Micro-Benchmarking MPI Neighborhood Collective Operations

Felix Donatus Lübbe*

Research Group for Parallel Computing
TU Wien, Austria
luebbe@par.tuwien.ac.at

Abstract. In this article, performance expectations for MPI neighborhood collective operations are formulated as *self-consistent performance guidelines*. A microbenchmark and an experimental methodology are presented to assess these guidelines. Measurement results from a large, InfiniBand-based cluster, the Vienna Scientific Cluster (VSC), as well as from a small commodity cluster computer are shown and discussed to illustrate the methodology and to gain first insights into the performance of current MPI implementations. Results show that the examined libraries seem to be sensitive to the order in which topological neighbors are specified, and that in some cases Cartesian topologies can be outperformed by simulating them with distributed graph topologies.

Keywords: MPI, Process topology, Neighborhood collectives, Performance Guidelines, Benchmarking

1 Problem Statement

Neighborhood collective operations have been introduced to the MPI standard in version 3.0 [5]. Not only could they simplify the code of, for example, multidimensional stencil computations, but also offer a performance benefit over naive handwritten exchange algorithms using `MPI_Send` and `MPI_Recv`.

So far no microbenchmarks are available to assess the performance of MPI neighborhood collectives on virtual topologies. Intel MPI Benchmarks 2017[1], OSU Micro-Benchmarks 5.3.2[2] and SKaMPI 5.0.4[3] [6] do not offer such functionality at all. While NBCBench 1.1[4] [2] can measure LibNBC's nonblocking neighborhood Alltoall(v) algorithms, it has not been extended and used to measure the corresponding MPI operations. Further, the used neighborhood is built using the deprecated operation `MPI_Graph_create`. The only parameter available for

* This work was supported by the Austrian Science Fund (FWF): P25530.

[1] `https://software.intel.com/en-us/articles/intel-mpi-benchmarks`, last checked 2017-05-26

[2] `http://mvapich.cse.ohio-state.edu/benchmarks/`, last checked 2017-05-26

[3] `http://liinwww.ira.uka.de/~skampi/`, last checked 2017-05-26

[4] `http://htor.inf.ethz.ch/research/nbcoll/perf/`, last checked 2017-05-26

topology construction is the number of neighbors per process. The structure of the neighborhood can not be varied further.

In [9] a microbenchmark has been used to compare the durations of a new family of sparse collective operations, which work on isomorphic neighborhoods, to those of the corresponding MPI neighborhood collectives. However, while the MPI operations served as a baseline to explicate performance expectations for the new operations, no expectations for the MPI functions have been formulated and assessed there.

In this article, performance expectations for MPI neighborhood collective operations, as well as for the topology creation functions `MPI_Cart_create`, `MPI_Dist_graph_create` and `MPI_Dist_graph_create_adjacent` are motivated and semiformalized using the concept of self-consistent performance guidelines [7]. A microbenchmark based on the one used in [9] is described in detail, which is able to semiautomatically assess these guidelines and generate plots of violations, partly using the concepts presented in [4]. Setup and results of first measurements on two different cluster computers, as well as the assessment of a subset of the presented guidelines are shown to illustrate the methodology and to gain first insights into the performance of current MPI libraries.

Section 2 describes performance guidelines for neighborhood collectives and topology creation functions. In Sect. 3 the benchmark is introduced. Section 4 details the experimental setup of the measurements carried out. The results of the experiments are shown and analyzed in Sect. 5. Section 6 concludes the article.

2 Performance Guidelines for Neighborhood Collectives

Self-consistent performance guidelines are a means to express performance expectations for MPI in a semiformal way by relating the durations of different (combinations of) MPI operations which yield the same effect. Since the MPI standard does not impose any performance requirements, the guidelines are argued for on the basis of self-evident user expectations, which are represented by a set of metarules in [7].

A guideline of the form $a \preceq b$ means that operation a shall not be slower than operation b, given all common parameters of both operations are equal [7]. Accordingly, $a \approx b$ means a and b shall perform similar. The relation is required to hold in the average case for many runs, while isolated counterexamples possibly due to lazy initialization or disturbing factors during the measurement are not considered a violation. If $a \preceq b$ is violated, performance would increase if the user replaced a with b in the violating scenario.

