Impact of Late-Arrivals on MPI Collective Operations

Christoph Niethammer, Dmitry Khabi, Huan Zhou, Vladimir Marjanovic and José Gracica
High Performance Comuting Center, University of Stuttgart
Stuttgart, Germany
Email: {niethammer,khabi}@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 - but there is also a lot of room for future optimizations.**

Keywords–collectives; late-arrivals; benchmarking; MPI

I. Introduction

Collective operations strongly affect the performance of many Message Passing Interface (MPI) applications, as they involve large numbers, usually 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 delayed processes is not well understood at the moment. Earlier work showed that even small system noise can have a tremendous effect on the collective performance [1] [2]. So, though algorithms are optimized for large process counts [3], 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 [4], 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 [5].

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 point to potential for improving performance by solving the issue of late arrivals.

This work is structured as follows. Section II, 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 III, we define a metric to quantify the amount of tolerated delay. Then results for different, application relevant collective operations are presented and evaluated on basis of absolute times as well as the delay overlap metric.

II. Methodology and Benchmark description

To study collective operations with respect to late arrivals, a micro benchmark suite was designed. This requires the use of a global clock, which is chosen to be the one of process with MPI rank 0. For this purpose the micro benchmark suite determines the clock offsets between process zero and all other processes. Based on the global time, the benchmark performs the following tasks:

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

The design of the benchmark suite allows for easy extendibility and addition of new benchmarks. The following MPI collective operations are included at the moment: Barrier, Ibarrier, Bcast, Ibcast, Reduce, Ireduce, Allreduce, Iallreduce, Alltoall and

Ialltoall. Within this work, results for blocking and non-blocking barrier, allreduce and alltoall operations are reported. Each benchmark is run with different number of processes and different data sizes. Each benchmark is run initially for several times before the real measurement is performed, to warm up the network, CPUs, etc. Then, the times for the real benchmark runs are recorded.

A. Clock offset determination

The local clocks of different processors report different times as they are not perfectly synchronized. They may even run at slightly different speeds [6] [7], which we do not take into account. This simplification is acceptable because the benchmark runs only for a relatively short time and a verification shows, that there is no significant change in the time differences over it's runtime. To compare the measured times, the error between the clocks has to be taken into account [8]. For the collective benchmarks, we consider the clock offset σ defined as the constant difference between the locally measured time t and the time measured at the remote processor t' at the time point when the benchmark is started

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

A modified ping pong experiment is used to determine the clock offset following Cristian's algorithm [9]: A root process sends a request to another process, which answers with his current local time. We improve the accuracy by adding another timer allowing to determine the timer delay, which is the time required to obtain the current time itself. From this experiment the ping pong latency λ_p and timer delay Δ, are obtained, see Figure 1.

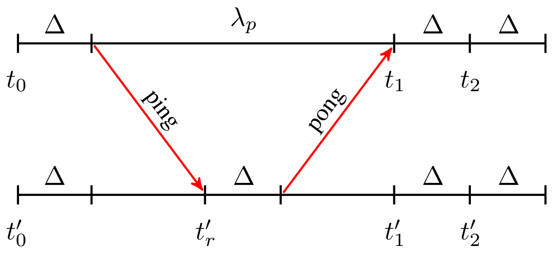

Figure 1. Modified ping pong experiment to determine ping pong latency λ_p, timer delay Δ and clock offset on the basis of the remote time t'_r, which is assumed to be taken at the mid of the ping pong.

To obtain the clock offset σ between local clock t and remote clock t' defined in (1), the time t'_r, which is assumed to be at the mid of the ping pong, is measured. The clock offset is then given by

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

where the timer delay Δ is obtained via

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

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 I, 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 100 measurements. In contrast, the clocks of different nodes vary by more than $10\,ms$ between each other.

Table I. 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 III)

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 using a barrier. The synchronization is not perfect as can be seen in Figure 2 and the processes finish the barrier at slightly different times. The time difference between the processes at the exit of the barrier is in the order of $4\,\mu s$ for 32 processes and the observed exit time pattern may be the result of a tree algorithm [10]. But so far, there is no better way of synchronization. Measuring the time differences and trying to improve the sync using delays with an accuracy of $1\,\mu s$ for faster processes, resulted in even worse synchronization.

