;
        double synctime =
                   0.01 - clockOffsetsAvg[comm_rank];
        double endtime =
                   synctime + delaytime + timer();
        MPI_Bcast(&endtime, 1,
                  MPI_DOUBLE,
                  syncRoot,
                  comm);
        double r = timer() - endtime;
        while(r < 0) {
                r = timer() - endtime;
        }
        return r;
}
```

Listing 2: Implementation of clock based synchronisation.

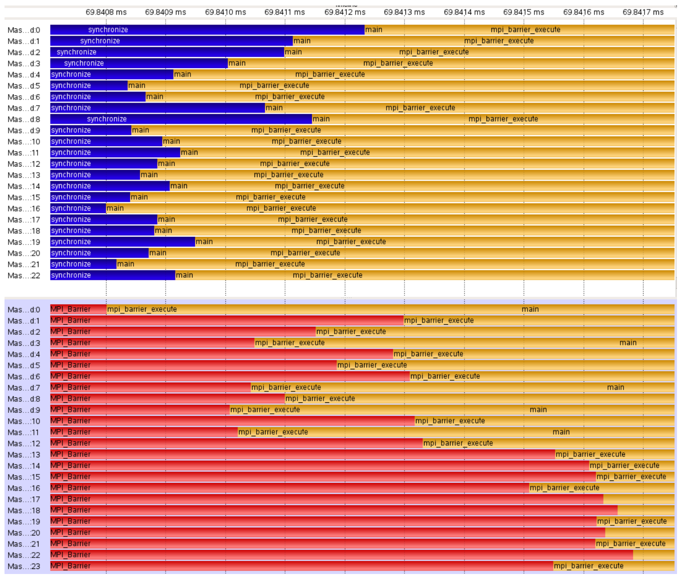

Figure 3. Time line trace image comparing the clock based (blue) with MPI barrier (red) synchronization before actual benchmark (orange). Traces were obtained with Score-P and Vampir for 24 PEs on 1 node of Hazel Hen (Cray XC40).

C. Collected data

For each measurement the process id as well as its start and end time are stored. The results can be output as ASCII text or in binary format using exchangeable data representation (XDR).

In the binary format time values are stored as double precision floating point values, which has 53 significant bits, corresponding to 15 decimal digits, which is more than sufficient for our purpose, as we collect times with no more than nano second resolution over a time frame of several minutes. The stored times are times corrected on the basis of the initially collected clock offsets.

If not mentioned otherwise explicitly, global times for the collective operations are reported, which is the time between the start time of the first process entering and the end time of the last process finishing the collective.

D. Delaying of single process

Load imbalances in programs cause some processes to enter collectives later than the rest. To study the influence of such late-arrivals on the overall collective time, the benchmark suite allows to delay processes by a given amount of time, see Figure 4.

The delay is implemented differently for the different synchronization methods:

For the barrier based synchronisation the delay is implemented indirectly by a separate delay function. The delay function busy loops for the specified time on the bases of the POSIX `gettimeofday` function, providing a microsecond accuracy.

The clock based synchronisation implements the delay directly shifting the internal start time point by the desired delay time. Therefore, the accuracy of the delay is the same as he the one for the synchronisation.

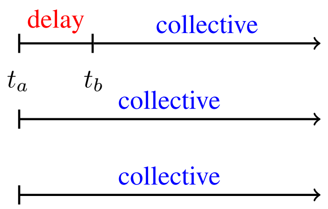

Figure 4. Processes are except one synchronized at time t_a and enter the collective. The one delayed process enters the collective at time $t_b = t_a + \delta$.

IV. Results

In the following, the influence of different delay times and different number of processes on the collective execution time is studied. Within the study blocking collectives and their non-blocking counterparts are compared side by side as they may be implemented in different ways. Here the call of the non-blocking MPI collectives are directly followed by an `MPI_Waitall` mimicing the blocking behaviour.

Two metrics are used within the examination of the results: The global collective time t_{global} and the overlap benefit b.

