All-gather Algorithms Resilient to Imbalanced Process Arrival Patterns

JERZY PROFICZ, Gdansk University of Technology, Centre of Informatics—Tricity Academic Supercomputer & networK (CI TASK), Poland

Two novel algorithms for the all-gather operation resilient to imbalanced process arrival patterns (PATs) are presented. The first one, Background Disseminated Ring (BDR), is based on the regular parallel ring algorithm often supplied in MPI implementations and exploits an auxiliary background thread for early data exchange from faster processes to accelerate the performed all-gather operation. The other algorithm, Background Sorted Linear synchronized tree with Broadcast (BSLB), is built upon the already existing PAP-aware gather algorithm, that is, Background Sorted Linear Synchronized tree (BSLS), followed by a regular broadcast distributing gathered data to all participating processes. The background of the imbalanced PAP subject is described, along with the PAP monitoring and evaluation topics. An experimental evaluation of the algorithms based on a proposed mini-benchmark is presented. The mini-benchmark was performed over 2,000 times in a typical HPC cluster architecture with homogeneous compute nodes. The obtained results are analyzed according to different PATs, data sizes, and process numbers, showing that the proposed optimization works well for various configurations, is scalable, and can significantly reduce the all-gather elapsed times, in our case, up to factor 1.9 or 47% in comparison with the best state-of-the-art solution.

CCS Concepts: • **Software and its engineering → Message passing**; • **Computer systems organization → Distributed architectures**; • **Theory of computation → Distributed algorithms**;

Additional Key Words and Phrases: All-gather, background disseminated ring, background sorted linear synchronized tree with broadcast, process arrival pattern, MPI

ACM Reference format:
Jerzy Proficz. 2021. All-gather Algorithms Resilient to Imbalanced Process Arrival Patterns. *ACM Trans. Archit. Code Optim.* 18, 4, Article 41 (July 2021), 22 pages.
https://doi.org/10.1145/3460122

1 INTRODUCTION

Nowadays, the **high-performance computing (HPC)** is an important, rapidly developing domain. Supercomputers help researchers to solve their problems and overcome numerous challenges in modern science. Virtual experiments performed using such environments are much faster and usually significantly cheaper. Moreover, sometimes such an approach is the only possible way to move forward new researches, especially in such subjects as molecular chemistry or quantum

New paper, not an extension of a conference paper.
This research was supported by Centre of Informatics—Tricity Academic Supercomputer & networK (CI TASK) in Gdansk University of Technology.
Author's address: J. Proficz, Gdansk University of Technology, Centre of Informatics—Tricity Academic Supercomputer & networK (CI TASK), Gabriela Narutowicza 11/12, Gdansk, Poland, 80-233; email: j.proficz@task.gda.pl.

1544-3566/2021/07-ART41
https://doi.org/10.1145/3460122

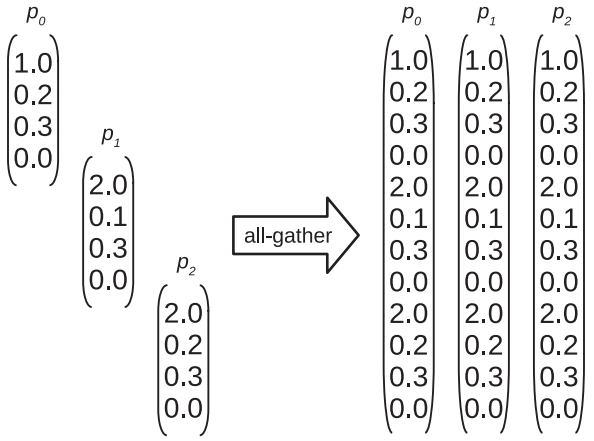

Fig. 1. Example of an all-gather collective operation. Cooperating processes are denoted as p_0, p_1, p_2.

physics. Thus, any improvements in the underlying HPC systems can accelerate the development of many non-strictly ICT-related science fields.

A typical supercomputer is usually based on a cluster architecture, where each node has its own operating system (e.g., Linux CentOS). It consists of a set of hundreds or even thousands of compute nodes interconnected by a fast network, usually InfiniBand [5] and/or Ethernet. The workload management is based on a queue system (e.g., SLURM [27]), in which jobs queued by users are scheduled and consecutively forwarded to the execution. Each job can consist of a number of processes, which in turn can contain their own threads cooperating by means of shared memory mechanisms, such as OpenMP [16]. The processes can be placed on different compute nodes and they usually use message-passing paradigm for data transfer and synchronization purposes, often supported by a number of **MPI (Message Passing Interface)** [14] implementations, such as open source OpenMPI [17], MPICH [15], or vendor locked ones, e.g., Intel MPI [7]. An example of such an HPC environment is Tryton supercomputer [9], located in **Centre of Informatics—Tricity Academic Supercomputer & networK (CI TASK)** at Gdansk University of Technology, where the research presented in this article was performed.

In such an HPC environment, the interconnected nodes are *de facto* separated servers, having their own operating systems, non-synchronized schedulers, and resource allocators, as well as different positions and distance in the network topology (e.g., in a tree). Thus, the start time and the ongoing performance of each process can differ, usually introducing significant noise into the time dependencies, and when a communication operation is executed, especially a collective one, each process can have different time of arrival (process arrival time, PAT), which in turn can create an **imbalanced process arrival pattern (PAP)**.

We can distinguish two kinds of communication operations: the individual ones when only two processes take part in data transfer, simply sending a message, and the collective ones when, in a non-trivial case, three or more parties are involved performing more complicated functionality such as a data reduction or broadcast. In this article, we consider an all-gather collective operation, which is responsible for gathering processed data from the participants, and then performing the backward distribution, so all of the cooperating processes are going to have the same complete, merged data. Figure 1 presents an example of an all-gather collective execution.

In a typical HPC environment, where message-passing paradigm is used (e.g., with an MPI [14] implementation), the individual communication is usually realized by send-receive operations,

whereas the collectives are represented by broadcast, reduce, all-reduce, scatter, gather, all-gather, and others. Each of these group operations can have multiple implementations (optimized for different environment settings) where different algorithms can be used, e.g., the broadcast can be implemented step-by-step (the root sends messages one by one to the leaves) or using a binomial tree (where the root sends the message to one process, then they both send the messages simultaneously to the others, and so on, creating a binomial tree structure).

In this article, we focus on the optimization of an all-gather operation in the context of resilience to imbalanced PAPs in an HPC environment based on homogeneous compute cluster architecture. Our contribution is as follows:

(1) A description of **Background Disseminated Ring (BDR)** PAP-aware algorithm based on the regular parallel ring algorithm often supplied in MPI implementations, which exploits an auxiliary background thread for early data exchange from faster processes to accelerate the performed all-gather operation.
(2) A description of **Background Sorted Linear synchronized tree with Broadcast (BSLB)** PAP-aware algorithm, built upon an already existing PAP-aware gather algorithm (Background Sorted Linear Synchronized tree) followed by a regular broadcast distributing gathered data to all participating processes.
(3) An experimental evaluation of the above algorithms performed in a typical HPC cluster environment, based on a mini-benchmark executed for larger data sizes (128K or more floats) followed by the result analysis, showing the acceleration of the all-gather operation up to factor 1.9 or 47%.

In the next section, we describe background topics covering collective algorithms resilient to imbalanced PAPs, methods of PAP monitoring and prediction, the used computation model, the evaluation criteria, and other currently used algorithms implementing all-gather operations. Afterwards, in Section 3, we present two novel PAP-aware algorithms: BDR and BSLB, including their pseudo-code, a detailed description, and an execution example. Then, in Section 4, an experimental evaluation of the proposed solutions is provided, including a description of mini-benchmark and the test setup, followed by the result analysis. In the last section, the final remarks and the perceived future works are provided.

2 BACKGROUND

The study on **process arrival patterns (PAPs)** in a typical HPC environment, with two examples of a cluster architecture, was presented in Reference [4]. A PAP is defined as a tuple $(a_0, a_1, \ldots a_{P-1})$, where a_i is a measured **process arrival time (PAT)** for process i, while P is the number of all processes participating in a given collective operation. Similarly, a **process exit pattern (PEP)** can be defined as a tuple $(f_0, f_1, \ldots f_{P-1})$, where f_i is the time when process i finishes the operation.

An **imbalanced** PAP is defined as a PAP where the worst case imbalance time:

$$\omega = \max_{i \in \langle 0, P-1 \rangle} (a_i) - \min_{i \in \langle 0, P-1 \rangle} (a_i) \tag{1}$$

is equal to or greater than the time to communicate one message in the collective operation: τ, thus $\frac{\omega}{\tau} \geq 1$. Otherwise, in the case when $\frac{\omega}{\tau} < 1$, we define the PAP as **balanced**. Figure 2 presents an example of process arrival and exit patterns as well as worst imbalance time of a single collective operation.