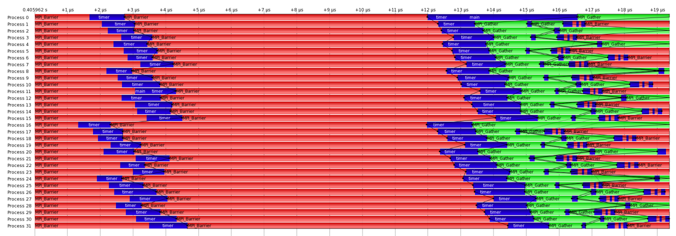

Figure 2. Vampirtrace image of synchronization barrier (red) before the benchmark is executed on Cray XE6 (32 PEs on 2 nodes). The processes leave the barrier in a time shifted front as can be seen by the timer calls enclosing the benchmarking loop (blue). Results are collected afterwards with an MPI_Gather operation (green).

For each measurement the process id (integer) as well as its start and end time (double) are stored, which requires $S = 20N$ Bytes of storage. The stored times are times corrected on the basis of the clock offset determined before. If not mentioned 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.

C. 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 one processes by a given amount of time, see Figure 3. The delay is implemented based on the POSIX `gettimeofday` function, providing a microsecond accuracy.

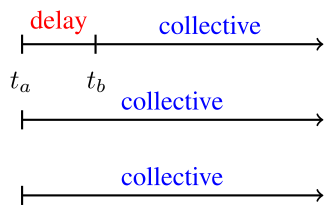

Figure 3. Processes are except one synchronized at time t_a and enter the collective. The one delayed process enters the collective at time $t_b = t_a + \delta$.

III. RESULTS

The influence of different delay times and the number of processes is studied. Blocking collectives and their non-blocking counterparts are compared side by side. Beside the actual collective times the benefit b of internal overlap of the delay with communication within the collectives itself will be examined:

$$b = \frac{t_0 + \delta - t_\delta}{t_\delta}\,, \tag{4}$$

with t_0 being the collective time for no delay and t_δ the collective time for delay δ. A positive benefit is found when there is overlap potential within the collective operation. 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 system at HLRS are reported. Hermit is a Cray XE6 system with 3552 dual-socket compute nodes and a total of $113\,664$ cores, which are connected via the Gemini 3D Torus network. The native Cray MPI implementation optimized for this system in combination with the GNU compiler was 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. This is responsible for some outlying data points, 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 [10]. 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.

The results in Figure 4 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 (4) of the `MPI_Barrier` and `MPI_Ibarrier` for different delay times, where the delayed rank was always rank 0, is shown in Figure 5. As 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 5 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 $8\,\mathrm{B}$ messages and a delay of $50\,\mu\mathrm{s}$ are presented in Figure 6. We see perfect logarithmic scaling up to 1024 processes, adding less than $5\,\mu\mathrm{s}$ when doubling the number of processes. For larger process counts the scaling is worse and adds up to $100\,\mu\mathrm{s}$ when doubling the number of processes. The behaviour for larger

