An Investigation into the Performance of Reduction Algorithms under Load Imbalance

Petar Marendić[1,2], Jan Lemeire[1,2], Tom Haber[3],
Dean Vučinić[1,2], and Peter Schelkens[1,2]

[1] Vrije Universiteit Brussel (VUB), ETRO Dept.,
Pleinlaan 2, B-1050 Brussels, Belgium
petar.marendic@vub.ac.be
[2] Interdisciplinary Institute for Broadband Technology (IBBT), FMI Dept., Gaston Crommenlaan 8 (box 102), B-9050 Ghent
[3] EDM, UHasselt, Diepenbeek
tom.haber@uhasselt.ac.be

Abstract. Today, most reduction algorithms are optimized for balanced workloads; they assume all processes will start the reduction at about the same time. However, in practice this is not always the case and significant load imbalances may occur and affect the performance of said algorithms. In this paper we investigate the impact of such imbalances on the most commonly employed reduction algorithms and propose a new algorithm specifically adapted to the presented context. Firstly, we analyze the optimistic case where we have a priori knowledge of all imbalances and propose a near-optimal solution. In the general case, where we do not have any foreknowledge of the imbalances, we propose a dynamically rebalanced tree reduction algorithm. We show experimentally that this algorithm performs better than the default OpenMPI and MVAPICH2 implementations.

Keywords: MPI, imbalance, collective, reduction, process skew, benchmarking.

1 Introduction

The reduction algorithm - *extracting a global feature from distributed data* such as the sum of all values - is a common and important communication operation. However, it has two downsides which degrade the performance of a parallel program. Firstly, all of the reduction algorithms scale superlinearly as a function of the number of processors, as shown later on in the text. Secondly, any reduction operation breaks the independence of process execution, as it requires a global process synchronization. Unless of course if the reduction could be performed in the background, i.e. asynchronously - a case we will not consider here. Reduction algorithms are vulnerable to load imbalances in the sense that if one process is delayed before starting the reduction, the execution of part of the reduction will also be delayed. One can however change the order in which process subresults

C. Kaklamanis et al. (Eds.): Euro-Par 2012, LNCS 7484, pp. 439–450, 2012.

are combined so that this concomitant delay is significantly reduced. In the extreme case where the imbalance dwarfs the combination times, this delay can be effectively eliminated.

1.1 Performance Cost Model

For analytical evaluation and comparison of various reduction algorithms we will employ a simple flat model as defined by [1] wherein there are p participating processes and each process has an input vector size of n bytes. We denote the local computation cost for one binary operation on two vector bytes as $\gamma[sB^{-1}]$. Communication time is modeled as $\alpha + n\beta$, where α is per message latency and $\beta[sB^{-1}]$ per byte transfer time.

We further assume that any process can send and receive one message at the same time, so that p parallel processes can send p messages in parallel.

The next section discusses the state-of-the art on reduction algorithms and the effect of load imbalances. In section 3 we present a static load balancing algorithm, while in section 4 we propose a new dynamic load balancing algorithm. Section 5 presents the experimental results.

2 Reduction

By definition, a reduction operation combines elements of a data vector residing on each of the participating processes by application of a specified reduction operation (e.g. maximum or sum), and returns the combined values in the output vector of a distinguished process (the root).

All reduction operators are required to be associative, but not necessarily commutative. However it is always beneficial to know whether a particular operator is commutative as there are faster ways of performing a reduction in that case.

One interesting case where a non-commutative operator arises is in the image compositing step of a distributed raytracing algorithm. In such an algorithm global data is distributed across processes and each process generates an image of its share of that data. To produce the final image, a composition needs to be performed on the produced images. This composition is a complex reduction step using the so-called 'over' operator and needs to happen in the correct back-to-front order.

The reduction that we've thus far been talking about is actually known as all-to-one reduction [2] since the end result is sent to one distinguished process. Variants of the reduction operation are the allreduce and reduce-scatter

2.1 Related Work

The simplest implementation of an all-to-one reduce is to have all processes send their local result to the root and the root combine these subresults in the next step. This approach is known as *Linear Reduction Algorithm*. It usually results in a bottleneck at the root process. Using our cost model, the complexity of this algorithm can be expressed as:

$$T(n,p) = (p-1)(\alpha + n\beta + n\gamma) \tag{1}$$

One straightforward way to eliminate this bottleneck is to employ a divide and conquer strategy that will order all participating processes in a binary tree and where at each step half of the processes will finish their work. This *Binary Tree Reduction* algorithm is efficient for small message sizes but suffers from suboptimal load balance, as the execution time is:

$$T(n,p) = \lceil \log_2 p \rceil (\alpha + n\beta + n\gamma) \tag{2}$$

Another variation on this idea is *Binomial Tree Reduction* where a binomial tree structure is used instead of a binary tree. This structure has the advantage of producing less contention on root nodes over that of the binary tree.