The analysis of a number of existing applications—benchmarks, which used extensively MPI collective communication [4], showed a ubiquity of imbalance PAP occurrence and their high

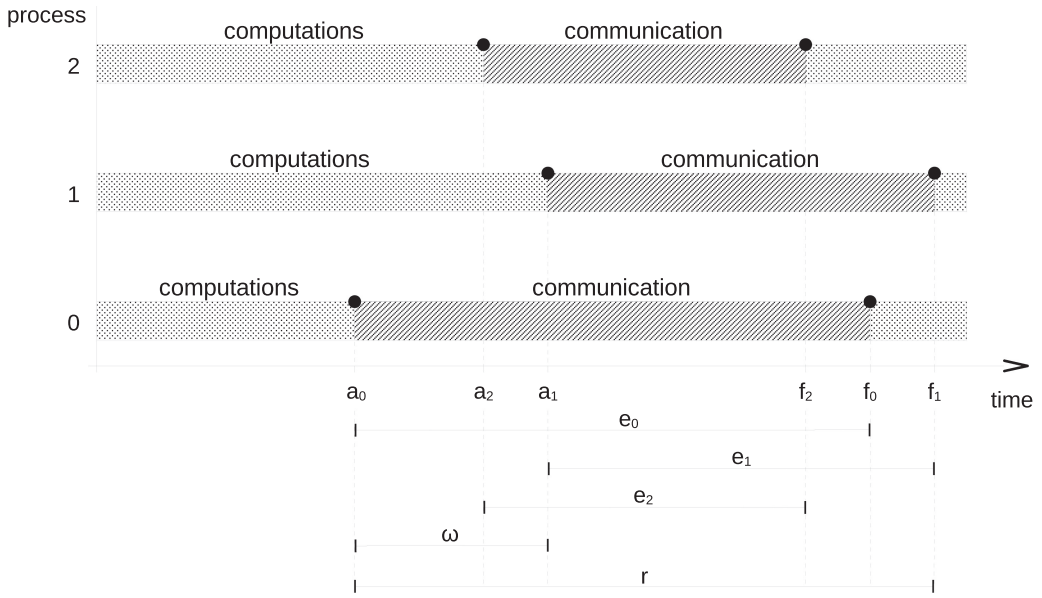

Fig. 2. Example of a process arrival pattern: (a_0, a_1, a_2), a process exit pattern: (f_0, f_1, f_2), a run time $r = f_1 - a_0$, and worst case imbalance time: $\omega = a_1 - a_0$, where y-axis labels: $\langle 0, 1, 2 \rangle$ indicate process identifiers $(P = 3)$, a_i, f_i, and $e_i = f_i - a_i$ are, respectively, arrival, exit, and elapsed times of a process i for the performed collective communication operation. In this case average elapsed time can be derived as: $\bar{e} = \frac{e_0+e_1+e_2}{3}$.

influence on the overall performance. Moreover, the further investigation into the sources of such behavior indicated that this phenomenon usually cannot be controlled directly by programmers, and the imbalances are going to occur in any typical HPC environment.

In the related works, we can distinguish two main approaches for improving performance of the parallel applications suffering by imbalanced PAPs' occurrences. The first one is based on introduction a self-tuning platform with a library of various collective algorithms, where the platform selects the most proper one according to its measured run time. Such solution was used in a STAR-MPI framework [4], where the performance of all-to-all operation was significantly improved. The second one is based on a direct algorithm design, where new PAP-aware algorithms are proposed for every specific collective operation.

2.1 Algorithms Resilient to Imbalanced PAPs in the Related Works

In Reference [18], Patarasuk et al. presented two PAP-aware broadcast algorithms dedicated for large messages, one for the non-blocking model of message-passing (`arrival_nb`) and the other for the blocking one (`arrival_b`). They performed a theoretical analysis and experimental research, both showing better performance of the proposed algorithms in comparison to the existing ones.

In Reference [26], Qian et al. presented three algorithms resilient to imbalanced PAPs, for all-to-all (`PAP_Direct, PAP_Shm_Direct`) and all-gather (`PAP_Direct`) collective operations. They are dedicated and strongly dependent on the underlying hardware, namely, on the InfiniBand [5] interconnecting network. They were based on the Direct and the SMP-aware algorithms proposed in Reference [25] and utilized specific RDMA mechanisms for early process arrival notification. The performed experiments showed the advantage of the new algorithms over their regular counterparts achieving an acceleration of factor 1.44.

Table 1. State-of-the-art Algorithms Resilient to Imbalanced PAPs

Operation	Algorithms
All-gather	`PAP_Direct` [26]
All-reduce	adaptive tree [11], PRR, SLT [20]
All-to-all	`PAP_Direct, PAP_Shm_Direct` [26]
Barrier	adaptive tree [11]
Broadcast	`arrival_b, arrival_nb` [18]
Gather	SLS, BSLS, SBN, BSBN [21]
Reduce	local redirect [12], Clairvoyant [13, 23]
Scatter	SLN, BSLN, SBN, BSBN [21]

Similarly to the above, Mamidala et al. presented [11] barrier and all-reduce PAP-aware algorithms based on hardware support provided by an InfiniBand [5] network with multicast and RDMA operations. Both algorithms are based on adaptive tree gather/reduce followed by broadcast, where a special token is placed in the tree root and can be moved to other processes in the case of their late arrival. The approach shows the best results when only one leaf process is delayed.

Two reduce algorithms resilient to imbalanced PAPs were presented in References [12] and [13]; both are based on the binomial tree approach, with the logarithmic complexity. The former (local redirect) was used for data vectors requiring atomicity and the latter (Clairvoyant) for the ones that can be split into segments, and their exchange can be scheduled before the network transfer. In both cases the theoretical analysis and experimental results showed the advantage of the newer approach. Recently, in Reference [23], we presented a new version of Clairvoyant algorithm, including a set of performance and usability improvements.

In Reference [20], we proposed two new PAP-aware algorithms for all-reduce collective operation. The first one extended typical linear tree approach: **Sorted Linear Tree (SLT)**, while the other was based on parallel ring: **Pre-Reduced Ring (PRR)**. In the experiments, PRR showed better performance in comparison to SLT as well as the regular algorithms used in typical MPI implementations. In Reference [24], PRR was tested in a geographically distributed compute cluster where two sets of nodes were connected by a 900-kilometer-long, fast (100 Gbps), optical fiber network. The results showed performance improvements for the PAP-aware algorithm; however, it was outperformed by hierarchical solutions typical for this type of environments.

Finally, in Reference [21], we presented a collection of eight algorithms resilient to imbalanced PAPs: **(Background) Sorted LiNear (BSLS/SLS)** tree for scatter, **(Background) Sorted Linear Synchronized tree (SLS/BSLS)** for gather, and **(Background) Sorted BiNomial tree (SBN/BSBN)** for both: scatter and gather operations. They were based on communication trees constructed according to the **process arrival times (PATs)** and optionally were supported by auxiliary background communication performed in anticipation of the group operation behavior. Similarly to the previous results, the experiments showed the advantage of the proposed approach in imbalanced PAPs' environment.

Table 1 summarizes state-of-the-art algorithms designed for addressing imbalanced PAPs. We can notice the lack of the all-gather hardware independent solution (`PAP_Direct` [26] is based on InfiniBand [5] built-in mechanisms), which we attempt to cover in this article.

2.2 PAP Monitoring and Model of Processing

All of the aforementioned algorithms require some *a priori* knowledge about arrival times of the cooperating processes (see Table 2). Some of them [12, 18] gather PAP-related data during its

Table 2. Approaches for PAP On-line Estimation

PAP detector	Operation–algorithms
Its own messaging—part of the algorithm	Broadcast: `arrival_b, arrival_nb` [18] Reduce: local redirect [12]
Hardware support (InfiniBand [5])	All-gather: `PAP_Direct` [26] All-reduce: adaptive tree [11] All-to-all: `PAP_Direct, PAP_Shm_Direct` [26]
Auxiliary background thread	All-reduce: SLT, PRR [20] Gather: SLS, BSLS, SBN, BSBN [21] Reduce: Clairvoyant [23] Scatter: SLN, BSLN, SBN, BSBN [21]
Static analysis or SMA (suggestion)	Reduce: Clairvoyant [13]

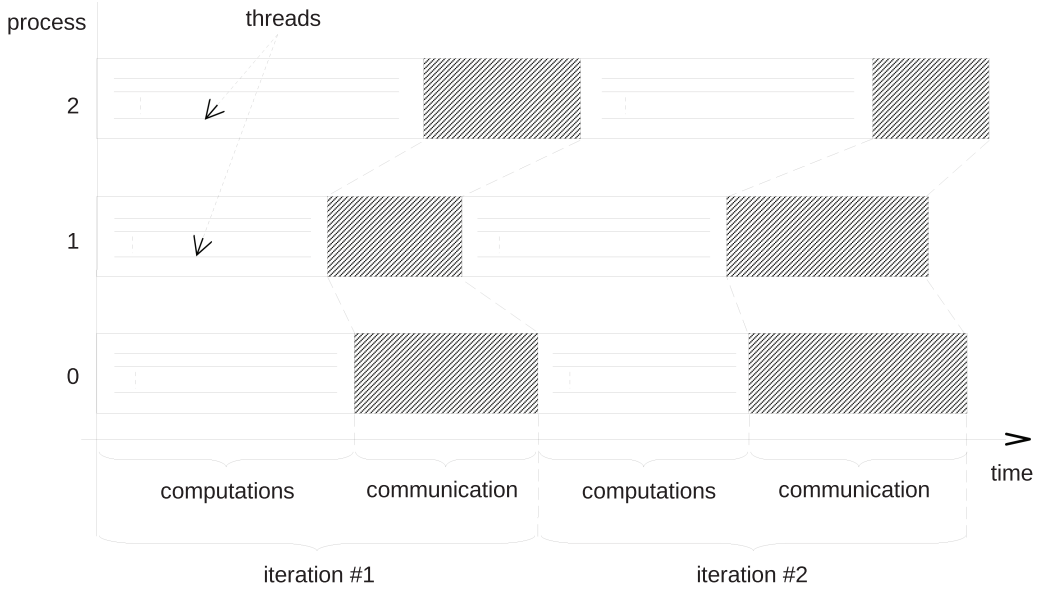

Fig. 3. Visualization of an iterative processing model.

execution with additional messaging at the beginning of the communication. The others [11, 26] use the network hardware support, particularly InfiniBand built-in mechanisms, to resolve which counterpart process has already arrived. In Reference [13], we can find suggestion that the PAP characteristics for each process are stable during the program run time, thus a static analysis of the source code or some statistical methods like **simple moving averages (SMA)** can be used for PATs estimation.