Global collective time: The global collective time is defined as the time between the earliest start time and the latest end time of the collective operation by any process:

$$t_{\text{global}} = \max(t_{\text{end}}) - \min(t_{\text{start}}) \, . \tag{4}$$

Delay overlap benefit: The overlap benefit metric gives a measure for the potential of internal overlap of the delay with communication in the collectives itself. The delay overlap benefit is defined as the fraction of overlapped execution time:

$$b = \frac{t_0 + \delta - t_\delta}{t_\delta} \, , \tag{5}$$

with t_0 being the collective time when the collective is called without any delayed processes and t_δ being the collective time when a process starts with delay δ.

A positive overlap benefit is found when there is overlap potential within the collective operation. A value of 1 indicates that the collective can hide a delay perfectly up to the time required for the undelayed collective. A value of zero can be observed when the delay just adds to the execution of a non delayed collective call without positive or negative side effects. A negative value means that the delay even results in additional cost compared to a synchronized collective, which is started after waiting delay time.

In the following, results for different collective operations on the Hermit and Hazel Hen systems at HLRS are reported. Hermit was a Cray XE6 system with 3552 dual-socket compute nodes and a total of 113 664 cores, which were connected via the Gemini 3D Torus network. Its successor Hazel Hen is a Cray XC40 system with 7712 dual-socket compute nodes and a total of 185 088 cores, which are connected in dragonfly topology via the Aries interconnect. The native Cray MPI implementations optimized for these system in combination with the GNU compiler were used for all tests.

All benchmarks were run during normal operation mode of the system so that other jobs on the system influenced the process placement and network usage. Benchmark runs were performed up to a maximum of 16 384 processes and were grouped into jobs with the same processor count. We report the found minimum values for the global times within 100 measurements. We use the minimum, as we are not interested in the average behaviour of the collectives but in the best we can get out of them on a system. This is responsible for some outlying data points, as we cannot guarantee to catch the best result even if multiple measurements were performed to reduce this effect. Obtaining the accurate minimum time for an operation under workload conditions is not always possible—especially for the longer benchmark runs using more processes, which get easily disturbed by other jobs.

For all measurements the MPI process with rank 0 was delayed. Most tree based algorithms—usually using rank 0 as tree root—should be badly affected by this choice, if they do not switch over using another process as the tree root.

A. Barrier

The first collective studied is the barrier. As the barrier is used for synchronization within the benchmark suite, the understanding of this operation is essential. While the time for `MPI_Barrier` is measured straightforward, the time for `MPI_Ibarrier` includes the time for the corresponding `MPI_Wait`.

A wide variety of different barrier algorithms exists [13]. Depending on the algorithm and the hardware support used within the implementation, different algorithms may profit differently. On the one hand, for example, the Central Counter barrier may hide the delay of a late arrival easily by concept, or the Binomial Spanning Tree Barrier could intelligently assign the delayed process to a node, which is involved in later communication steps. On the other hand, for example, the Dissemination Barrier requires a ring like communication in each step—which will not tolerate a late arrival.

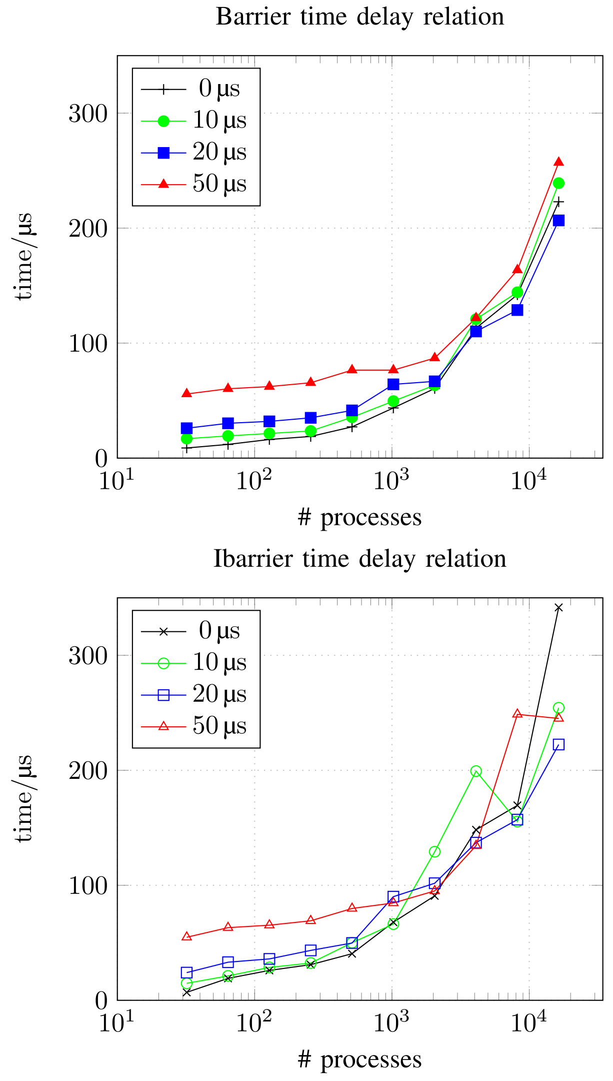

Figure 5. MPI_Barrier and MPI_Ibarrier global times for different delay times on Hornet.