In this section, the following performance guidelines will be motivated:

$$\texttt{Cart\_create} \preceq \texttt{Dist\_graph\_create\_adjacent}_{\text{Cart}} \quad \text{(GL1)}$$

$$\texttt{Dist\_graph\_create\_adjacent} \preceq \texttt{Dist\_graph\_create} \quad \text{(GL2)}$$

$$\texttt{X\_create}_{\texttt{reorder=0}} \preceq \texttt{X\_create}_{\texttt{reorder=1}} \quad \text{(GL3)}$$

$$\texttt{Neigh\_allgather(v)} \preceq \texttt{Neigh\_alltoall(v)} \quad \text{(GL4)}$$

$$\texttt{Neigh\_allgather} \preceq \texttt{Neigh\_allgatherv} \tag{GL5}$$

$$\texttt{Neigh\_alltoall} \preceq \texttt{Neigh\_alltoallv}$$

$$\preceq \texttt{Neigh\_alltoallw} \tag{GL6}$$

$$\texttt{Allgather(v)}_{\text{full}} \preceq \texttt{Neigh\_allgather(v)}_{\text{full}} \tag{GL7}$$

$$\texttt{Alltoall(v/w)}_{\text{full}} \preceq \texttt{Neigh\_alltoall(v/w)}_{\text{full}} \tag{GL8}$$

$$\texttt{Neigh\_x}_{\texttt{Cart}} \preceq \texttt{Neigh\_x}_{\texttt{Graph\_adj}(\text{Cart})} \tag{GL9}$$

$$\texttt{Neigh\_x}_{\texttt{Graph\_adj}} \approx \texttt{Neigh\_x}_{\texttt{Graph}} \tag{GL10}$$

$$\texttt{Neigh\_x}_{\texttt{rank list ordering 1}} \approx \texttt{Neigh\_x}_{\texttt{rank list ordering 2}} \tag{GL11}$$

$$\texttt{Neigh\_x}_{\texttt{reorder=1}} \preceq \texttt{Neigh\_x}_{\texttt{reorder=0}} \tag{GL12}$$

GL1 states that if a Cartesian-shaped topology is constructed, the specialized `MPI_Cart_create` should not be slower than `MPI_Dist_graph_create_adjacent`, which can construct topologies of arbitrary shape (cf. metarule 3).

A `DISTGRAPH` topology can be created either by `MPI_Dist_graph_create` or by `MPI_Dist_graph_create_adjacent`. While in a call to `MPI_Dist_graph_create` every process may specify an arbitrary set of edges of the topology graph, `MPI_Dist_graph_create_adjacent` imposes the precondition that every process passes exactly its incident edges. Because of this additional requirement, `MPI_Dist_graph_create_adjacent` shall not be slower (GL2, cf. metarule 2).

GL3 asserts that allowing a topology constructor to change the mapping of rank numbers to actual processes by setting the reorder flag to 1 should not speed up the actual creation, since reordering would be beneficial for subsequent communication operations on the topology and disabling it is, from a performance point of view, only reasonable to save extra cost during communicator creation.

`MPI_Neighbor_allgather` could be mimicked by `MPI_Neighbor_alltoall`, if the send buffer is copied locally n times, and should therefore not be slower. The same is true for the respective vector variants (GL4). `MPI_Neighbor_allgatherv` and `MPI_Neighbor_alltoallv/-w` can mimick their regular counterparts, which should therefore not be slower (GL5, GL6, cf. metarule 3).

The neighborhood collectives can be used to simulate the global collectives `MPI_Allgather(v)` and `MPI_Alltoall(v/w)`, if a fully connected graph topology is created. While neighborhood collectives support any topology, global collectives always follow a complete graph and because of this specialization should not be slower (GL7, GL8, cf. metarule 3).

If a Cartesian-shaped topology is created using one of the distributed graph constructors, neighborhood collectives should not get faster compared to the semantically stricter Cartesian topology created by `MPI_Cart_create` (GL9, cf. metarule 2). However, their performance for any `DISTGRAPH` topology should be independent of the constructor, because `DISTGRAPH` constructors produce semantically equivalent topologies (GL10).

GL11 states that neighborhood collectives should perform similar on isomorphic topologies, independent of the ordering of the list of ranks passed to the topology constructor to define edges. If this was not respected by an MPI library, the user would be tempted to find the "sweet" ordering herself, possibly

breaking performance portability between libraries. If the implementations of the neighborhood collectives of a specific library had such sweet orderings, the topology constructor should reorder its input lists accordingly.

Allowing the ranks to be reordered during communicator creation shall not slow down any neighborhood collective since the whole point with reordering is optimizing communication performance (GL12).

3 The Benchmark

The microbenchmark used for the experiments comprises a kernel executing the actual measurements and a framework of scripts responsible for control flow, input generation and output analysis. It makes use of findings from [9,4,3].