In our previous works [20, 21, 23, 24], as well as in this article, we assume an **iterative model of the processing**, i.e., the parallel program executes consecutively iterations and each iteration consists of two phases: computations, where the processes perform calculations, and communication, with the data containing results of the computation phase being exchanged by the cooperating processes. The former causes heavy CPU load, in opposition to the latter when the network is intensively used.

Moreover, as it is presented in Figure 3, we postulate that each process consists of a number of concurrent threads, which cooperate with each other using shared memory mechanisms. Thus, the

internode parallelism is implemented by processes and the intranode parallelism by the threads. Furthermore, we assume that each process is executed by only one node and each compute node hosts only one process. We can generalize the model, releasing the last assumption, and allow to assign more processes to the same node. However, in such case, we would need to provide a solution considering non-unified send-receive times between processes (e.g., by the introduction of a process hierarchy, similar to Reference [24]), yet we consider it a separate problem, out of the scope of this article, that we plan to cover in future works.

Thus, for the PAP monitoring and prediction purposes, we use an auxiliary background thread (in this and our previous works implemented as an additional pThread [19] instance), mainly active in the computation phase when the application code does not utilize the network interfaces. The thread uses the unloaded network to exchange progress data, distributing information about other processes arrival times over the collaborating processes. Moreover, it is also possible to employ the thread to do additional, network-related activities to improve the overall efficiency of the processing, e.g., exchange preliminary collective data [21] or "warm up" the connections between the processes [20]. The solution presented in this article uses such a thread to support the proposed all-gather algorithms, including preliminary data exchange within so-called pre-steps performing typical MPI communication.

2.3 Algorithm Evaluation Criteria

In the related works, we can distinguish two different approaches to evaluate the proposed PAP-aware algorithms. The first one is related to a performance assessment of a real application being optimized using the proposed solution. However, in the real world, the applications differ in many aspects of their execution and sometimes improvement to one parallel program can cause performance degradation in another one. Thus, in such a situation it is necessary to provide some other means to be used for the evaluation.

The other approach is based on performing some mini-benchmark that emulates a whole range of possible configurations with various process numbers, data sizes, and randomly generated or fixed delays in PATs. The collective operations are executed consecutively multiple times and the measurements of communication time are performed. After the experiments, the results are compared and the algorithms are evaluated according to a specific criterion. We can distinguish the following evaluation criteria, provided in the previous researches:

- average elapsed time [4]: It is the average time, in which all the cooperating processes stay in the collective operation (see an example in Figure 2):

$$\bar{e} = \frac{1}{P}\sum_{i=0}^{P-1} e_i = \frac{1}{P}\sum_{i=0}^{P-1} f_i - a_i. \tag{2}$$

- run time [13]: It is a period from the first process arrival till the last one leaving the evaluated collective operation (see an example in Figure 2):

$$r = \max_{i\in\langle 0,P-1\rangle}(f_i) - \min_{i\in\langle 0,P-1\rangle}(a_i). \tag{3}$$

- absorption time [13], representing the absorbed part of the imbalance, mitigated by the evaluated algorithm:

$$A = r' - r + \omega, \tag{4}$$

where r' is a run time of the evaluated algorithm for perfectly balanced (flat) PAP, i.e., $(0, 0, \ldots, 0)$, r is the run time (see Equation (3)), and ω is the worst case imbalance time (see Equation (1)).

It can be noted, that for asymmetric operations, e.g., broadcast, where one of the processes is emphasized, usually called a root, the run time (see Equation (3)) or even the root process elapsed time seems to be appropriate criteria for the algorithm estimation. However, for symmetrical operations, e.g., all-gather, where all processes are equal, the elapsed time (see Equation (2)) seems to more correlated with the total application execution time. Thus, similarly to Reference [20] in this article, we use the elapsed time measurements for the algorithm evaluation.

2.4 Regular All-gather Algorithms in Use

We are going to focus on larger data sizes (at least 128K of floats or 0.5 MB) and for such an assumption, the most popular open source MPI implementations: OpenMPI [17] and MPICH [15], whose source code we analyzed, use the following algorithms: **neighbor exchange (NEX)**, **parallel ring (RING)**, and additionally, a **linear gather with broadcast (LNBC)**.

The NEX algorithm [1] is used by OpenMPI for even numbers of the collaborating processes, where each process exchanges its data repetitively with the right and left neighbor, consecutively increasing the data size. In the case of MPICH and for OpenMPI when there is the odd process number, a RING algorithm is used, where each process sends its data to the next neighbor, but in this case the size of the sent messages is fixed. Additionally, for the reference purposes, we utilize LNBC algorithm, which is implemented by a regular gather operation, based on a linear tree algorithm, combined with a consecutive broadcast.

There is a number of other all-gather algorithms based on a binomial tree, e.g., Bruck algorithm [28]; however, the aforementioned MPI implementations typically use them for smaller data sizes, and in general, these algorithms seem to be ineffective for the larger ones; thus, we do not take them into consideration in this research.

Moreover, recently there are some other works related to optimization of all-gather algorithms, where the additional, specific constraints are considered, e.g., in Reference [8], Kang et al. provided a solution for intergroup cooperation, rapidly accelerating data gathering between two disjointed process sets; in Reference [29], Zhou et al. analyzed and improved all-gather behavior for multi-/many-core processor in compute clusters; in Reference [2], Cho el al. presented an efficient communication library, with an all-gather implementation, for distributed deep learning that is highly optimized for popular GPU-based platforms. Our work extends this filed with imbalanced PAP related study, where similarly to these researches the all-gather is the mainly considered collective operation.

3 PROPOSED SOLUTION

In this article, we propose two novel algorithms implementing an all-gather collective operation, optimizing its performance in an imbalanced PAP environment based on a compute, homogeneous cluster architecture. As we mentioned in Section 2.2, both of them work under the assumption of the iterative processing model and use the **auxiliary, background thread** to perform some additional actions on their behalf, even before the all-gather operation is called from the main program.

The proposed approach to the PAP monitoring and the PAT prediction requires certain methods for checking the progress of the computations. For this purpose, we introduced three special functions: `PAT_ProcessingStart()`, `PAT_ProcessingEnd()`, and `PAT_Edge()` to be called back and report the progress of the computations. `PAT_ProcessingStart()` and `PAT_ProcessingEnd()` invocations signal the beginning and the end of the performed computation phase in a given process, respectively. While a `PAT_Edge()` call indicates an intermediate milestone, marked by a given percentage of the elapsed computations.

Listing 1 presents the auxiliary background thread pseudo-code. While working, the thread waits for the `PAT_Edge()` call (line 5), and along with the computation phase start time provided by the main program during `PAT_ProcessingStart()` function call and stored in variable *processingStart*, the thread is able to calculate the current and estimate the end processing times (lines 6–7). The current time is indicated by the standard MPI function: `MPI_Wtime()`. Internally, the approximation is made using a **linear extrapolation**; however, in future works, a different approach can be used, e.g., a solution based on EMA (exponential moving averages) over consecutive iterations, even without the necessity of providing the exact progress value. Afterwards, the PAT estimations are exchanged between the cooperating processes (line 8).

LISTING 1: Pseudo-code of the auxiliary, background thread

```
1:  variables:
2:  working ▷ true as long as the process is operating
3:  processingStart ▷ set to the start time of the computation phase

4:  while working do
5:      wait for PAT_Edge() function call
6:      calculate time since processing start as MPI_Wtime()-processingStart
7:      estimate the PAT for the current process
8:      exchange PATs with other processes
9:      perform additional algorithm-related activities
10: end while
```

Apart of the above monitoring-related activities, the auxiliary thread can be used to perform additional algorithm-specific actions (line 9) that are grouped in the algorithms' pseudo-code as **background** blocks, in opposition to **foreground** blocks, which are executed in the main program thread as a result of the corresponding collective operation calls.

3.1 Background Disseminated Ring

The first proposed algorithm: **Background Disseminated Ring (BDR)**, presented in Listing 2, is based on a regular **parallel ring (RING)** approach with additional operations performed in the background within a scope of the auxiliary thread, even before the all-gather call. The idea of the algorithm is to use the background thread to prepare a schedule of exchanging messages (lines 22–46) and execute the pre-steps (lines 47–52), and then, after calling all-gather operation by the main application code, the rest of the communication is performed in the foreground (lines 56–67).

Each process provides its own part of data, i.e., a segment, denoted as *inp* (line 5), which are going to be transferred to the other ones. We assume that the transfer time (point-to-point send-receive) of such a segment over the network, between any two processes, is constant and equals τ, can be measured or estimated before the algorithm execution and provided as an input parameter (line 6). In a full-duplex network, every process can send and receive data at the same time; thus, we can split the algorithm execution into steps in which a process can send or receive one segment. We define the steps performed between arrivals of the first and the last process as pre-steps.