The results in Figure 5 show a nearly logarithmic scaling of the blocking and non-blocking barrier operation up to approximately 2048 processes. For higher process counts, the behaviour seems to have a linear scaling. But we note here that a single cabinet of the Hermit system has 96 nodes with a total of 3072 cores. Jobs exceeding this number of processes are more likely to be spread around the system and therefore affected by network contention caused by other applications. So, finding the minimum time for the barrier operation with our benchmark may not have provided the correct result in this case.

The delay benefit as defined in (5) of the `MPI_Barrier` and `MPI_Ibarrier` for different delay times, where the delayed rank was always rank 0, is shown in Figure 6. As

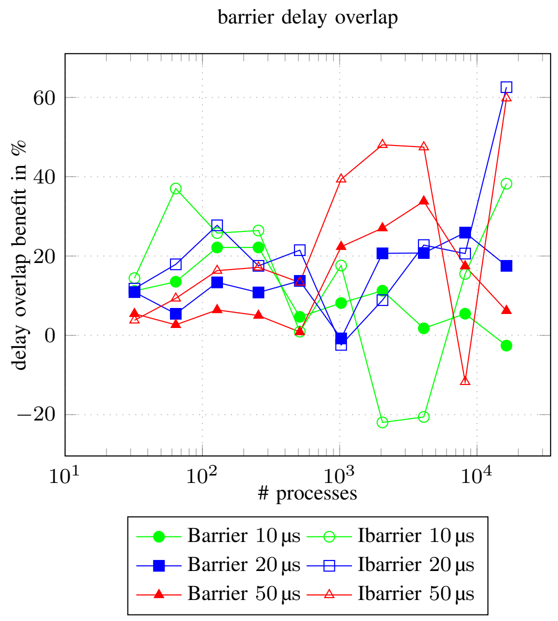

Figure 6. Delay benefit of the `MPI_Barrier` and `MPI_Ibarrier` as defined in (5) for different delay times on Hornet.

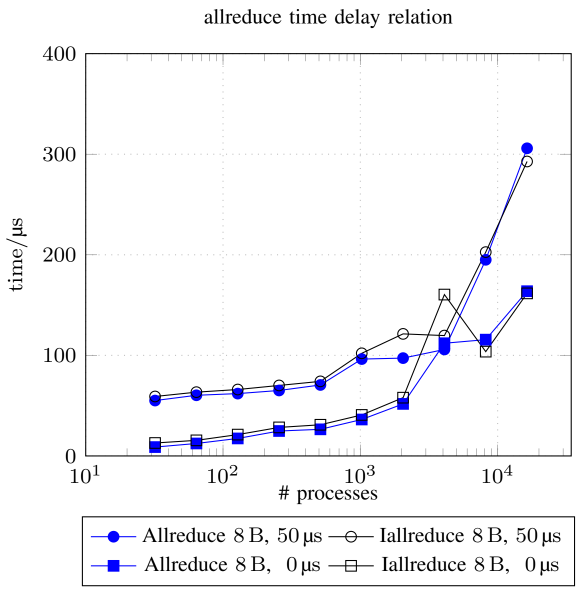

Figure 7. Hornet: `MPI_Allreduce` (circles) and `MPI_Iallreduce` (squares) global times for 8 B message size and a delay time of 50 µs (blue) together with perfectly synchronized reference data (black).

the benefit is mostly positive the implemented blocking and non-blocking barrier algorithm already seem to tolerate smaller delays. The non-blocking version `MPI_Ibarrier` seems to perform slightly better than the blocking variant here. Figure 6 shows an change in behaviour at 1024 processes: While at the beginning smaller delays have a higher overlap benefit, for more processes a larger benefit can be seen for longer delays. It is unclear if at this point an algorithm switch occurs within the MPI implementation.