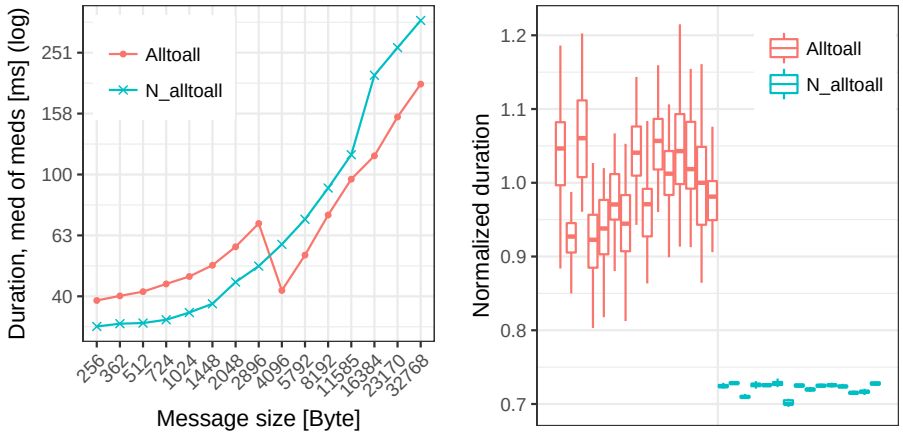
The main goal of the benchmark is to help identify performance problems of MPI implementations in specific environments. Although some decision metrics are defined to enable automatically finding violations of performance guidelines, the quantification of a violation is of less interest than the fact that a violation has been found. While the question for the severeness of a certain violation might uncover sensational answers, it will not help much solving it, except maybe to set priorities for which one to tackle first. In fact, investigating its cause by, for example, looking into algorithms and parameters of the MPI implementation is the step meant to follow the use of the benchmark.

3.1 Kernel

The kernel implements *measurement setups* for different MPI operations, following the form of Algorithm 1. All input parameters are read from a CSV input file with each line describing one experiment. Apart from the parameters specific to the topology and explained below, an experiment description includes the communication operation to use, the MPI datatype, the message length, the number of consecutive repetitions n_{rep} and a 64-bit integer w which is decremented down to 0 in a loop written in inline assembler to accurately simulate local computation of specific CPU time when measuring the overlap of nonblocking operations.

The synchronization is implemented as a handwritten dissemination barrier like in [9] to improve comparability between different MPI implementations, which might use different algorithms for their `MPI_Barrier`. `MPI_Wtime` is used to retrieve wall clock time with high resolution. The maximum duration of all parallel processes is output as the result time $\Delta t[i]$ of each single repetition i.

Currently, four different types of neighborhoods are supported, three of which can be described using a similar mask of relative coordinates for each process and a mapping of process ranks to a virtual Cartesian grid of variable dimensionality: The *Cartesian* neighborhood just uses the `MPI_Cart_create` topology constructor, giving the simplest form of a multidimensional isomorphic neighborhood by only including the two immediate neighbors along each dimension in the Cartesian grid. The *Moore* neighborhood of radius r, on the other hand, includes



(a) Absolute durations for all message sizes, accumulated over all runs.

(b) Box plot of durations in individual runs for message size 2896 B, normalized to median of medians of `Alltoall`. Boxes mark quartiles, whiskers the total range of durations.

Fig. 1. Results comparing `Alltoall` to `Neigh_alltoall`, campaign full-rand-jupiter, 35×16 processes, Full neighborhood with `RAND` ordering and `reorder` $= 0$, outliers removed.

Algorithm 1 Measurement procedure in the kernel

for all experiment descriptions in input file **do**
  Create topology communicator
  for $i = 0$ to $n_{\text{rep}} - 1$ **do**
    Barrier synchronization
    $t_0 \leftarrow$ `Wtime`
    `MPI_(I)Neighbor_X`
    Simulate work: Count from w down to 0
    (`MPI_Wait`)
    $\Delta t[i] \leftarrow$ `Wtime` $- t_0$
    Barrier synchronization
  end for
  Free topology communicator
  `MPI_Reduce`($\Delta t[]$, length i, maximum, to rank 0)
  Output $\Delta t[]$ at rank 0
end for

all ranks in the grid within a hypercube with edges of length $2r + 1$ around the process. The *von Neumann* neighborhood of radius r is a subset of the Moore neighborhood, including only those ranks with relative coordinates c with a Manhattan distance $\leq r$ from the process, i.e. $\sum_{i=1}^{n_{\text{dim}}} |c_i| \leq r$.

The communicators for Moore and von Neumann neighborhoods are constructed using `MPI_Dist_graph_create` or `MPI_Dist_graph_create_adjacent`. The relative coordinates are computed and then translated to a list of source- and destination ranks respectively using an intermediate Cartesian communicator and `MPI_Cart_rank`. The list of source ranks is coordinatewise inverse to the list of destination ranks. `MPI_Dist_graph_create_adjacent` constructs the topology using both lists. In case `MPI_Dist_graph_create` is used, each process only passes its destination list, leaving the ordering of any internal processwise source list within the communicator to the MPI implementation.