The pre-steps part of the schedule enables dissemination of the data provided by the faster processes (with earlier PATs) and is going to be finished when the last process arrives. In the first loop (lines 22–33), the segments from the faster processes are scheduled to be disseminated to the other processes, possibly the ones still in the computation phase. In case more sending processes are going to transfer a segment to the same destination, the slower one is preferred in the inner loop (lines 23).

LISTING 2: Pseudo-code of background disseminated ring (BDR) algorithm

input:
P ▷ number of processes/nodes (one process per node)
a_r ▷ arrival time of process r
id ▷ id of the current process (rank)
inp ▷ data (one segment) to be sent by the current process, available in the foreground
τ ▷ period of time in which transfer of one data segment is performed
output:
out_r ▷ entire data (all segments) to be gathered, where r is the source process id
variables:
pa_i ▷ process ids, initially sorted (descending) according to the arrival time: a_i
$psNo_q$ ▷ number of pre-steps of process q, initially set to $\lfloor \frac{\max_r(a_r)-a_q}{\tau} \rfloor$
$prevSent_r$ ▷ number of processes to which process r already sent its data, initially 0
$rsched_r^s$ ▷ schedule: receive operation to be performed by process r in step s, initially $\emptyset$
$ssched_r^s$ ▷ schedule: send operation to be performed by process r in step s, initially $\emptyset$
$rstep_r$ ▷ last step in $rsched$ for process r
$sstep_r$ ▷ last step in $ssched$ for process r
rc ▷ receiver process id
seg ▷ origin process id of the data segment to be transferred
s ▷ step of the communication

background:
▷ setting up the schedule
for $s = 0$ to $(\max_q psNo_q) - 1$ **do** ▷ schedule the pre-steps
  for $\forall r \in pa$ **do** ▷ iterate over processes from the slowest one
    $rc = (r - 1 - prevSent_r + P) \mod P$
    if $\{psNo_r \geq (\max_q psNo_q) - s\} \wedge \{prevSent_r < P - 1\} \wedge \{rsched_{rc}^s = \emptyset\}$ **then**
      schedule send(inp, rc) to $ssched_r^s$
      $sstep_r = s$
      schedule out_r=recv(r) to $rsched_{rc}^s$
      $rstep_{rc} = s$
      $prevSent_r = prevSent_r + 1$
    end if
  end for
end for
for $i = 0$ to $P - 1$ **do** ▷ schedule the ring
  $rc = (i + 1) \mod P$
  for $j = 0$ to $P - 1$ **do**
    $seg = (i - j + P) \mod P$
    if $prevSent_{seg} + j < P - 1$ **then**
      schedule send(out_{seg}, rc) to $ssched_i^{s+j}$
      $sstep_r = s + j$
      schedule out_{seg}=recv(i) to $rsched_{rc}^{s+j}$
      $rstep_{rc} = s + j$
    end if
  end for
end for

46: ▹perform the background part of the schedule
47: $s = 0$
48: **while** $ssched^s_{id} = \emptyset$ **do**
49: execute recv from $rsched^s_{id}$ as a blocking operation
50: $s = s + 1$
51: **end while**

52: **foreground:**
53: $out_{id} = inp$ ▹ set up the output for the current process
54: wait to finish the background operations
55: ▹perform the rest operations from the schedule
56: **while** $\{s \leq sstep_{id}\} \vee \{s \leq rstep_{id}\}$ **do**
57: **if** $rsched^s_{id} \neq \emptyset$ **then**
58: execute recv from $rsched^s_{id}$ as a non-blocking operation
59: **end if**
60: **if** $ssched^s_{id} \neq \emptyset$ **then**
61: execute send from $ssched^s_{id}$ as a blocking operation
62: **end if**
63: **if** $rsched^s_{id} \neq \emptyset$ **then**
64: wait for recv from $rsched^s_{id}$ to be finished
65: **end if**
66: $s = s + 1$
67: **end while**

The receivers are assigned in *id* descending order, one-by-one, beginning from sending process $id - 1$ with modulo P wrapping (line 24). A segment is going to be disseminated if the following conditions are met (line 25): (i) it is already available in the sending process, (ii) there is at least one process that the segment was not already sent to, and (iii) the destination process does not have scheduled any receive operation from another process in the actual step. The schedule is split into two independent send and receive parts: *ssched* and *rsched*, respectively, and is filled in greedy manner—no adjustments are made after the operation is scheduled (lines 26–29).

In the second loop (lines 34–45), executed after all pre-steps are planned, the pending segments are scheduled to be distributed basing on the RING algorithm. The outer loop (line 34) iterates over the processes where receivers are assigned in *id* ascending order, one-by-one beginning from sending process $id + 1$ with modulo P wrapping (line 35), while the inner loop (lines 36–44) schedules the concrete segments. The main difference to the RING algorithm is that the segment transmission finishes earlier for processes with the already disseminated segments (the condition in line 38). The implementation of the above scheduling procedure (lines 21–45) includes additional performance optimization, skipped in the pseudo-code for the sake of simplicity, causing the empty steps without any operation to be performed, not to be included into the schedule.

The final schedule is executed in two parts, the first one (lines 47–51) in the background, before the input data *inp* is available for the current process, and the other (lines 56–67) in the foreground as a part of the all-gather operation call made by the main computing program. Thus, the background part finishes when the schedule indicates the first send operation (line 48) and is continued after the input data is assigned as a part of the result (line 53). The full-duplex transmission is exploited, with the blocking send (line 61) and the receive operation split into two parts: the actual call (line 58) and wait for its finish (line 64). Unfortunately, the necessity of sending just-received

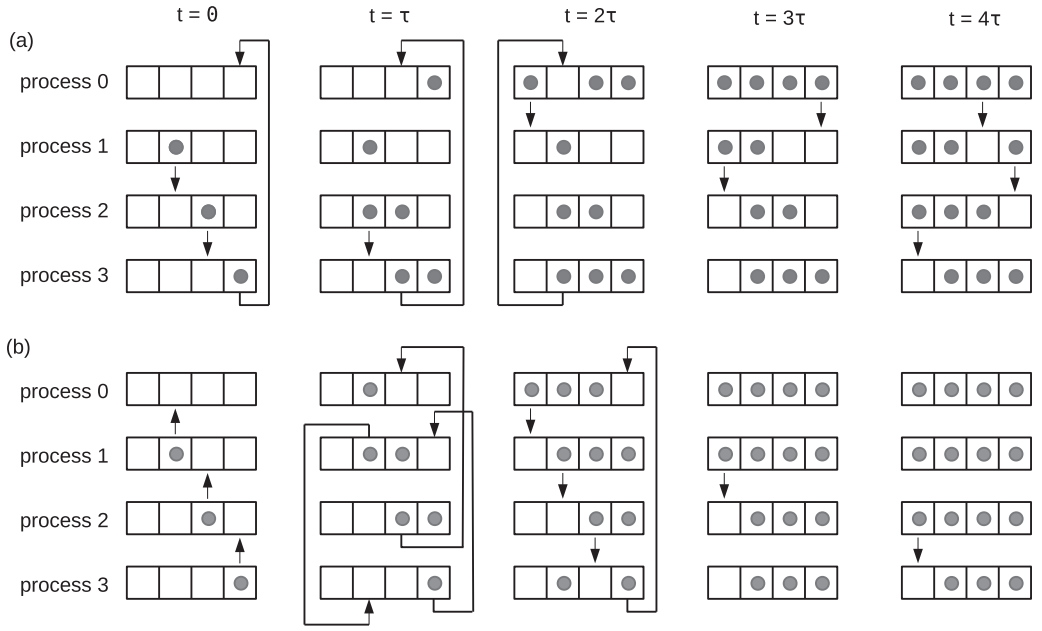

Fig. 4. Impact of an imbalanced PAP (delayed process 0 by 2τ) on ring based all-gather algorithm executions: (a) regular parallel ring (RING), (b) background disseminated ring (BDR). An arrow represents a segment transfer, and a dot indicates segment data being delivered.

data segment, a feature inherited from the regular parallel ring algorithm, prevents us from more advanced optimization, where more non-blocking receive operations would be collected for one wait call (`MPI_Waitall()` in the MPI implementation).

The computational complexity of the scheduling depends on two factors: the number of the cooperating process P, and the maximal number of pre-steps over all processes, which can be calculated as: $S = \lfloor \frac{\omega}{\tau} \rfloor$ (note that $\frac{\omega}{\tau}$ is the imbalance factor definition [4]). Thus, after the evaluation of the scheduling loops (lines 22–33 and 34–45), the complexity can be estimated as $O(SP + P^2)$.

An example of the BDR execution is presented in Figure 4(b), where the process 0 is delayed by $a_0 = 2\tau$, while the others arrived at the same time $a_{1-3} = 0$. This enables the processes 1–3 to perform 2 pre-steps, which are used to disseminate their input data. Therefore, the segments from processes 1 and 2 are available for process 0 at the moment the all-gather is called ($t = 2\tau$); and the process finishes this operation in just one step. In comparison to RING (see Figure 4(a)), where processes 1–3 spend $e_{1-3} = 5\tau$ and process 0 spends $e_0 = 3\tau$ periods of time in the call, the BSLS decrease them to $e_0 = 1\tau$ for process 0 and $e_2 = 4\tau$ for process 2. What can be denoted by the average elapsed time ($\bar{e}$) decreasing from 5τ to 4τ (the reduction of 20%).

Let us consider a situation when the PAT estimation would go totally wrong; this means the slowest process would be perceived as the first one arrived. In such a case, the BDR algorithm would prevent the data transfer until the delayed process arrives, which could be a reason for extending the time of the operation for the pre-steps execution. Thus, in comparison to the regular parallel ring, the average elapsed times could increase up to $(P - 2) \times \tau$.