B. Allreduce

An important collective to aggregate data of multiple processes into a single value is the allreduce operation. It may be used to determine, e.g., global energies in molecular simulations, time step lengths in finite element based programs or residues in linear solvers. While the time for `MPI_Allreduce` is measured straightforward, the time for `MPI_Iallreduce` includes the time for the corresponding `MPI_Wait`.

Again, the influence of delaying the process with rank 0 for different number of processes is studied. Results for 8 B messages and a delay of 50 µs are presented in Figure 7 for Hornet and in Figure 9 for Hazel Hen.

For Hornet, we see perfect logarithmic scaling up to 1024 processes, adding less than 5 µs when doubling the number of processes. For larger process counts the scaling is worse and adds up to 100 µs when doubling the number of processes. The behaviour for larger message sizes is similar. It is unclear how the synchronization barrier influences the behaviour, as we showed earlier that the processes do not exit from it perfectly at the same time. Also, the barrier itself

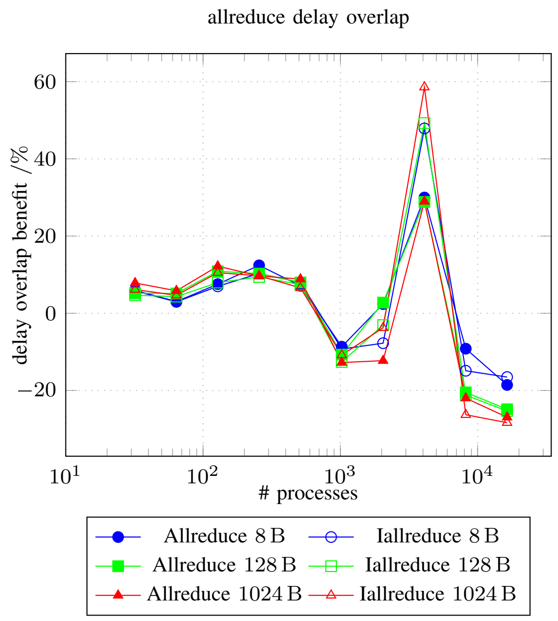

Figure 8. Hornet: Delay benefit of the allreduce collective for different message sizes at a delay time of 50 µs).

does not scale well for larger process counts according to our benchmark results, too, see Figures 2 and 5.

We have to mention a data outlier for the non-delayed Allreduce/Iallreduce benchmark runs with 4096 processes—which were grouped within one job. The job collecting these data was likely disturbed by other jobs and seems not to have been able to find an accurate value for the minimum collective time.

The delay benefit of the blocking and non-blocking allreduce operations presented in Figure 8 shows slight overlap for smaller number of processes. For more than 1024 processors the delay has a negative effect onto the overall performance. The message size does not have an influence on the delay benefit for the chosen values. The peak for 4096 processes is caused by too high values for the perfectly synchronized collectives time t_0.

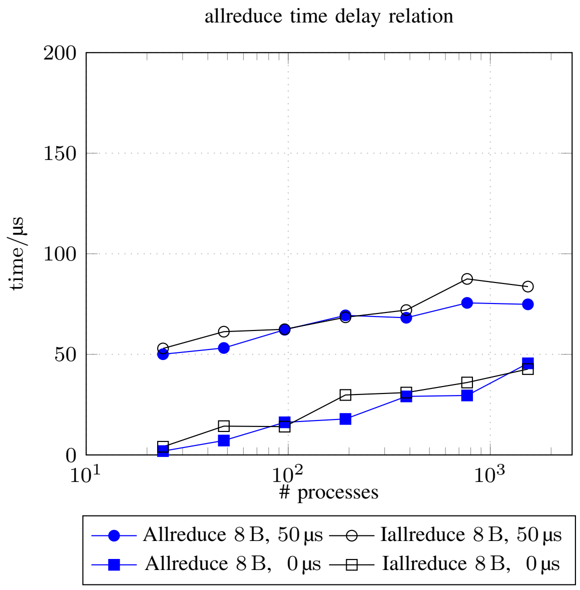

Figure 9. Hazel Hen: `MPI_Allreduce` (circles) and `MPI_Iallreduce` (squares) global times for 8 B message size and a delay time of 50 µs (blue) together with perfectly synchronized reference data (black).