Other well known algorithms are *Direct-Send* and *Scatter-Gather*. In direct-send, every process is responsible for $\frac{1}{p}$th of the data vector and scatters the remaining chunks to their corresponding processes. In the second stage, once all processes have reduced the chunks they had received, a gather operation is performed on these subresults to form the result vector at the root process. This approach will result with maximal utilization of active computational resources, and with only a single communication step. However, it will also generate $p \times (p-1)$ messages to be exchanged among all participating processes. In a communication network where each of the participating processes are connected by network links, this will likely generate link contention as multiple processes will simultaneously be sending messages to the same process [3]. The execution time for direct-send is:

$$T(n,p) = p \times (p-1)(\alpha + \frac{n}{p}\beta) + n\gamma \tag{3}$$

It should be noted that this only states the time to perform a Reduce-Scatter, i.e. having the result vector scattered across participating processes. To implement an All-to-One reduction we need to follow up the reduce-scatter step by a gather to the root, which is typically performed with a binomial tree algorithm. Reduce-scatter can be also be performed by other well known methods, such as *Binary Swap* or *Radix-k* algorithms. Another well-performing algorithm of this type is Rabenseifner's algorithm which was shown to perform well with longer message sizes [4,1,5].

$$T(n,p) = 2\log_2 p\alpha + n\beta + (1 - \frac{1}{p})(n\beta + n\gamma),\ \textit{where}\ p = 2^x, x \in \mathbb{N} \tag{4}$$

This algorithm is considerably more efficient than the binary tree reduction when the complexity of the reduction operation is significant.

As far as we know, no work has been done on analyzing and optimizing reduction algorithms under load imbalances. We will show in the following chapter that this leaves many real world scenarios unaccounted for.

2.2 Load Imbalances

We can identify three sources of imbalances:

- (Type 1) imbalances in the phase that precedes reduction
- (Type 2) imbalances in the amount of data that is sent at each step
- (Type 3) imbalances in the completion time of the combination operation.

The distributed raytracing algorithm we previously mentioned is a nice example of type 1, 2 and 3 imbalance occurrences. As it is typical for applications of this sort to generate images in the multi megapixel range, compression schemes are often employed to reduce the amount of data to be sent across the network. The time to combine these images using the *over operator* is a linear function of their size, where the effective size of the image is measured in non-black, that is relevant only pixels. Since this size varies across processes, the time to combine such images will vary as well.

3 Static Load Balancing under Perfect Knowledge

Here we assume perfect knowledge of the load imbalances and the time reduction phases will be finished. We analyze which reduction scheme gives the minimal completion time. For the communication and combination step we assume one of the following two performance models. The one of Fig. 1 is based on three parameters σ, τ and ψ, while the one of Fig. 2 is a simplification in which $\tau = 0$. We assume that the three parameters are constant during the total reduction. The parameters incorporate α, β and γ discussed before. Parameter σ denotes the time which is consumed on the sending process. Parameter τ denotes the time in which the sending process has already started the communication, but the receiving process does not yet have to participate, in the sense that no cycles are consumed. This happens if the message has not arrived yet or part of the receiving is performed in the background. These cycles can be used for other computations, so during τ, the receiving process might be busy with other things. We then assume that the receiving process is ready after τ. After that, it consumed ψ cycles to finish the communication and combination phase. When the receiving process would not be ready after τ, the phase will start when ready and still consume ψ. Not that ψ includes the receiving and combination phase.

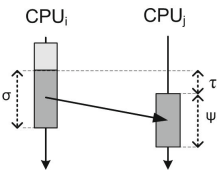

Fig. 1. Performance model of a communication and combination phase

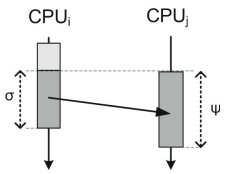

Fig. 2. Simplified performance model of a communication and combination phase

Under these assumptions, we propose the following algorithm. The algorithm is executed by each process when it starts the reduction.