3.2 Background Sorted Linear Synchronized Tree with Broadcast

In general, an all-gather operation can be implemented using a combination of a simple gather, where the whole data is collected by an arbitrary selected root process, and a consecutive

broadcast, which disseminates the gathered data from the root to other processes. Usually this approach is less effective than specific all-gather focused algorithms, e.g., NEX or RING; however, in some cases, unexpectedly, it can work faster than such solutions. Thus, the second proposed algorithm, **Background Sorted Linear synchronized tree with Broadcast (BSLB)**, presented in Listing 3, is based on the existing **Background Sorted Linear Synchronized tree (BSLS)** gather-only algorithm (line 8), which we proposed in Reference [21], followed by a regular broadcast algorithm (line 9) provided by the used MPI implementation.

In BSLS tree gather algorithm, a non-root process splits its data vector into two segments, where the first one is significantly smaller. Then, such a process waits for a signal from the root, which is used to trigger the transmission of this short segment to inform back that the rest of the data is ready to send. The root process sends the signals to the processes according to their PAT order. Thus, the faster processes can start the transmission ahead of the later ones, and additionally this procedure can decrease the risk of the network contingency. Moreover, the receiving of the data is performed by the root process in the background, using the auxiliary thread, even before the main program finishes the computations, which increases the resilience of the algorithm to the delays in this process.

Thus, similarly to the previous algorithm, BSLS uses the auxiliary thread during the computation phase and is split into background and foreground parts. The thread is used to notify the partner processes, which are structured into an arrival time-based binomial tree and receive their data even before the computation phase is finished. However, in opposition to BDR, the algorithm is based on sorting of the processes according to their arrival times, without each segment to be prescheduled. Therefore, it does not need additional information about the transfer time of a single segment: τ.

LISTING 3: Pseudo-code of Background Sorted Linear synchronized tree with Broadcast (BSLB) algorithm

```
1: input:
2: P ▷number of processes/nodes (one process per node)
3: a_r ▷arrival time of process r
4: id ▷id of the current process (rank)
5: inp ▷data (one segment) to be sent by the current process, available in the foreground
6: output:
7: out_r ▷the data (all segments) gathered from the processes, where r is process rank

8: call BSLS gather with P, a_r, id, inp and 0 as root process
9: call MPI BCast with P, rank, 0 as root process and BSLS result as input data
```

Let us consider a situation when the PAT estimation would go totally wrong; this means the slowest process would be perceived as the first one arrived. In such a case, the BSLS part of the algorithm could have worse performance than its counterpart: a regular binomial tree, and in the worst case, it could cause the elapsed times to be longer up to $log_2(P) \times \tau$. More details about this algorithm, including the complexity analysis and an example, can be found in Reference [21].

4 EXPERIMENTAL EVALUATION

The proposed algorithms were implemented using open-source MPI [14] implementation: Open-MPI [17]. Their evaluation is based on average elapsed time measurements using a specially prepared mini-benchmark executed in a controlled experimental HPC environment.

4.1 Mini-benchmark

Listing 4 presents the pseudo-code of a mini-benchmark measuring the algorithm's average elapsed time. The input parameters consist of the total size of gathered data: N, number of iterations to be performed during the test: $itNo$, maximum delay of a process, indicating a possible range of the PATs: $maxDelay$, the $algorithm$ to be evaluated: NEX, LNBC, BSLB, BDR, RING, the number of cooperating processes: P and an id of the current process, equal to the MPI rank value.

LISTING 4: Pseudo-code of the mini-benchmark

```
1:  input:
2:  N ▷ number of elements (floats) in the gathered data
3:  itNo ▷ number of iterations
4:  maxDelay ▷ maximal delay of the processes
5:  algorithm ▷ tested algorithm, one of NEX, LNBC, BSLB, BDR, RING
6:  P ▷ number of processes/nodes
7:  id ▷ process id of the current process (rank): 0...P − 1
8:  output:
9:  avgET ▷ average elapsed time measured for the given algorithm
10: variables:
11: halfTime ▷ 50% of the emulated computation time
12: myET ▷ elapsed time measured in one iteration for the current process
13: sumET ▷ sum of the elapsed times over the iterations and processes, initially 0.0

14: MPI_Init()
15: PAT_Init()
16: PAT_UseAlg(algorithm)
17: for i = 1 to itNo do                                   ▷ repeat itNo times
18:     data = generateRandomData(N/P)
19:     halfTime = 100 ms+random(0 ... maxDelay)/2
20:     MPI_Barrier()                                      ▷ double barrier for process synchronization
21:     MPI_Barrier()
22:     PAT_ProcessingStart()
23:     usleep(halfTime)
24:     PAT_Edge(50%)
25:     usleep(halfTime)
26:     PAT_ProcessingEnd()
27:     startTime = MPI_Wtime()
28:     makeOperation(algorithm, data)                     ▷ call all-gather to be tested
29:     endTime = MPI_Wtime()
30:     checkCorrectness(data)
31:     myET = endTime − startTime
32:     MPI_Allreduce(sumET, myET, MPI_SUM...)
33: end for
34: PAT_Finalize()
35: MPI_Finalize()
36: avgET = sumET/(P×itNo)                                 ▷ measured final result
```

At the beginning each process initiates MPI and PAT libraries by calling `MPI_Init()` and `PAT_Init()`, respectively (lines 14–15). Then, the used algorithm needs to be indicated in

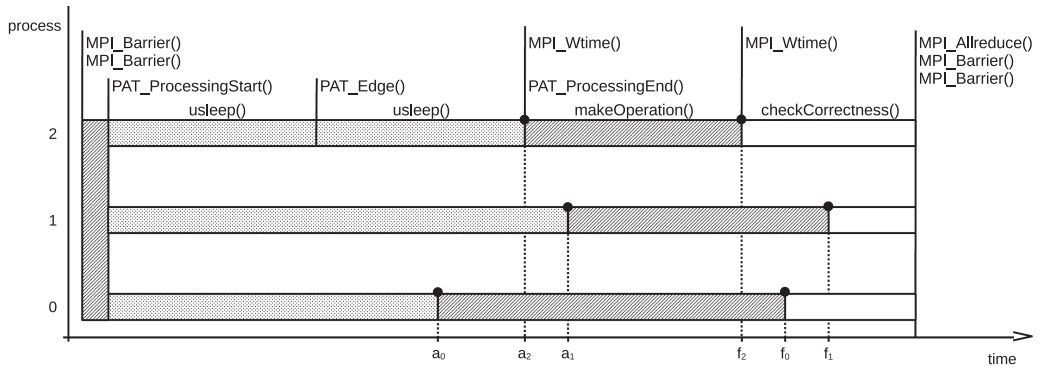

Fig. 5. Example of an iteration execution of the mini-benchmark for 3 processes ($P = 3$) including function calls, and process arrival and exit time measurement points: a_i, f_i, where $i = 0, 1, \ldots P - 1$.

function `PAT_UseAlg()` (line 16) and the tests can be performed in the main loop (lines 17–33), which is repeated $itNo$ times.

During each iteration, new data and the PAT are randomly generated (lines 18–19), and similarly to Intel Benchmark [6] and Reference [13], two `MPI_Barrier()` calls are consecutively invoked to synchronize the physical times of the cooperating processes (lines 20–21). Therefore, the start of the computation phase is indicated to the algorithm implementation, with the execution of the `PAT_ProcessingStart()` function (line 22), and the first part of computations (50%) is mocked by the `usleep()` system call (line 23).

The period of the computations depends directly on a *maxDelay* input parameter, which indicates a maximum time difference between the PATs, and the actual delay is pseudo-randomly generated using the uniform distribution. Afterwards, the internal milestone is marked by calling `PAT_Edge()` function (line 24), enabling the auxiliary, background thread to perform its activities, including the interprocess PATs' estimation. Next, the second half of the computations is mocked and finished with the `PAT_ProcessingEnd()` function call (lines 25-26).

Afterwards, the communication phase is performed with the evaluated all-gather algorithm implementation invoked along with the times measurement using `MPI_Wtime()` function (lines 27–29). Finally, each iteration is finished by checking the result correctness in function `checkCorrectness()` (line 30) and determining the sum of elapsed times calculated and distributed over all benchmarked processes (lines 31–32). Figure 5 presents an example of one iteration execution of the mini-benchmark including performed function calls.

After performing the benchmark iterations, both PAT and MPI libraries are notified about finishing the program (lines 34–35) and the final mini-benchmark result: The average elapsed time is calculated.