The results for Hazel Hen shown good scaling up to over 1000 PEs in Figure 9. The delay benefit is positive in nearly all cases as can be seen from Figure 10.

C. Alltoall

The alltoall operation is another important collective pattern used in many parallel codes to distribute data in an application. It is the most time consuming collective operation but it may benefit a lot from intelligent algorithms, taking into account delayed processes.

The same measurements as that for the allreduce operation were performed. Results in Figure 11 show a nearly perfect linear scaling for the alltoall algorithm up to the maximum of 16 384 processes used during the benchmarks on Hornet. The message size has a strong influence on the execution time of the alltoall collective but does not affect the overall scaling behaviour.

The results for the delay benefit for the alltoall collective on Hornet, presented in Figure 12, show zero effect for small messages and an inconclusive behaviour for larger messages, which may be caused by the fact, that our benchmark does not find the minimum time as already mentioned before. So, we find slight decreases as well as huge gains in performance.

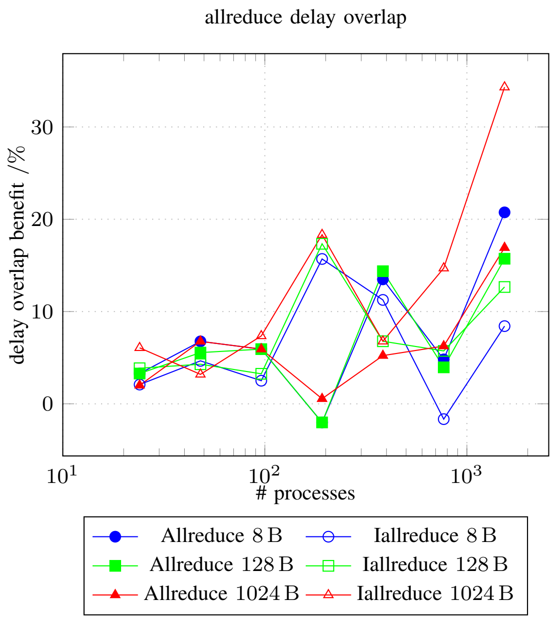

Figure 10. Hazel Hen: Delay benefit of the allreduce collective for different message sizes at a delay time of 50 µs).

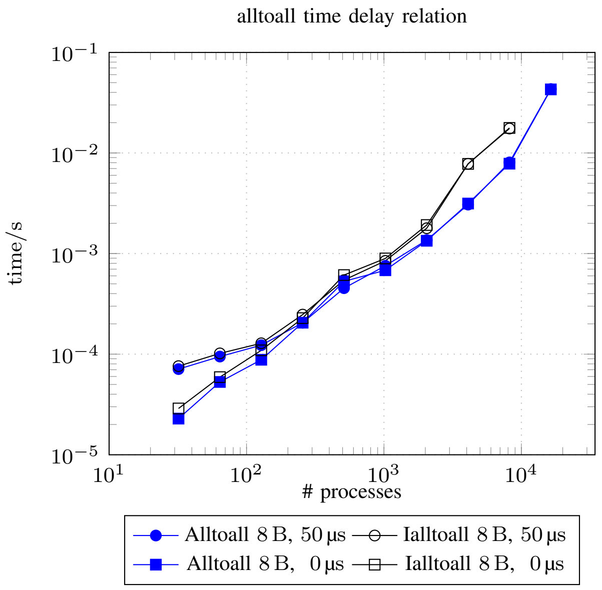

Figure 11. Hornet: `MPI_Alltoall` (circles) and `MPI_Ialltoall` (squares) global times for 8 B message size and a delay time of 50 µs (blue) together with perfectly synchronized reference data (black).

The allreduce results on Hazel Hen show that the delay benefit is nearly zero for small messages there as well. What is interesting here, is the fact that the delay benefit for the blocking versions is better than for their non blocking counterparts, as well as the execution times.