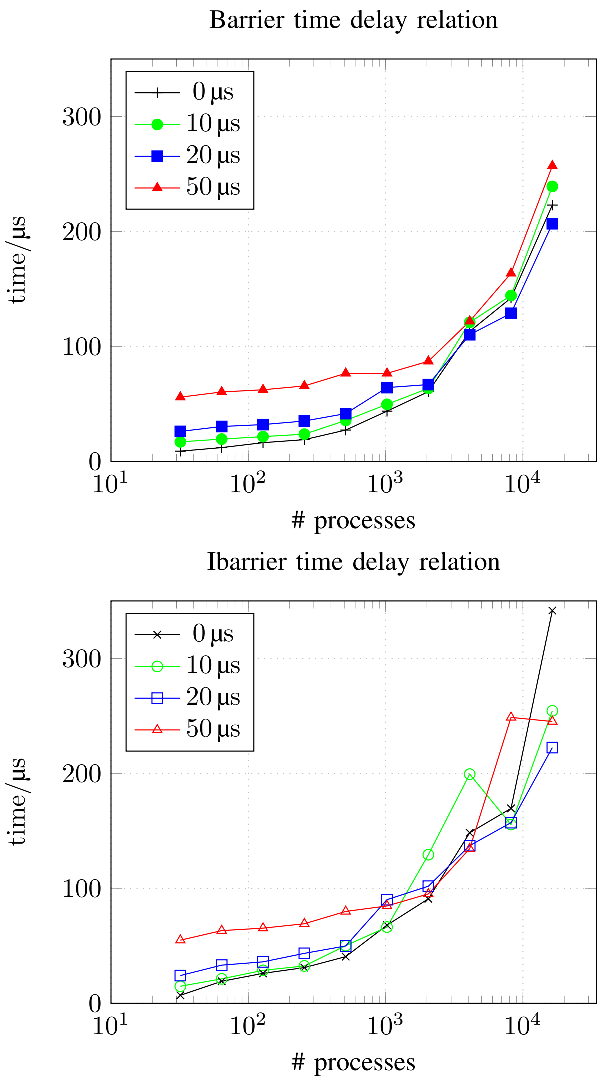

Figure 4. MPI_Barrier and MPI_Ibarrier global times for different delay times.

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 and it does not scale well for larger process counts according to our benchmark results, too, see Figures 2 and 4.

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 7 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

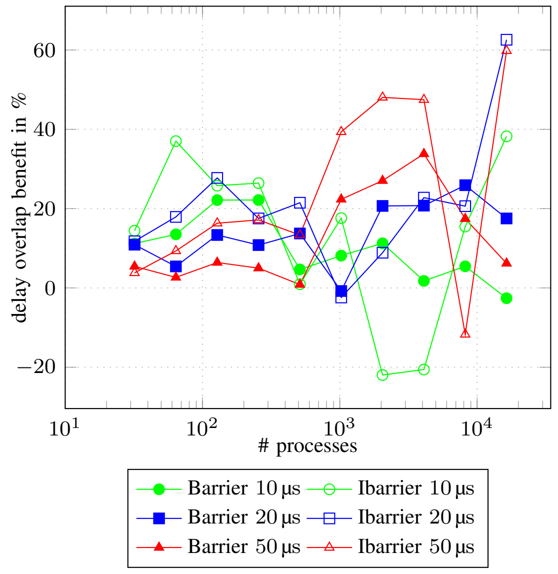

Figure 5. Delay benefit of the `MPI_Bbarrier` and `MPI_Ibarrier` as defined in (4) for different delay times.

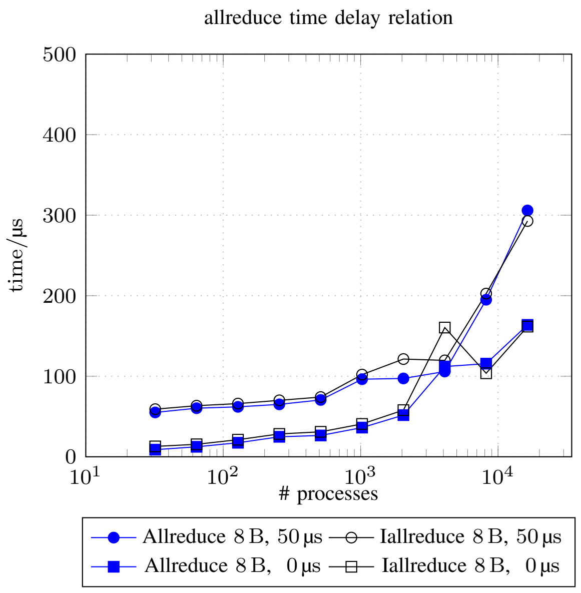

Figure 6. `MPI_Allreduce` (circles) and `MPI_Iallreduce` (squares) global times for 8 B message size and a delay time of 50 µs (blue) together with perfectly synchronized reference data (black).

benefit for the chosen values. The peak for 4096 processes is caused by too high values for the perfectly synchronized collectives time t_0.