4.2 Test Configuration

The experimental evaluation was performed in a special laboratory, which is an isolated partition of the supercomputer Tryton [9]. We decided to separate one rack case with 48 compute nodes to keep the network noise on the lowest possible level. This approach enabled us to control the process delays according to the desired PATs, thus the experiments were able to emulate the assumed PAP conditions. We use Ethernet intraconnect without any hardware acceleration mechanisms so the results and the final conclusions could be as general as possible. The compute nodes were equipped with 2 × Xeon E5-2670 v3 @2.3 GHz CPUs, 128 GB of RAM, and operated under Linux

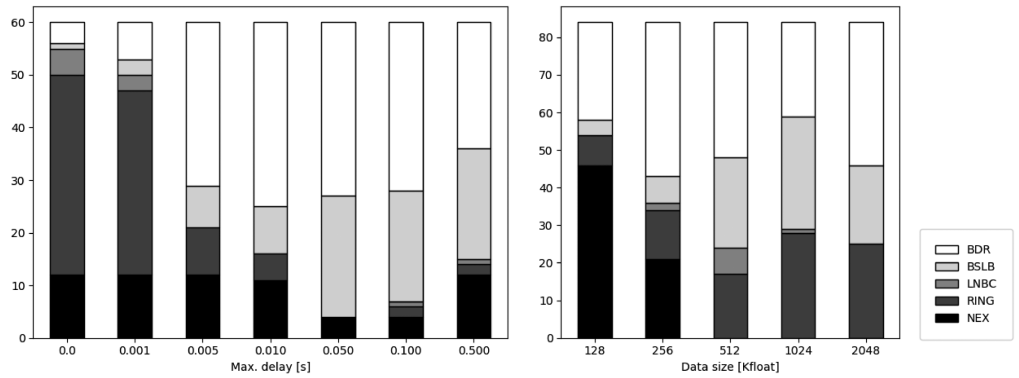

Fig. 6. Mini-benchmark results' distribution over the maximum delay (left-hand side) and the data size (right-hand side) factors. Each bar visualizes the performed experiments, where the tested algorithms' implementations had the shortest average elapsed times.

CentOS v. 6.10 with kernel v. 2.6. The mini-benchmark along with the algorithms was implemented in C language, compiled in 64-bit mode with GCC v. 7.3.0, and linked with OpenMPI v. 3.0.

The aforementioned mini-benchmark was executed for all evaluated algorithms: NEX, LNBC, BSLB, BDR, RING, various PATs, with the randomly generated delays, with maximum ranges: $maxDelay = 0, 1, 5, 10, 50, 100, 500$ milliseconds, different gathered data vector sizes: $N =$ 128K, 256K, 512K, 1M, 2M of floats and incremented number of compute nodes/processes: $P = 4, 8, 12 \ldots 44, 48$. All these combinations resulted in 2,100 experiments, each one with $itNo = 256$ measurement iterations.

4.3 Result Analysis

Tables 3 and 4 present the experimental results of the mini-benchmark execution. For each test case the fastest algorithm is indicated, and when it is one of the proposed PAP-aware algorithms, the acceleration in comparison to the next, best regular algorithm (one of NEX, RING, the ones used in OpenMPI implementation, and LNBC) is provided. According to the assumption presented in Section 2.3, we use the average elapsed time to perform this comparison.

As we could expect, the PAP-aware algorithms outperform the regular (state-of-the-art) ones for configurations affected by imbalanced PAPs. The best relative results were recorded for medium delay values (50 ms) and larger data sizes (over 256K of floats); see Figure 6. The best case when the BDR was faster than the best regular algorithm by factor 1.9 or 47% was observed for 28 nodes/processes, 5 ms maximum delay, and the data size set to 256K of floats.

Figure 7 presents the results of mini-benchmark executions for a wide range of the gathered vector sizes on 36 compute nodes, and the PAT maximum delay equals 100 ms. We can notice that for all algorithms the larger data size causes the increase of the all-gather elapsed times. However, for such conditions, either of the PAP-aware algorithms outperforms regular implementations, with BDR better working for smaller and BSLB for larger data sizes.

Figure 8 presents the experimental results showing the influence of the increasing maximum delay on average run times of the tested algorithms. The chart is drawn up for 32 compute nodes/processes and relatively small data size: 256K of floats. We can observe that for such conditions our PAP-aware BDR outperforms BSLB and regular all-gather algorithms in the whole range of delays.

In comparison to the above case, Figure 9 shows the chart of mini-benchmark results for 48 compute nodes/processes and for a large data size: 2M of floats. In this case, for maximum delays

Table 3. Mini-benchmark Results, Part 1/2

MD →	0	1	5	10	50	100	500
Size ↓				Process/node number: 4			
128	RING	RING	BDR1.15	BDR1.17	BDR1.10	BDR1.07	BDR1.02
256	RING	RING	BDR1.05	BDR1.11	BDR1.14	BDR1.09	BDR1.03
512	BDR1.01	RING	RING	BDR1.07	BDR1.16	BDR1.14	BDR1.04
1,024	RING	RING	RING	RING	BDR1.12	BDR1.17	BDR1.07
2,048	RING	RING	RING	RING	BDR1.07	BDR1.09	BDR1.09
Size ↓				Process/node number: 8			
128	BSLB1.02	BSLB1.01	BSLB1.06	BSLB1.07	BDR1.02	BDR1.04	BDR1.01
256	RING	RING	BSLB1.11	BSLB1.16	BDR1.13	BDR1.10	BDR1.03
512	RING	RING	BDR1.03	BDR1.07	BDR1.10	BDR1.13	BDR1.04
1,024	RING	RING	RING	RING	BDR1.06	BDR1.11	BDR1.08
2,048	RING	BDR1.01	BDR1.01	BDR1.02	BDR1.08	BDR1.12	BDR1.12
Size ↓				Process/node number: 12			
128	RING	RING	BDR1.36	BDR1.23	BDR1.07	BDR1.02	RING
256	RING	RING	BDR1.07	BDR1.07	BDR1.13	BDR1.09	BDR1.03
512	RING	RING	RING	BDR1.06	BSLB1.04	BDR1.14	BDR1.04
1,024	RING	RING	RING	BDR1.03	BDR1.11	BDR1.13	BDR1.07
2,048	BDR1.03	BDR1.02	BDR1.01	BDR1.02	BDR1.07	BDR1.03	BDR1.11
Size ↓				Process/node number: 16			
128	RING	RING	BDR1.43	BDR1.32	BDR1.09	BDR1.02	RING
256	LNBC	BSLB1.02	BSLB1.09	BSLB1.15	BSLB1.01	BSLB1.01	LNBC
512	LNBC	LNBC	BSLB1.05	BSLB1.35	BSLB1.17	BSLB1.09	BSLB1.02
1,024	RING	RING	BDR1.02	RING	BSLB1.37	BSLB1.23	BSLB1.05
2,048	BDR1.02	BDR1.01	RING	BDR1.03	BDR1.04	RING	BDR1.10
Size ↓				Process/node number: 20			
128	NEX	NEX	NEX	BDR1.01	BDR1.01	BDR1.01	NEX
256	RING	RING	BDR1.84	BDR1.61	BDR1.18	BDR1.03	BDR1.01
512	LNBC	LNBC	BSLB1.17	BSLB1.30	BSLB1.14	BSLB1.10	BSLB1.02
1,024	RING	RING	BDR1.02	BDR1.03	BSLB1.30	BSLB1.24	BSLB1.05
2,048	BDR1.01	BDR1.01	RING	BDR1.01	BDR1.04	RING	BDR1.10
Size ↓				Process/node number: 24			
128	NEX	NEX	NEX	NEX	BDR1.01	BDR1.01	NEX
256	RING	RING	BDR1.74	BDR1.74	BDR1.21	BDR1.04	BDR1.01
512	RING	RING	BDR1.03	BSLB1.13	BSLB1.26	BSLB1.15	BSLB1.04
1,024	RING	RING	BDR1.02	BDR1.03	BSLB1.40	BSLB1.24	BSLB1.05
2,048	RING	RING	BDR1.01	BDR1.02	BSLB1.17	BSLB1.23	BSLB1.11

The columns indicate maximum delays (MD) provided in milliseconds for a given experiment, the gathered data sizes (Size) are presented in the rows, and denoted in K of floats. The table is split into sections related to the size of the laboratory cluster used for the evaluation: 4, 8...24 nodes/processes. For each experiment configuration, the fastest algorithm is presented, and in the case of the PAP-aware ones, the speedup factor is additionally shown in comparison to the best regular algorithm execution.

larger than 10 ms, BSLB is the most efficient solution. These results are consistent with the previous observation indicating that BSLB performs better for larger data sizes. The rapid performance boost observed for $maxDelay \geq 50$ ms can be explained by an additional optimization of network communication, where the strict ordering of the exchanged messages decreases the network contingency causing faster data delivery.