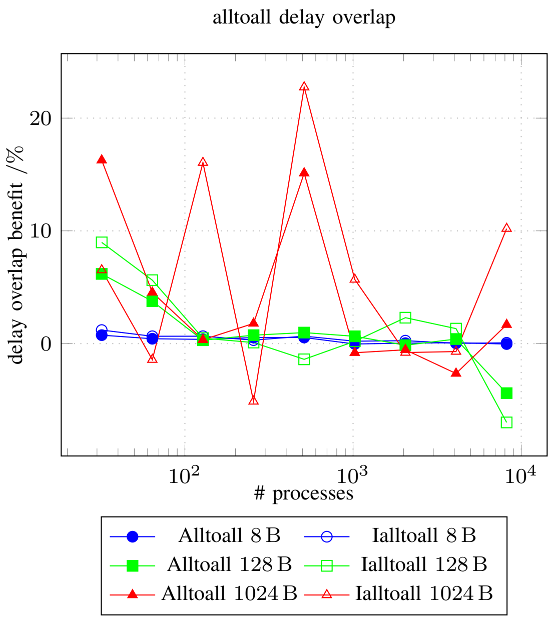

Figure 12. Hornet: Delay benefit for the alltoall collective for different message sizes at a delay time of 50 µs.

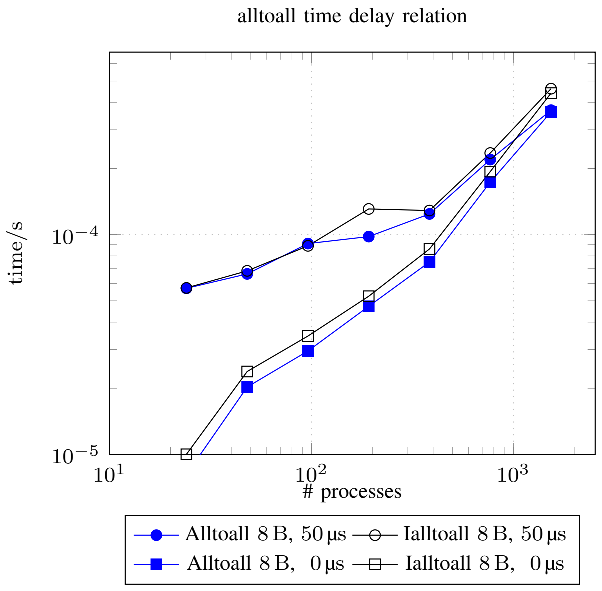

Figure 13. Hazel Hen: `MPI_Alltoall` (circles) and `MPI_Ialltoall` (squares) global times for 8 B message size and a delay time of 50 µs (blue) together with perfectly synchronized reference data (black).

D. Broadcast

The broadcast (bcast) operation is another collective pattern found frequently for any kind of initial or intermediate data distribution. For example, it is used to distribute configuration parameters from an input file, which should be not opened and read by all processes at the same time on today's HPC file systems. It is also an operation, which is used within optimized

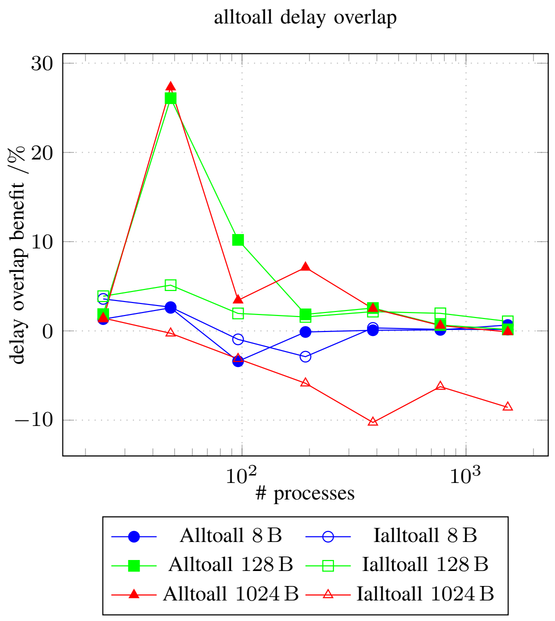

Figure 14. Hazel Hen: Delay benefit for the alltoall collective for different message sizes at a delay time of 50 µs.

versions of more complicated collective operations as part of the underlying communication patterns and algorithms.

The same measurements as before were performed. Results from the Hazel Hen system are presented in Figure 15.