Algorithm 1. The static optimized reduction algorithm

While step S2 has not been performed, do the following

C check if another process has sent its subresults to you

S1 if so, receive it and accumulate it with own subresult
S2 if not, send own subresult to the first process that will be executing check C

Under the assumption of 'perfect knowledge' we know during step S2 which process will be first to be ready to receive a message. Secondly, in step C we assume that we can test without cost that there is a message on the way. We neglect the fact that the test for an incoming message (a 'MPI_Probe call') could give a negative answer while process just has posted a message which is not yet detectable by the receiving process.

The algorithm gives the optimal reduction when using the simplified performance model.

Lemma 1. *The static optimized reduction algorithm (Alg. 1) gives the minimal completion time under the simplified performance model.*

Proof. At S2, a process has to decide to whom sending its subresults. By sending it to the first process ready to receive (step C), say process 2 at t_2, the receiving and combination will be ready first, at $t_2 + \psi$. By sending it to another process, ready at t_3 with $t_2 < t_3$, the merge step will only be ready later, at $t_3 + \psi$. This would not give an advantage. Process 2 could start sending its subresults at t_2 to process 4, which will finish at $t_4 + \psi$. But this is not faster than any other process that would merge with process 4. The earliest that this can finish is $t_4 + \psi$. In this way we have proven that no other choice of receiving process can complete the reduction earlier.

Alg. 1 is, however, not always optimal for the first communication model. In some very specific cases, an alternative merge order gives a better completion time. Consider the case shown in Fig. 3. P1 sends its data to P2 (the first one to finish next) and P3 merges with P4. A better merge order is shown in Fig. 4. Here P1 communicates with P3 instead and P2 with P4. The first message (P1 to P3) arrives later but the second one (P2 to P4) arrives earlier than in the first scheme. Due to the configuration of the imbalances in P5, P6, P7 and P8, this gives rise to a merge order which finishes τ earlier.

Hence, the given algorithm is suboptimal. Nonetheless, it will be optimal in most cases. Only in exceptional fine-tuned cases, alternative schemes will exist. Moreover, the difference with the optimal completion time will be small because we expect τ to be quite small. Concluding, in most cases, the algorithm will be optimal. In the exceptional, suboptimal cases the algorithm will be approximately optimal.

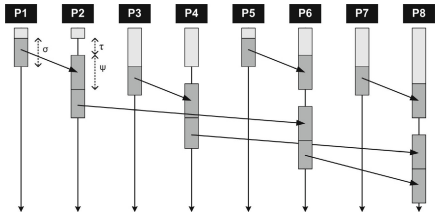

Fig. 3. Case in which the static load balancing algorithm is not optimal

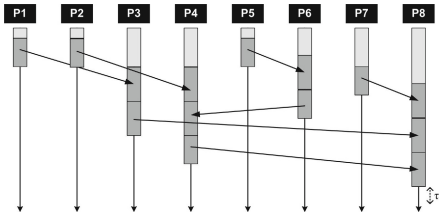

Fig. 4. Alternative merge order which completes faster than the static load balancing algorithm

It must be stressed that this algorithm is impractical, since the knowledge on when processes will be ready will not be present (it is difficult to predict and would in general lead to too much overhead to communicate). On the other hand, the algorithm gives a hint of how the optimal reduction would have been. Every solution can be compared to it.

4 Dynamic Load Balancing

Our initial idea was to take a regular binary tree reduction algorithm and augment it by installing a timeout period at each node of the tree. Should at any time a node time out waiting on data from its children, it would delegate these busy child nodes to its parent, reducing the number of steps their messages will eventually have to take on their way to the root. This process would continue until the root node received contribution from all nodes. Benchmarking however showed that this algorithm lacked robustness, as it was hard to pick a proper timeout value for varying vector sizes, process numbers and operator complexities.

We therefore turned our attention to a more deterministic algorithm that although tree-based was capable of dynamically reconfiguring its structure to minimize the effect an imbalance might have. The algorithm allows neighbours that are ready to start combining their subresults. The processes are ordered in a linear sequence and will send their local subresult to their right neighbour when finished, as described by Algorithm 2. Since the right neighbour might already have terminated, first a handshake is performed by sending a *completion*

message and waiting for the *handshake* message. Once a process is finished, its left neighbour should be redirected to its 'current' right neighbour. This is illustrated by the example run shown in Fig. 5.