Table 4. Mini-benchmark Results, Part 2/2

MD →	0	1	5	10	50	100	500
Size ↓				Process/node number: 28			
128	NEX	NEX	NEX	NEX	BDR1.01	BDR1.01	NEX
256	RING	RING	BDR1.90	BDR1.74	BDR1.24	BDR1.04	BDR1.01
512	RING	RING	BDR1.04	BSLB1.06	BSLB1.25	BSLB1.16	BSLB1.03
1,024	RING	RING	BDR1.02	BDR1.04	BSLB1.24	BSLB1.21	BSLB1.05
2,048	RING	RING	BDR1.01	BDR1.02	BSLB1.16	BSLB1.23	BSLB1.12
Size ↓				Process/node number: 32			
128	NEX	NEX	NEX	NEX	BDR1.01	BDR1.01	NEX
256	RING	BDR1.01	BDR1.58	BDR1.66	BDR1.23	BDR1.04	BDR1.01
512	LNBC	LNBC	BSLB1.13	BSLB1.12	BSLB1.03	LNBC	BSLB1.01
1,024	RING	RING	BDR1.02	BDR1.04	BSLB1.31	BSLB1.20	BSLB1.05
2,048	RING	RING	BDR1.01	BDR1.01	BSLB1.19	BSLB1.16	BSLB1.13
Size ↓				Process/node number: 36			
128	NEX	NEX	NEX	NEX	BDR1.01	NEX	NEX
256	NEX	NEX	NEX	NEX	BDR1.02	BDR1.01	NEX
512	RING	RING	BDR1.40	BDR1.60	BDR1.40	BDR1.08	BDR1.01
1,024	RING	RING	BDR1.02	BDR1.03	BSLB1.32	BSLB1.19	BSLB1.05
2,048	RING	RING	BDR1.01	RING	BSLB1.32	BSLB1.48	BSLB1.17
Size ↓				Process/node number: 40			
128	NEX	NEX	NEX	NEX	NEX	NEX	NEX
256	NEX	NEX	NEX	NEX	BDR1.01	BDR1.01	NEX
512	RING	RING	BDR1.40	BDR1.36	BDR1.41	BDR1.09	BDR1.01
1,024	RING	RING	BDR1.02	BDR1.03	BSLB1.25	BSLB1.19	BSLB1.06
2,048	RING	RING	RING	BDR1.01	BSLB1.33	BSLB1.25	BSLB1.18
Size ↓				Process/node number: 44			
128	NEX	NEX	NEX	NEX	NEX	NEX	NEX
256	NEX	NEX	NEX	NEX	BDR1.01	BDR1.01	NEX
512	RING	BDR1.02	BSLB1.31	BDR1.43	BDR1.40	BDR1.11	BDR1.01
1,024	LNBC	BSLB1.02	BSLB1.17	BSLB1.17	BSLB1.05	BSLB1.04	BSLB1.01
2,048	RING	RING	BDR1.01	BDR1.01	BSLB1.53	BSLB1.48	BSLB1.15
Size ↓				Process/node number: 48			
128	NEX	NEX	NEX	NEX	NEX	NEX	NEX
256	NEX	NEX	NEX	NEX	NEX	BDR1.01	NEX
512	RING	BDR1.04	BDR1.28	BDR1.39	BDR1.37	BDR1.12	BDR1.01
1,024	RING	RING	BDR1.02	BDR1.03	BSLB1.67	BSLB1.41	BSLB1.12
2,048	RING	RING	BDR1.01	BDR1.01	BSLB1.52	BSLB1.46	BSLB1.20

The columns indicate maximum delays (MD) provided in milliseconds for a given experiment, the gathered data sizes (Size) are presented in the rows, and denoted in K of floats. The table is split into sections related to the size of the laboratory cluster used for the evaluation: 28, 32...48 nodes/processes. For each experiment configuration, the fastest algorithm is presented, and in the case of the PAP-aware ones, the speedup factor is additionally shown in comparison to the best regular algorithm execution.

Finally, Figure 10 presents the results of the experiments in the context of scalability, with the experiments performed for 100 ms maximum PAT delay and medium data size: 512K of floats. We can draw a conclusion that the PAP-aware algorithms work better for larger sets of nodes. This is especially noticeable for BDR, which inherited this behavior from its ancestor: **parallel ring**

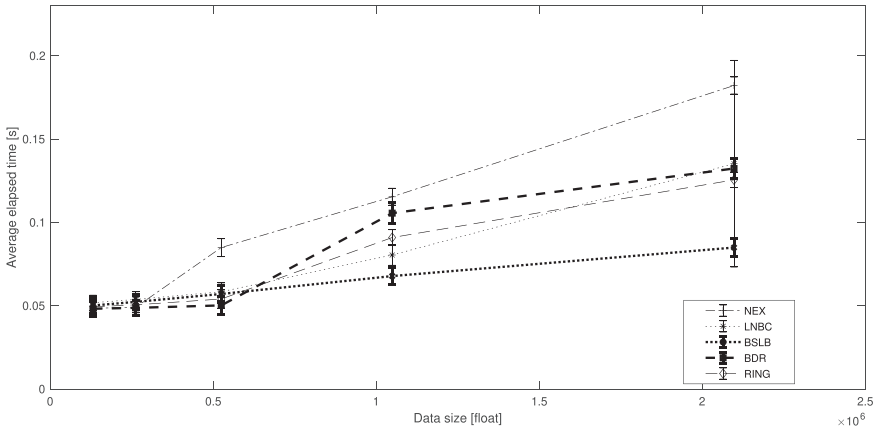

Fig. 7. Mini-benchmark results showing influence of the increasing gathered data size on average elapsed times of the tested algorithms. The experiments were performed on 36 nodes, and the maximum delay of the arrival times equal 100 ms. The error bars are set to $\pm\sigma$ (68% of the measurements for the normal distribution).

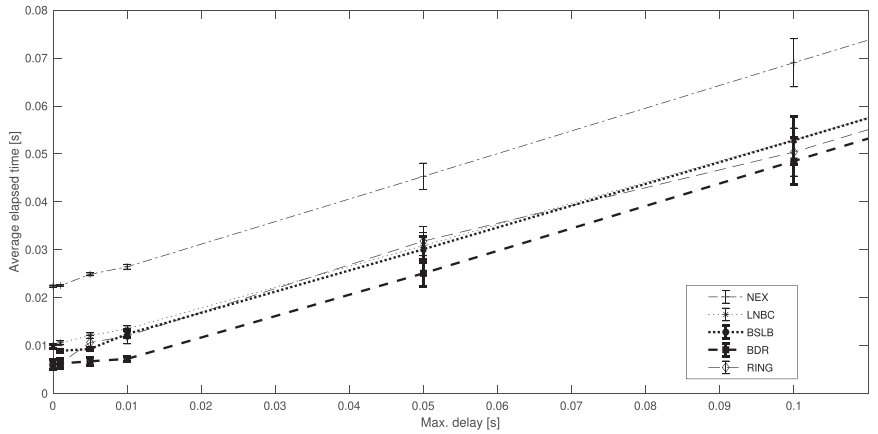

Fig. 8. Mini-benchmark results showing influence of the increasing maximum delay on average elapsed times of the tested algorithms. The experiments were performed on 32 nodes and the message size equal to 256K of floats. The error bars are set to $\pm\sigma$ (68% of the measurements for the normal distribution).

algorithm (RING), which we assume to be caused by the decrease of the data segment size, with the increasing process number, and thus the lowered network load and contingency.

From the statistical point of view, the results seem to be stable, with a small standard deviation in comparison to the measured values. The exceptions can be noticed for smaller PAT delays and data vector sizes, when tree-based algorithms, both the regular and PAP-aware ones, have more volatile elapsed times (see Figure 9).

We can conclude the result analysis with the remark that the PAP-aware algorithm works well for larger compute node/process numbers and significant PAT delays. The BDR is better for smaller and BSLB for larger data sizes, with both having the advantage over the regular algorithms.

5 FINAL REMARKS

In this article, we presented two novel all-gather algorithms resilient to the imbalanced PAPs. They were tested and evaluated using a typical for HPC compute cluster architecture with the

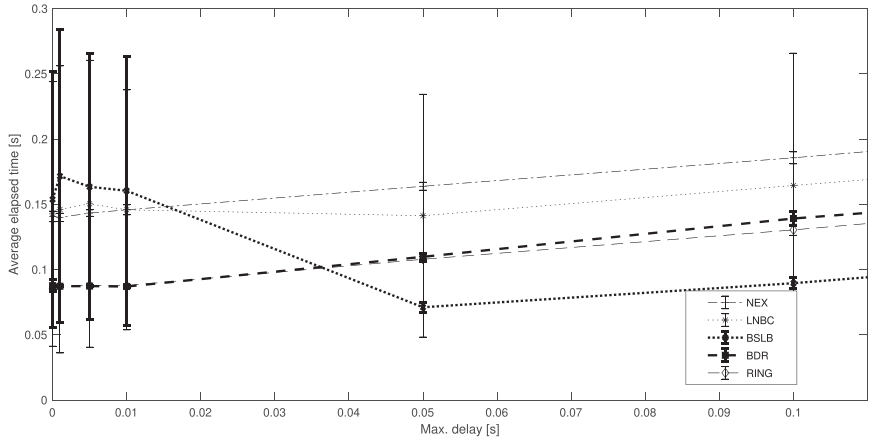

Fig. 9. Mini-benchmark results showing influence of the increasing maximum delay on average elapsed times of the tested algorithms. The experiments were performed on 48 nodes for the message size equal to 2M of floats. The error bars are set to $\pm\sigma$ (68% of the measurements for the normal distribution).

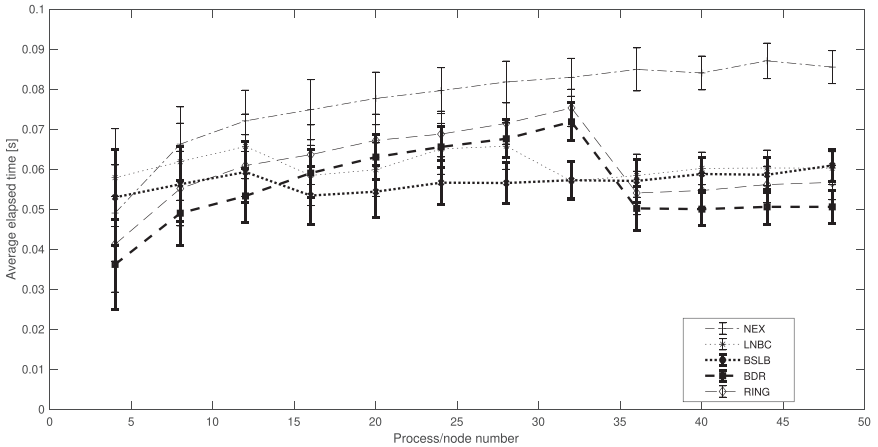

Fig. 10. Mini-benchmark scalability results, showing influence of the increasing number of processes/nodes on average elapsed times of the tested algorithms. The experiments were performed for the maximum delay of the arrival times equal to 100 ms and 512K of floats gathered data size. The error bars are set to $\pm\sigma$ (68% of the measurements for the normal distribution).

homogeneous nodes connected by 1 Gbps Ethernet interconnect. This comprehensive verification was based on a proposed mini-benchmark, which was executed over 2,000 times to perform an experimental comparison with the regular solutions widely used in the state-of-the-art open source MPI implementations.