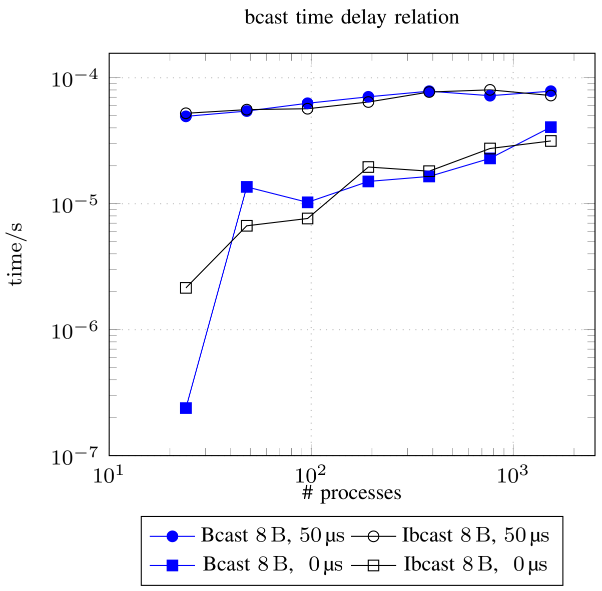

Figure 15. Hazel Hen: `MPI_Bcast` (circles) and `MPI_Ibcast` (squares) global times for 8 B message size and a delay time of 50 µs (blue) together with perfectly synchronized reference data (black).

The results for the delay benefit for the bcast collective, presented in Figure 16, show zero effect for small messages and an inconclusive behaviour for larger messages, which may be caused by the fact that our benchmark does not find the minimum time as already mentioned before. So we find slight decreases as well as huge gains in performance.

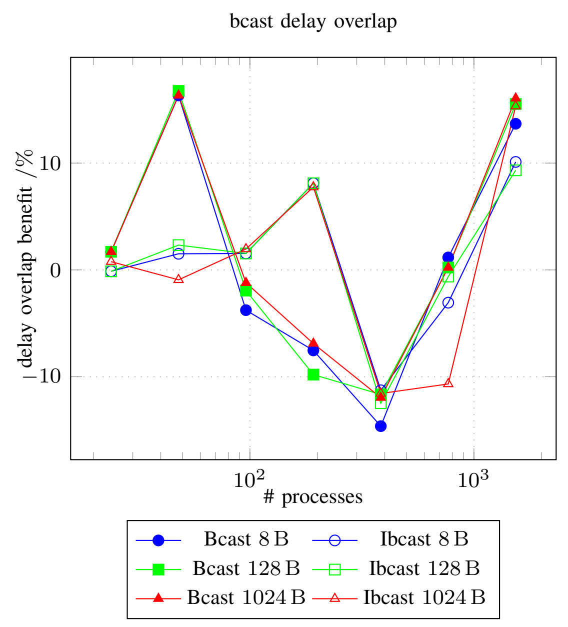

Figure 16. Hazel Hen: Delay benefit for the bcast collective for different message sizes at a delay time of 50 µs.

V. Conclusion and Outlook

In this paper, we have evaluated the impact of late arrivals, i.e., a delayed process, on the performance of the collective operations `MPI_(I)Barrier`, `MPI_(I)Allreduce` and `MPI_(I)Alltoall` on Cray XE6 and `MPI_(I)Bcast` on Cray XC40.

For the detail study we introduce a new benchmark, which allows to delay single MPI process out of a synchronized set. For the process synchronisation we show the effectiveness of a simple Barrier and compare it to the approach of a time based synchronisation scheme. Our findings show that the time based synchronization has better potential to achieve a flat synchronization then a simple barrier, which we find to show the structure of a tree based implementation.

For the evaluation of the results we make use of the global time and the newly defined delay overlap benefit metric. The first specifies the time span from the first process entering the collective to the finishing time of the last process leaving the collective. The overlap benefit metric is the fraction of delay time, which can be overlapped by the collective when comparing the collective times of a collective under a late arrival process with a collective executed starting with well synchronized processes.

The results show that blocking and non-blocking collective barriers can tolerate small delays, i.e., hide a part of the load imbalance within an application. The collectives `MPI_(I)Allreduce` tolerate small delays for up to 1024 processes but is badly affected for larger processes counts. The `MPI_(I)Alltoall` operations tolerate small delays well for up to 1024 processes and the delays have no negative effects for large processes counts. The alltoall operation can profit a lot in some cases for larger message sizes, while we see no negative effects for small messages. The broadcast operation on the Cray XC40 scales well, but shows an inconclusive behaviour when it comes to the tolerance of late arrivals.