Input parameters for Cartesian, Moore and von Neumann neighborhoods are the number of dimensions n_{dim}, the number of finite dimensions n_{fin}, i.e. how many of the dimensions are nontoroidal, and the `reorder` flag of the MPI topology constructor specifying whether the MPI library is allowed to change the mapping of rank numbers to processes to better fit the actual network topology and accelerate communication. This flag does not affect the intermediate Cartesian communicator used to construct Moore and von Neumann neighborhoods, whose `reorder` flag is always set to 0.

Further parameters for Moore and von Neumann neighborhoods include the radius r and the ordering of the list of relative coordinates before translation to rank numbers. Possible orderings are first- and last-coordinate-major (`FMAJ`, `LMAJ`) and randomized (`RAND`). `FMAJ` (`LMAJ`) means coordinatewise sorted in ascending order with first (last) coordinate changing last. `RAND` means the list is permuted using a different random seed for each process.

The dimensions of the Cartesian grid in case of Cartesian, Moore and von Neumann neighborhoods are calculated using the `TUW_Dims_create` function implementing the algorithm described in [8], as the result of `MPI_Dims_create` has proven to break portability between MPI libraries in the past.

The fourth neighborhood type, the *Full* neighborhood, connects all processes in a complete graph, i.e. every process is neighbor of all other processes. Like with Moore and von Neumann neighborhoods, it can be selected whether `MPI_Dist_graph_create` or `MPI_Dist_graph_create_adjacent` is used as the constructor. Input parameters further include the reordering flag and the ordering of the sources- and destinations list. `LINEAR` ordering means all rank numbers starting from the process itself are enumerated incrementally (sources) and decrementally (destinations) modulo the total number of processes. `RAND` ordering means both sources and destinations array are randomized independently and with a different random seed for each process. If `MPI_Dist_graph_create` is used, only the destinations array is passed.

3.2 Framework Scripts

Control flow of a measurement campaign is programmed in bash scripts, which offer an immediate way to automatize calling programs and manage input- and output files. The work flow to run a measurement campaign is semiautomatized by encapsulating five user-invoked steps: (1) building the kernel, (2) creating input files and job scripts, (3) submitting the job scripts to the scheduling system, (4) archiving the results, (5) analyzing results for performance guideline violations and drawing plots of detected violations.

The files to configure a measurement campaign include a *campaign configuration* file, in which a list of process deployments is specified, e.g. $[2 \times 8, 4 \times 8]$ for 2 and 4 nodes with 8 processes each. The number of distinct calls to the kernel, n_{run}, is set, as well as a maximum run time, after which the scheduler will kill the job. A separate *environment configuration* is referenced, containing all machine specific settings like paths and the syntax of the `mpirun` command.

The input to the kernel, i.e. the actual experiments carried out, is generated in step (2) using a Python script, which makes modelling all kinds of relations between different input parameters easy, e.g. "for all n_{dim} create experiments with $n_{\text{fin}} \in \{0, 1, \ldots, n_{\text{dim}}\}$". For each run of the kernel, a separate input file is created with a different random permutation of the same set of experiments to mitigate systematic bias by disturbing factors.

Processing results and assessing the guidelines is done in an R script in step (5). Some assessment configuration needs to be set up by the user: all parameters of the campaign must be subdivided into a *guideline parameter*, a *varied parameter* and *grouping parameters*. The guideline parameter contains the levels to be compared within a guideline – if `Neigh_allgather` and `Neigh_alltoall` are to be compared, the guideline parameter would be the *measurement setup*. A list of guidelines of the form $a \preceq b$, with a, b being levels of the selected guideline parameter, must be provided. The varied parameter will be on the x axis of subsequently generated plots and could, for example, be the message size. All remaining parameters, e.g., neighborhood type, n_{dim}, n_{fin}, ..., are considered grouping parameters, with every combination of their levels implying a unique group. For each group containing at least one violation, plots will be generated. The script must be rerun for every different guideline parameter.

The script will first calculate the median $m_l^r := \text{med}(\text{dropOutliers}(\Delta t_l^r[0], \ldots, \Delta t_l^r[n_{\text{rep}} - 1]))$ of the n_{rep} single durations of each run r and each combination of parameter levels l after filtering outliers. This results in n_{run} medians $m_l^0, \ldots, m_l^{n_{\text{run}}-1}$ for each combination of parameter levels l. Outliers are values outside of $[q_1 - 1.5(q_3 - q_1), q_3 + 1.5(q_3 - q_1)]$, with quartiles q_1, q_3, like suggested by Tukey [1, Subsec. 3.2.4].

For each guideline $a \preceq b$ and each unique combination of the grouping parameters and the varied parameter, the n_{run} medians for a and b are selected. The Wilcoxon rank sum test is then carried out to test whether the medians of a are shifted to the right of b [4,1, Subsec. 7.4.6]. Further, the *violation ratio* $v := \frac{\text{med}(m_a^0, \ldots, m_a^{n_{\text{run}}-1})}{\text{med}(m_b^0, \ldots, m_b^{n_{\text{run}}-1})}$ is computed to quantify the difference between a and b. $a \preceq b$ is considered violated for the selected parameter levels, if $v \geq v_{\text{thres}}$ and

the test returns a p-value $\leq p_{\text{thres}}$. $p_{\text{thres}}, v_{\text{thres}}$ are set by the user. The threshold for v filters very small violations considered significant by the statistical test.