The performed tests proved that both of the proposed algorithms present good results in the imbalanced PAP environment, showing a significant reduction of the all-gather elapsed times up to factor 1.9 or 47%. As we could expect, the proposed solution not only optimizes the performance for more imbalanced PAPs, but also works better with the larger data sizes. Finally, we can observe that the algorithms are well scalable with the increasing numbers of the compute nodes. Similarly to our previous works [20, 21, 23, 24], this approach can be directly used for HPC applications supported by iterative process model, such as various scientific simulations.

Our future works of the imbalanced PAP subject will cover the following topics:

- design of PAP-aware algorithms dedicated for specific architectures, e.g., hierarchical or long-distance distributed compute clusters [24],
- research on the PAP-aware solutions optimized for energy consumption factors [10],
- introduction of the proposed approach into a larger set of applications, especially in HPC environments,
- the evaluation of the ultra-scale HPC environments for imbalanced PAPs using advanced models and simulation tools, e.g., References [3, 22].

We strongly believe that the rapidly developing parallel systems require PAP-aware solutions to improve their performance and scalability.

ACKNOWLEDGMENTS

To prof. Henryk Krawczyk, my mentor, for his help, advice, and guidelines in the world of science, and to the whole team of Centre of Informatics—Tricity Academic Supercomputer & networK (CI TASK) in Gdansk University of Technology.

REFERENCES

[1] Jing Chen, Linbo Zhang, Yunquan Zhang, and Wei Yuan. 2005. Performance evaluation of allgather algorithms on terascale Linux cluster with fast Ethernet. In *Proceedings of the 8th International Conference on High-performance Computing in Asia-Pacific Region (HPCASIA'05)*, Vol. 2005. IEEE. DOI: https://doi.org/10.1109/HPCASIA.2005.75

[2] Minsik Cho, Ulrich Finkler, M. Serrano, David Kung, and H. Hunter. 2019. BlueConnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. *IBM J. Res. Dev.* 63, 6 (Nov. 2019), 1:1–1:11. DOI: https://doi.org/10.1147/JRD.2019.2947013

[3] Paweł Czarnul, Jarosław Kuchta, Mariusz Matuszek, Jerzy Proficz, Paweł Rościszewski, Michał Wójcik, and Julian Szymański. 2017. MERPSYS: An environment for simulation of parallel application execution on large scale HPC systems. *Simul. Model. Pract. Theor.* 77 (Sep. 2017), 124–140. DOI: https://doi.org/10.1016/j.simpat.2017.05.009

[4] Ahmad Faraj, Pitch Patarasuk, and Xin Yuan. 2008. A study of process arrival patterns for MPI collective operations. *Int. J. Parallel Prog.* 36, 6 (Dec. 2008), 543–570. DOI: https://doi.org/10.1007/s10766-008-0070-9

[5] InfiniBand. [n.d.]. InfiniBand Trade Assosciation. Retrieved on 2021 from https://www.infinibandta.org/.

[6] Intel-MPI-benchmark. [n.d.]. Intel MPI benchmark. Retrieved on 2021 from https://software.intel.com/content/www/us/en/develop/articles/intel-mpi-benchmarks.html.

[7] IntelMPI. [n.d.]. Intel MPI library homepage. Retrieved from https://software.intel.com/en-us/mpi-library.

[8] Qiao Kang, Jesper Larsson Träff, Reda Al-Bahrani, Ankit Agrawal, Alok Choudhary, and Wei-keng Liao. 2019. Scalable algorithms for MPI intergroup allgather and allgatherv. *Parallel Comput.* 85 (July 2019), 220–230. DOI: https://doi.org/10.1016/j.parco.2019.04.015

[9] Henryk Krawczyk, Michał Nykiel, and Jerzy Proficz. 2015. Tryton supercomputer capabilities for analysis of massive data streams. *Polish Marit. Res.* 22, 3 (Jan. 2015), 99–104. DOI: https://doi.org/10.1515/pomr-2015-0062

[10] Adam Krzywaniak, Pawel Czarnul, and Jerzy Proficz. 2019. Extended investigation of performance-energy trade-offs under power capping in HPC environments. In *Proceedings of the International Conference on High-performance Computing & Simulation (HPCS'19)*. IEEE, 440–447. DOI: https://doi.org/10.1109/HPCS48598.2019.9188149

[11] A. R. Mamidala, Jiuxing Liu, and D. K. Panda. 2004. Efficient barrier and allreduce on infiniband clusters using multicast and adaptive algorithms. In *Proceedings of the IEEE International Conference on Cluster Computing*. IEEE, 135–144. DOI: https://doi.org/10.1109/CLUSTR.2004.1392611

[12] Petar Marendić, Jan Lemeire, Tom Haber, Dean Vučinić, and Peter Schelkens. 2012. An investigation into the performance of reduction algorithms under load imbalance. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 7484 LNCS. Springer Berlin, 439–450. DOI: https://doi.org/10.1007/978-3-642-32820-6_44

[13] P. Marendic, J. Lemeire, D. Vucinic, and P. Schelkens. 2016. A novel MPI reduction algorithm resilient to imbalances in process arrival times. *J. Supercomput.* (2016). DOI: https://doi.org/10.1007/s11227-016-1707-x

[14] MPI. [n.d.]. The standarization forum for Messsage Passing Interface (MPI). Retrieved from http://mpi-forum.org/.

[15] MPICH. [n.d.]. MPICH High-Performance Portable MPI. Retrieved on 2021 from https://www.mpich.org/.

[16] OpenMP. [n.d.]. The OpenMP API specification for parallel programming. Retrieved on 2021 from https://www.openmp.org/.

[17] OpenMPI. [n.d.]. Open MPI: Open Source High Performance Computing. Retrieved on 2021 from https://www.open-mpi.org/.

[18] Pitch Patarasuk and Xin Yuan. 2008. Efficient MPI Bcast across different process arrival patterns. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 1–11. DOI: https://doi.org/10.1109/IPDPS.2008.4536308

[19] POSIX threads. [n.d.]. POSIX threads programming. Retrieved on 2021 from https://computing.llnl.gov/tutorials/pthreads/.

[20] Jerzy Proficz. 2018. Improving all-reduce collective operations for imbalanced process arrival patterns. *J. Supercomput.* 74, 7 (July 2018), 3071–3092. DOI: https://doi.org/10.1007/s11227-018-2356-z

[21] Jerzy Proficz. 2020. Process arrival pattern aware algorithms for acceleration of scatter and gather operations. *Cluster Comput.* 23, 4 (Dec. 2020), 2735–2751. DOI: https://doi.org/10.1007/s10586-019-03040-x

[22] Jerzy Proficz and Paweł Czarnul. 2016. Performance and power-aware modeling of MPI Applications for cluster computing. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 9574. Springer, Cham, 199–209. DOI: https://doi.org/10.1007/978-3-319-32152-3_19

[23] Jerzy Proficz and Krzysztof M. Ocetkiewicz. 2020. Improving Clairvoyant: reduction algorithm resilient to imbalanced process arrival patterns. *J. Supercomput.* (Nov. 2020). DOI: https://doi.org/10.1007/s11227-020-03499-1

[24] Jerzy Proficz, Piotr Sumionka, Jarosław Skomiał, Marcin Semeniuk, Karol Niedzielewski, and Maciej Walczak. 2020. Investigation into MPI all-reduce performance in a distributed cluster with consideration of imbalanced process arrival patterns. In *Advanced Information Networking and Applications. AINA 2020. Advances in Intelligent Systems and Computing*, L. Barolli, F. Amato, F. Moscato, T. Enokido, and M. Takizawa (Eds.). Vol. 1151. Springer, Cham, 817–829. DOI: https://doi.org/10.1007/978-3-030-44041-1_72

[25] Ying Qian and Ahmad Afsahi. 2008. Efficient shared memory and RDMA based collectives on multi-rail QsNetII SMP clusters. *Cluster Comput.* 11, 4 (Dec. 2008), 341–354. DOI: https://doi.org/10.1007/s10586-008-0065-8

[26] Ying Qian and Ahmad Afsahi. 2011. Process arrival pattern aware Alltoall and Allgather on InfiniBand clusters. *Int. J. Parallel Prog.* 39, 4 (Aug. 2011), 473–493. DOI: https://doi.org/10.1007/s10766-010-0152-3

[27] SLURM.[n.d.]. SLURM Workload Manager. Retrieved on 2021 from https://slurm.schedmd.com/.

[28] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of collective communication operations in MPICH. *Int. J. High-perf. Comput. Applic.* 19, 1 (Feb. 2005), 49–66. DOI: https://doi.org/10.1177/1094342005051521

[29] Huan Zhou, José Gracia, and Ralf Schneider. 2019. MPI collectives for multi-core clusters. In *Proceedings of the 48th International Conference on Parallel Processing: Workshops*. ACM, New York, NY, 1–10. DOI: https://doi.org/10.1145/3339186.3339199

Received December 2020; revised March 2021; accepted April 2021