Algorithm 2. Local reduction algorithm

S1 Initialize left and right neighbours according to the neighbours in the predefined linear ordering of the processes. The processes at the head or tail do not have left or right neighbour respectively.
S2 Send *completion* message to right neighbour.
S3 Wait for incoming messages.

S3.1 On receipt of *completion* message, initiate *handshake*.
S3.2 On receipt of *redirect* message, change right to new node.

S4 Complete *handshake*, exchange *data* and perform reduction. Change left to sender's left neighbour.
S5 If data was received goto 3
S6 Wait for message from right neighbour and *redirect* to left.

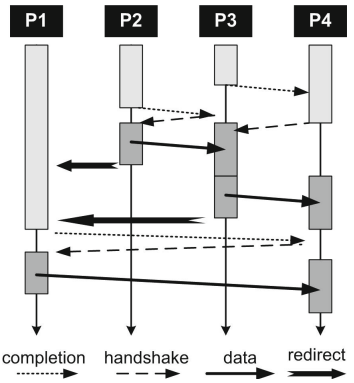

Fig. 5. Example run of the local reduction algorithm

5 Experimental Results

We devised an experiment with the primary purpose of ordering several well known algorithms in terms of their performance under various conditions of load imbalance. Our benchmarking scheme was as follows: before running the actual timed benchmark for a given algorithm, we perform a single warm up run; second, we synchronize the participating processes with a single call to MPI::Barrier; third, we sleep each process for *delay* milliseconds, where *delay* is a time offset that we distribute individually across participating processes using one of the schemes enumerated below; then, we run the algorithm k times for

each predefined operator, where k is a parameter to our benchmark program, random shuffling the data vector between each iteration. Finally, we repeat this process r times, each time random generating a new data vector and a new time offset for each participating process.

5.1 Completion Time

In ideal conditions, where no load imbalances and process skew are present, the completion time of a reduction operation should typically be measured from its initiation till its termination at the root process, as this best reflects the time a given user would have to wait before the results of a reduction operation become available to him.

However, in real-life applications it is rarely the case that all participating processes begin the reduction at the same time, even when they have been explicitly synchronized with a call to MPI::Barrier [6]. To compound matters, we explicitly introduce process skew ensuring that participating processes will be initiating and completing their share of the reduction operation at different time instances.

With this in mind, and having resolved to report only a single number as the elapsed time of a reduction operation, several different schemes of reducing the initiation and termination times at each process present themselves. [7] and [6] enumerate the following approaches

T_1 the time needed at a designated process (termination - initiation time at root)
T_2 maximum of the times on all participating processes
T_3 the time between the initiation of the earliest process and the termination of the last
T_4 the average time of all processes
T_5 minimimum of the times on all participating processes

One can however also take into account the imbalance time of a given process, and treat both this time and the reduction time as one unit of interest, as would be the case in image rendering where the imbalance time is the time required to generate an image of the local data and the reduction time, the time to perform image compositing that results with an image of global data. Thus we decided to report the mean time T_1 plus the imbalance time at root process, across r iterations.

5.2 Benchmark Parameters

To study the performance of reduction algorithms we developed a test suite that can create workloads controlled by the following parameters:

- Statistical imbalance distributions:
  - a Gaussian distribution with mean m and standard deviation s.
  - a Gamma distribution with mean $m{=}\theta.k$, where θ is the scale parameter and k the shape parameter. It is frequently used as a probability model for waiting times.

- data vectors sizes. No imbalances were generated (Type 2).
- reduction operator complexity. No imbalances were generated (Type 3), completion time was a function of vector size.

We decided to take into consideration imbalances of type 1 only, as it is our opinion that they are most representative of real life scenarios.