We have shown that the overlap availability of non-blocking collectives and benefit of the overlapping depends on the type of the collective operations, size of the communicator and the amount of data to be communicated.

This work shows that the state of the art implementation of the relatively new MPI 3.0 non-blocking collective specification in Cray MPI is mostly head up or better than their blocking counterparts. We expect new algorithms and hardware with better overlapping capabilities and communication offloading support in the future. Our preliminary work in this area shows already some potential to hide small delays of single processes for barrier, allreduce and alltoall operations. The techniques for overlapping communication may also improve collective operations in the case of system noise.

Future studies about other important collectives are planed as well as detailed analysis of delaying other processes than rank 0. Studies are planed to evaluate other MPI library implementations. Here open source implementations can provide insights into the algorithms as well as the cross over points between them for different message sizes and process counts, allowing better understanding of the results.

Acknowledgement

This work has been supported by the European Community's FP7 programme through the project CRESTA (Grant Agreement no. 287703) and H2020 programme through the project Mont Blanc 3 (Grant Agreement no. 671697). We gratefully acknowledge funding by the German Research Foundation (DFG) through the German Priority Programme 1648 Software for Exascale Computing (SPPEXA). This work made use of computational resources provided by the High Performance Computing Center Stuttgart (Hermit). We thank P. Manninen (Cray Finnland) and R. W. Nash (EPCC) for valuable discussions and assistance.

References

[1] C. Niethammer, D. Khabi, H. Zhou, V. Marjanovic, and J. Gracia, "Impact of Late-Arrivals on MPI Collective Operations," in INFOCOMP 2015: The Fifth International Conference on Advanced Communications and Computation, 2015, pp. 60–65.

[2] T. Hoefler, T. Schneider, and A. Lumsdaine, "The Effect of Network Noise on Large-Scale Collective Communications," Parallel Processing Letters, Dec. 2009, pp. 573–593.

[3] K. B. Ferreira, P. G. Bridges, R. Brightwell, and K. T. Pedretti, "The impact of system design parameters on application noise sensitivity," Cluster Computing, vol. 16, no. 1, 2013, pp. 117–129.

[4] R. Thakur and R. Rabenseifner, "Optimization of collective communication operations in mpich," International Journal of High Performance Computing Applications, vol. 19, 2005, pp. 49–66.

[5] T. Hoefler, A. Lumsdaine, and W. Rehm, "Implementation and performance analysis of non-blocking collective operations for mpi," in Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, ser. SC '07. New York, NY, USA: ACM, 2007, pp. 52:1–52:10.

[6] H. Pritchard, D. Roweth, D. Henseler, and P. Cassella, "Leveraging the Cray Linux Environment Core Specialization Feature to Realize MPI, Asynchronous Progress on Cray XE Systems," in Proc. Cray User Group, 2012.

[7] R. Rabenseifner and A. E. Koniges, "Effective communication and file-i/o bandwidth benchmarks," in PVM/MPI, ser. Lecture Notes in Computer Science, Y. Cotronis and J. Dongarra, Eds., vol. 2131. Springer, 2001, pp. 24–35.

[8] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," Commun. ACM, vol. 21, no. 7, Jul. 1978, pp. 558–565.

[9] S. J. Murdoch, "Hot or not: Revealing hidden services by their clock skew," in 13th ACM Conference on Computer and Communications Security (CCS 2006). ACM Press, 2006, pp. 27–36.

[10] D. Becker, R. Rabenseifner, and F. Wolf, "Implications of non-constant clock drifts for the timestamps of concurrent events," in Cluster Computing, 2008 IEEE International Conference on, Sept 2008, pp. 59–68.

[11] F. Cristian, "Probabilistic clock synchronization," Distributed Computing, vol. 3, no. 3, 1989, pp. 146–158.

[12] S. Hunold and A. Carpen-Amarie, "MPI benchmarking revisited: Experimental design and reproducibility," CoRR, vol. abs/1505.07734, 2015. [Online]. Available: http://arxiv.org/abs/1505.07734

[13] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm, "A Survey of Barrier Algorithms for Coarse Grained Supercomputers," Chemnitzer Informatik Berichte, vol. 04, no. 03, Dec. 2004.