For each violation, an *overview plot* of the affected group is created, which shows the medians of the medians of the result times for both parameter levels a, b in absolute numbers on a log scale, with the varied parameter, e.g. the message size, on the x axis. Further, for each violation, a *focus plot* is generated, which shows the distributions of the raw results within the individual runs as box plots, normalized to the median of the medians of the durations of a. Figures 1a and 2a give examples for overview plots, Figures 1b and 2b for focus plots.

4 Experimental Setup

Five measurement campaigns on two different cluster computers have been carried out to assess a subset of the formulated guidelines (see Table 2). In the *nbhcoll* campaigns, neighborhood collective operations have been executed on Cartesian topologies, as well as on von Neumann and Moore neighborhoods of radius 1, which could all be used in real-world applications performing stencil computations [9]. In the *full* campaigns, a complete graph is used as topology, making the neighborhood collectives behave like their global collective counterparts, which have been measured here as well. Campaigns *full-rand-jupiter* and *full-tuned-jupiter* have been set up and executed because of findings from *full-jupiter*; see Sect. 5 for details.

The von Neumann neighborhood of radius 1 exactly resembles a Cartesian topology; the subsequently used notation `Cart` $\preceq$ `Vneum` refers to GL9. GL11 is tested by comparing neighbor list orderings `FMAJ` and `LMAJ` (nbhcoll) or `LINEAR` (full) to `RAND`. The term `reorder=1` $\preceq$ `reorder=0` refers to GL12.

Table 2 lists all parameters of the executed experiments together with the parameter levels used in the respective campaigns. For example, in campaign nbhcoll-jupiter, the two operations `Neigh_allgather` and `Neigh_alltoall` have each been measured with 15 different message sizes, on three different topologies, with four different numbers of dimensions, two different values for the number of finite dimensions, three different orderings of the list of neighbors in case of von Neumann and Moore neighborhoods (Cartesian topologies do not have an ordering), and both possible values for the reorder flag during communicator creation. This makes a total of 3360 unique combinations of parameter levels, which are experimentally measured $n_{\text{rep}} = 50$ times in each of $n_{\text{run}} = 30$ runs. If, for example, the guideline `Neigh_allgather` $\preceq$ `Neigh_alltoall` is evaluated, i.e. the *measurement setup* is chosen as the guideline parameter, the statistical test is executed for the $\frac{3360}{2} = 1680$ unique combinations of the remaining parameter levels. Since the varied parameter is the message size, results are presented in $\frac{1680}{15} = 112$ groups.

The first system, Jupiter, has 36 nodes with two AMD Opteron 6134 8-core processors at 2.3 GHz and 32 GiB memory each, connected via a Mellanox MT4036 InfiniBand QDR crossbar switch. The second system is VSC3 at the Vienna Scientific Cluster, consisting of 2020 nodes with two Intel Xeon E5-2650v2

8-core processors at 2.6 GHz and 64 GiB memory each. The nodes are connected by an InfiniBand QDR-80 fat tree architecture. On Jupiter, both nodes and network links involved in the measurements were dedicated to the benchmark. On VSC3, only the nodes were dedicated, while network switches were possibly shared with other jobs. The benchmark has been compiled and run using gcc 4.4.7 and Open MPI 2.0.1 on Jupiter and gcc 5.3.0 and Intel MPI 2017.1 on VSC3.

The dimensions of the virtual Cartesian grid of processes for the different process deployments and number of dimensions in the nbhcoll campaigns are listed in Table 1.

Table 1. Dimensions array returned by `TUW_Dims_create` for different n_{dim} and n_{procs}.

	Process deployment		
n_{dim}	10×16	20×16	35×16
2	$\{16, 10\}$	$\{20, 16\}$	$\{28, 20\}$
3	$\{8, 5, 4\}$	$\{8, 8, 5\}$	$\{10, 8, 7\}$
4	$\{5, 4, 4, 2\}$	$\{5, 4, 4, 4\}$	$\{7, 5, 4, 4\}$

5 Results

Table 3 lists the numbers of violations of different guidelines for the nbhcoll campaigns on Jupiter and VSC3. Each cell contains two rows: first, the total numbers of violations and tests, second the numbers of groups containing at least one violation as well as the total number of groups. In a group, all parameters are similar except the message length (varied parameter) and the respective guideline parameter. The threshold values for the assessment are set to $p_{\text{thres}} = 0.001$ and $v_{\text{thres}} = 1.03$. Different thresholds have been tried, but for higher p-values and lower violation ratios, violations were often not clearly visible in the plots.