We included the following algorithms into our analysis:

1. Binary Tree Reduction
2. OpenMPI's and MPICH's default implementation
3. All-to-All followed by local reductions and final gather to root
4. Our local reduction algorithm

The experiments were performed on a cluster machine available in the Intel Exascience lab, Leuven. It is a 32 node cluster with 12 cores per node (2 x Xeon X5660 processor with $\gamma = 1.79\ 10^{-10}$s defined as $2/R_{peak}$ with R_{peak} the theoretical maximum floating point rate of the processing cores) and QDR Infiniband interconnect (measured MPI bandwidth of $\sim$ 3500MB/s, $\alpha = 3.13$ μs, $\beta = 0.00263\mu$s)

5.3 Performance of Default Implementation

Our tests have shown that the default implementation of the reduction algorithm under MVAPICH2, when tested without any load imbalances, is consistently slower than our All-to-All reduce implementation. Even though we don't report the timings here, we have run the same battery of tests on OpenMPI as well, but the ranking of the tested algorithms was unchanged. In addition, we checked the performance of the default implemented Reduce-Scatter algorithm and have confirmed that it too is faster than the default Reduce algorithm. This leads us to believe that the default implementation is in fact a binomial tree reduction algorithm that performs well for small vector sizes only. This is a rather surprising revelation considering that significantly better algorithms have been published more than 5 years ago [4,1,5].

5.4 Impact of Data Size

For small vector sizes, the default implementation was regularly the fastest, scaling very well with increasing number of processors which is indicative of binomial tree reduction. On the other hand, for big and very big vectors the All-to-All (and Reduce-Scatter) algorithms outperform everyone else thanks to their linear γ factor scaling (Eq. 4). We should point out that All-to-All reduce can only be applied if the data vector can be sliced - something that is not always feasible with custom user defined data. In such case the only recourse is to revert to the binomial tree algorithm. Our benchmarks have shown that for reasonably big vectors the local reduce algorithm is of approximately the same performance as the default implementation and considering its superior performance under load imbalance we can confidently state that in this case it is the more robust algorithm of the two.

5.5 Impact of Reduction Operator's Complexity

Reading the relevant literature, one cannot escape the sentiment that little research has been performed in evaluating the performance of reduction algorithms with operators of varying complexity. We decided therefore to include into our tests two operators: std::plus and a special operator that is two orders of magnitude slower. Additionally, for sake of generality we assumed that both of these operators are non-commutative. The significant disparity in complexity of these two operators made it immediately apparent how inadequate the default implementation is, as it was completely outperformed by All-to-All (and Reduce-Scatter) implementations with execution times up to 6 times slower and with significantly worse scaling as visible in Fig. 7. It was however consistently faster than our local reduce algorithm in test runs without imbalance.

5.6 Impact of Load Imbalances

To test the impact of load imbalances, we identified two interesting cases:

1. there is a single slow process.
2. imbalances are distributed according to a gamma distribution for which k=2 and θ=0.5, where the 90th percentile was 2.7 the mean imbalance.

In the first case where one of the processes is experiencing a slowdown, our local reduction algorithm proves itself as the best performer, often exhibiting flat scaling due to its ability to hide communication and computation overheads behind the incurred imbalance (see Fig. 8). The improvement we were able to achieve with our algorithm is dependent on the ratio of imbalance time and the time to reduce two vectors as is visible from Table 1. However, when the imbalances are distributed according to a gamma distribution law the local reduction algorithm only remains competitive up to and including vector size of 4MB (see Fig. 9). For a 40MB vector, the All-to-All implementation was the fastest.

Table 1. Speedup obviously depends on the imbalance time. Here, the time to reduce two vectors was 4 ms. All runtimes are reported in seconds.

128 processors with a 1024000 elements vector					
	Imbalances in ms				
Algorithm	10	20	30	40	60
LocalReduce	0.0466108	0.0458913	0.0466785	0.050597	0.0695058
Default	0.0502286	0.0599159	0.0696198	0.079698	0.0996538
Speedup	1.08	1.32	1.49	1.56	1.43

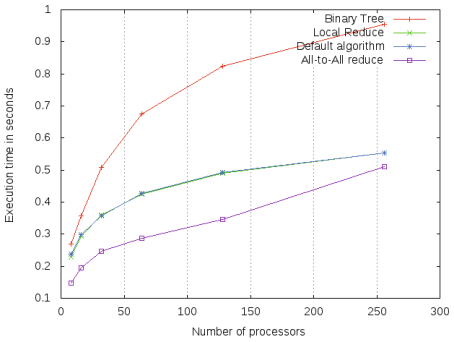

Fig. 6. Performance result for a 2^{20} elements vector using operator std::plus

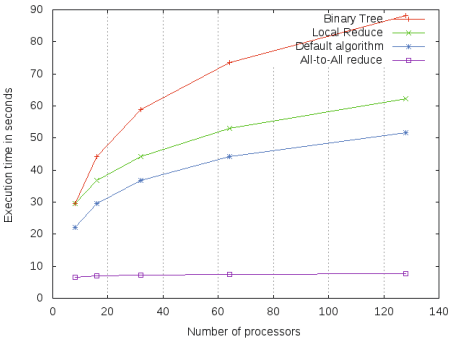

Fig. 7. Performance result for a 2^{20} elements vector using special slow operator