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

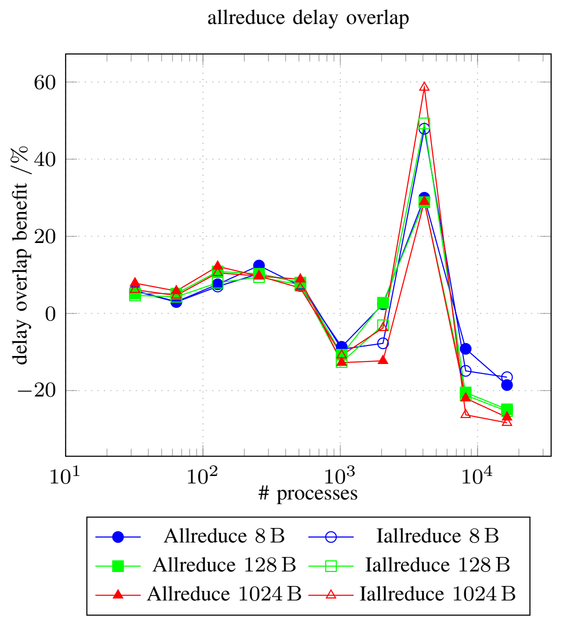

Figure 7. Delay benefit of the allreduce collective for different message sizes at a delay time of 50 µs).

delayed processes.

The same meassurements as that for the allreduce operation were performed. Results in Figure 8 show a nearly perfect linear scaling for the alltoall algorithm up to the maximum of 16 384 processes used during the benchmarks. The message size has a strong influence on the execution time of the alltoall collective but does not affect the overall scaling behaviour.

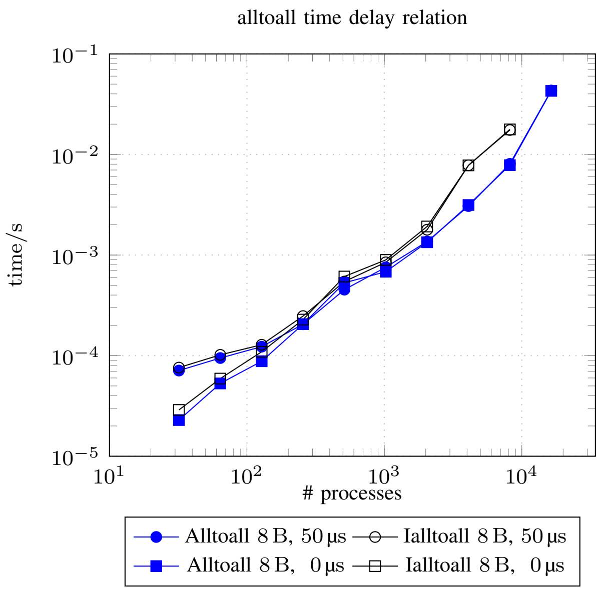

Figure 8. MPI_Alltoall (circles) and MPI_Ialltoall (squares) global times for 8 B message size and a delay time of 50 µs (blue) together with perfectly synchronized reference data (black).

The results for the delay benefit for the alltoall collective, presented in Figure 9, 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.

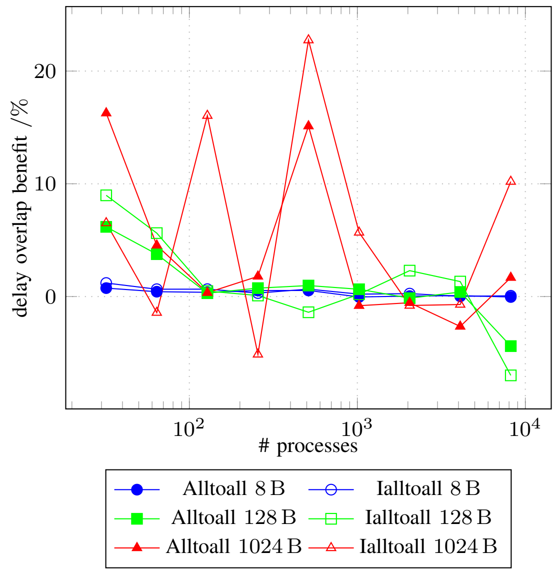

Figure 9. Delay benefit for the alltoall collective for different message sizes at a delay time of 50 µs.

IV. 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. 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.

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.

For the future studies about other important collectives like bcast 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 CRESTA project that has received funding from the European Community's Seventh Framework Programme (ICT-2011.9.13) under Grant Agreement no. 287703. 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] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

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

[8] 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.

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

[10] 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.