In the nbhcoll campaigns, the guideline `Neigh_allgather` $\preceq$ `Neigh_alltoall` was only violated for the smaller numbers of processes. On Jupiter, violations occurred for $n_{\text{dim}} \in \{2, 4\}$ and $n_{\text{fin}} = 0$, on all three neighborhoods, for all orderings of the neighborhood coordinates, with ratios up to 1.049. The two violations on VSC3 occurred with a fourdimensional Moore neighborhood, $n_{\text{fin}} = 4$, `LMAJ` ordering, `reorder` $\in \{0, 1\}$ and a message size of 4 KiB. Their exceptionally high ratio of about 13.8 each stems from a peculiar effect observed on VSC3 for different measurements: the relative dispersion of many runs is in the same order of magnitude like the violation ratio, with the quartiles of many runs spanning from the median of medians of the `Neigh_allgather` times to the median of medians of the `Neigh_alltoall` times. Usually, dispersion was much lower, like in the figures from Jupiter in this article. Unfortunately, due to time restrictions, this effect could not be investigated further for this article. The measurements should be rerun with a different node allocation to eliminate a possible interdependency

Table 2. Measurement campaigns referenced in this article.

	nbhcoll-jupiter	nbhcoll-vsc3	full-jupiter	full-rand-jupiter	full-tuned-jupiter
n_{run}	30	15	15	15	15
n_{rep}	50	50	50	50	50
Measurement setups	`Neigh_allgather` `Neigh_alltoall`	`Neigh_allgather` `Neigh_alltoall`	`Neigh_allgather` `Neigh_alltoall` `Allgather` `Alltoall`	`Neigh_alltoall` `Alltoall`	`Neigh_alltoall` `Alltoall`
Message sizes	$256\,\text{B}, 362\,\text{B}, 512\,\text{B}, \ldots, 32\,\text{KiB}$ (factor $\sqrt{2}$ between sizes)				
Neighborhoods	Cartesian v.Neum.* Moore* *$r = 1$, using `Dist_graph_create_adjacent`	Cartesian v.Neum.* Moore*	Full using `Dist_graph_create_adjacent`	Full using `Dist_graph_create_adjacent`	Full using `Dist_graph_create_adjacent`
n_{dim}	$1, 2, 3, 4$	$1, 2, 3, 4$	—	—	—
n_{fin}	$0, n_{\text{dim}}$	$0, n_{\text{dim}}$	—	—	—
Order of adj.list (Moore, v.Neum., Full)	`FMAJ` `LMAJ` `RAND`	`FMAJ` `LMAJ` `RAND`	`LINEAR` only `Neigh_*`: `RAND`	`RAND`	`LINEAR` `RAND`
Reorder	$0, 1$	$0, 1$	$0, 1$	0	0
Open MPI MCA Parameters	default	—	default	default	`coll_tuned_use_dynamic_rules=true`, `coll_tuned_alltoall_intermediate_msg=256`

between node allocation, virtual topology and communication algorithm. Note that temporary network effects can already be excluded as a cause due to the randomization of experiments.

On Jupiter, the guideline `Cart` $\preceq$ `Vneum` has been violated only with four-dimensional neighborhoods, while the violation ratio did not exceed 1.045. On VSC3, most violations happened for $n_{\text{dim}} = 4$ as well, including the most severe ones with ratios up to 1.222. For 35×16, violations occurred only with $n_{\text{dim}} = 4$, for 20×16 with $n_{\text{dim}} \in \{3, 4\}$, and for 10×16 even with $n_{\text{dim}} \in \{2, 3, 4\}$.

The guideline `FMAJ` $\preceq$ `RAND` was violated only by Moore neighborhoods with $n_{\text{dim}} \in \{2, 3, 4\}$ on Jupiter, with a ratio of up to 1.147. `LMAJ` $\preceq$ `RAND` was violated by both Moore and von Neumann neighborhoods, but only for $n_{\text{dim}} = 4, n_{\text{fin}} = 0$ and 10×16 processes. Moore neighborhoods yielded a ratio of up to 1.206. On VSC3, violations of both guidelines occurred for all values of the grouping parameters. The biggest ratio observed in all experiments, 141.9, occurred for `FMAJ` $\preceq$ `RAND`, `Neigh_alltoall`, a Moore neighborhood with $n_{\text{dim}} = 2, n_{\text{fin}} = 2$, independent of reordering and for a message size of 11 585 B. This enormous ratio was due to the same effect on VSC3 mentioned above. However, most of the other reported violations did not suffer from this effect.