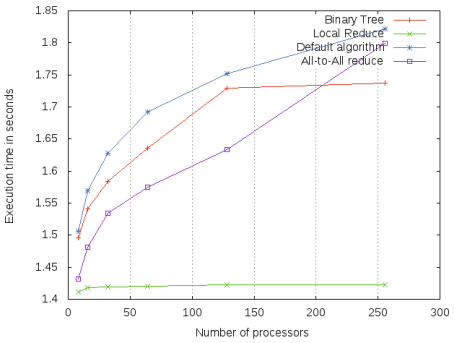

Fig. 8. Performance result for a 2^{20} elements vector using operator std::plus with an imbalance of 120ms at one node

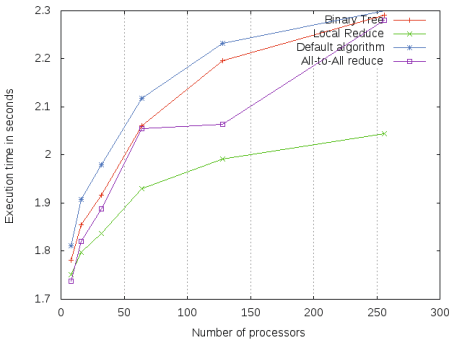

Fig. 9. Performance result for a 2^{20} elements vector using operator std::plus with gamma distributed imbalances

6 Conclusions

We can establish two important conclusions: when designing reduction algorithms one should take into account operators significantly more complex than std::plus, as our tests have confirmed that algorithms well suited for cheap operations do not necessarily perform as well when expensive operations come into play; secondly, load imbalances do impact the performance of state-of-the art reduction algorithms and there are ways, as we have shown, to mitigate this.

The next step should be to investigate what benefits could be achieved by using the ideas here presented in a real world scenario such as a distributed raytracing algorithm that exhibits all three identified types of imbalances and verifying whether the results obtained with these synthetic tests are indeed relevant.

Acknowledgments. This work is funded by Intel and by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT), in the context of the Exascience lab.

References

1. Rabenseifner, R., Träff, J.L.: More Efficient Reduction Algorithms for Non-Power-of-Two Number of Processors in Message-Passing Parallel Systems. In: Kranzlmüller, D., Kacsuk, P., Dongarra, J. (eds.) EuroPVM/MPI 2004. LNCS, vol. 3241, pp. 36–46. Springer, Heidelberg (2004)
2. Kumar, V., Grama, A., Gupta, A., Karypis, G.: Introduction to Parallel Computing. Benjamin/Cummings, Redwood City (1994)
3. Yu, H., Wang, C., Ma, K.L.: Massively parallel volume rendering using 2-3 swap image compositing. In: Proceedings of IEEE/ACM Supercomputing 2008 Conference, SC (2008)
4. Thakur, R., Rabenseifner, R., Gropp, W.: Optimization of collective communication operations in MPICH. International Journal of High Performance Computing Applications 19(1), 49–66 (2005)
5. Rabenseifner, R.: Optimization of Collective Reduction Operations. In: Bubak, M., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2004, Part I. LNCS, vol. 3036, pp. 1–9. Springer, Heidelberg (2004)
6. Hoefler, T., Schneider, T., Lumsdaine, A.: Accurately measuring overhead, communication time and progression of blocking and nonblocking collective operations at massive scale. International Journal of Parallel, Emergent and Distributed Systems 25(4), 241–258 (2010)
7. Worsch, T., Reussner, R., Augustin, W.: On Benchmarking Collective MPI Operations. In: Kranzlmüller, D., Kacsuk, P., Dongarra, J., Volkert, J. (eds.) PVM/MPI 2002. LNCS, vol. 2474, pp. 271–279. Springer, Heidelberg (2002)