The only guideline violated in campaign full-jupiter was `LINEAR` $\preceq$ `RAND` and most violations were quite clear. Top ratios increased with number of processes from 1.083 (10×16) to 1.828 (35×16). While for 10×16 processes, only the smaller message sizes up to 1448 B were affected, for 30×16 processes violations occurred in the whole spectrum of the message sizes used.

In the full-jupiter campaign, to save time, the global collectives were only executed on topologies with `LINEAR` ordering because ordering was assumed to make no difference for them. Since the architecture of the benchmark only allows for the levels of one parameter being compared to each other, with all other parameters being the same, comparing neighborhood collectives with `RAND` orderings to global collectives was not possible in this campaign. However, the fact that the guideline `LINEAR` $\preceq$ `RAND` was violated so often together with the observation of `Alltoall` and `Neigh_alltoall` performing similar with `LINEAR` ordering for small message sizes lead to the assumption that `Alltoall` $\preceq$ `Neigh_alltoall` could be violated on full `RAND` topologies. Therefore, campaign full-rand-jupiter was set up and executed, and indeed showed the expected violations for small message sizes (cf. Fig. 1).

A closer look into Open MPI revealed that the algorithm for `Alltoall` is changed by default at a message size of 3000 B. The so called *MCA parameters* allow to change such thresholds at runtime. In the campaign full-tuned-jupiter, the violations could be healed by setting the threshold message size to 256 B (cf. Fig. 2). In the case of `LINEAR` ordering, `Alltoall` now was considerably faster than `Neigh_alltoall` as well.

In the three campaigns nbhcoll-jupiter, nbhcoll-vsc3 and full-jupiter, the `reorder=1` $\preceq$ `reorder=0` guideline was never violated and the reordering flag did not seem to have an effect on violations of the other guidelines. Subsequent experiments just creating the topologies used in the campaigns with reordering enabled

and checking for a change in process-to-rank mapping in the new communicator confirmed this conjecture. Campaigns full-rand-jupiter and full-tuned-jupiter have therefore been set up with reordering disabled in general.

Table 3. Number of guideline violations in experiments with different neighborhoods of radius 1 on Jupiter and VSC3. Format: $n_{\text{vioTests}}/n_{\text{tests}}$ ($n_{\text{vioGroups}}/n_{\text{groups}}$).

	nbhcoll-jupiter			nbhcoll-vsc3		
	10×16	20×16	35×16	10×16	20×16	35×16
`Neigh_allgather` $\preceq$ `Neigh_alltoall`	15/1680 (15/112)	1/1680 (1/112)	0/1680 (0/112)	2/1680 (2/112)	0/1680 (0/112)	0/1680 (0/112)
`Cart` $\preceq$ `Vneum`	19/480 (8/32)	2/480 (2/32)	1/480 (1/32)	117/480 (20/32)	36/480 (13/32)	9/480 (7/32)
`FMAJ` $\preceq$ `RAND`	81/960 (16/64)	118/960 (20/64)	104/960 (14/64)	87/960 (36/64)	276/960 (36/64)	25/960 (13/64)
`LMAJ` $\preceq$ `RAND`	27/960 (7/64)	0/960 (0/64)	0/960 (0/64)	198/960 (33/64)	208/960 (35/64)	14/960 (13/64)
`reorder=1` $\preceq$ `reorder=0`	0/1680 (0/112)	0/1680 (0/112)	0/1680 (0/112)	0/1680 (0/112)	0/1680 (0/112)	0/1680 (0/112)

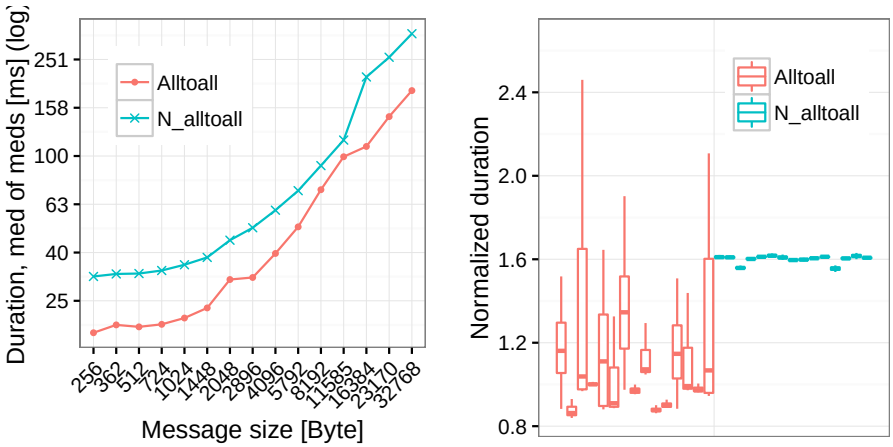
6 Conclusion and Outlook

Performance guidelines help to express expectations for neighborhood collectives in a formal way, to enable computers to automatically check them on a large number of measurements. Results show that current MPI implementations probably have room for improvement of their performance, although admittedly, not every violation can easily be attributed to the MPI implementation in the complex environment of a cluster computer without further investigation. Still, especially the cases where simulating a Cartesian with a similar `DISTGRAPH` topology increases performance are surprising, since algorithms could benefit from the fixed structure of Cartesian topologies. This, together with the violations of GL11, suggests that the examined MPI implementations are sensitive to the ordering of neighbors.

In the future it would be interesting to execute similar campaigns on further cluster computers, especially such with a network topology resembling a Cartesian grid. Measuring with bigger neighborhoods could be interesting as well, although the question arises whether there are problems from the real world which would be affected by the results. Of course, the remaining guidelines formulated, but not evaluated in this article should be tested – especially those dealing with different methods of communicator creation. Since some MPI implementations nowadays

Table 4. Number of guideline violations in campaign full-jupiter. Format: $n_{\text{vioTests}}/n_{\text{tests}}$ ($n_{\text{vioGroups}}/n_{\text{groups}}$).

	full-jupiter		
	10×16	20×16	35×16
`Allgather` $\preceq$ `Neigh_allgather`	0/30 (0/2)	0/30 (0/2)	0/30 (0/2)
`Alltoall` $\preceq$ `Neigh_alltoall`	0/30 (0/2)	0/30 (0/2)	0/30 (0/2)
`Neigh_allgather` $\preceq$ `Neigh_alltoall`	0/60 (0/4)	0/60 (0/4)	0/60 (0/4)
`LINEAR` $\preceq$ `RAND`	22/60 (4/4)	30/60 (4/4)	50/60 (4/4)
`reorder=1` $\preceq$ `reorder=0`	0/90 (0/6)	0/90 (0/6)	0/90 (0/6)



(a) Absolute durations for all message sizes, accumulated over all runs.

(b) Box plot of durations in individual runs for message size 2896 B, normalized to median of medians of `Alltoall`. Boxes mark quartiles, whiskers the total range of durations.

Fig. 2. Results comparing `Alltoall` to `Neigh_alltoall`, campaign full-tuned-jupiter, 35×16 processes, Full neighborhood with `RAND` ordering and `reorder` $= 0$, outliers removed.

promise true asynchronous progress, guidelines for nonblocking neighborhood collectives should be formulated and assessed as well.

Acknowledgments. The author would like to thank Alexandra Carpen-Amarie, Sascha Hunold, Jesper Larsson Träff and Thomas Worsch for helpful discussions concerning this article, as well as the anonymous reviewers for their valuable feedback. The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-319-64203-1_5.

References

1. Hedderich, J., Sachs, L.: Angewandte Statistik. Springer Spektrum, 15th edn. (2016)
2. Hoefler, T., Lumsdaine, A., Rehm, W.: Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In: Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07. IEEE Computer Society/ACM (Nov 2007)
3. Hunold, S., Carpen-Amarie, A.: Reproducible MPI benchmarking is still not as easy as you think. IEEE Trans. Parallel Distrib. Syst. 27(12), 3617–3630 (2016), http://dx.doi.org/10.1109/TPDS.2016.2539167
4. Hunold, S., Carpen-Amarie, A., Lübbe, F.D., Träff, J.L.: Automatic verification of self-consistent MPI performance guidelines. In: Dutot, P., Trystram, D. (eds.) Euro-Par 2016: Parallel Processing - 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9833, pp. 433–446. Springer (2016), http://dx.doi.org/10.1007/978-3-319-43659-3_32
5. MPI Forum: MPI: A message-passing interface standard, version 3.0 (Sep 2012)
6. Reussner, R.H., Sanders, P., Träff, J.L.: SKaMPI: a comprehensive benchmark for public benchmarking of MPI. Scientific Programming 10(1), 55–65 (2002), http://content.iospress.com/articles/scientific-programming/spr00094
7. Träff, J.L., Gropp, W.D., Thakur, R.: Self-consistent MPI performance guidelines. IEEE Trans. Parallel Distrib. Syst. 21(5), 698–709 (2010), http://dx.doi.org/10.1109/TPDS.2009.120
8. Träff, J.L., Lübbe, F.D.: Specification guideline violations by MPI_Dims_create. In: Proceedings of the 22nd European MPI Users' Group Meeting, EuroMPI 2015, Bordeaux, France, September 21-23, 2015. pp. 19:1–19:2 (2015), http://doi.acm.org/10.1145/2802658.2802677
9. Träff, J.L., Lübbe, F.D., Rougier, A., Hunold, S.: Isomorphic, sparse MPI-like collective communication operations for parallel stencil computations. In: Proceedings of the 22nd European MPI Users' Group Meeting, EuroMPI 2015, Bordeaux, France, September 21-23, 2015. pp. 10:1–10:10 (2015), http://doi.acm.org/10.1145/2802658.2802663