Process arrival pattern aware algorithms for acceleration of scatter and gather operations

Jerzy Proficz[1]

© The Author(s) 2020

Abstract

Imbalanced process arrival patterns (PAPs) are ubiquitous in many parallel and distributed systems, especially in HPC ones. The collective operations, e.g. in MPI, are designed for equal process arrival times, and are not optimized for deviations in their appearance. We propose eight new PAP-aware algorithms for the scatter and gather operations. They are binomial or linear tree adaptations introducing additional process ordering and (in some cases) additional activities in a special background thread. The solution was implemented using one of the most popular open source MPI compliant library (OpenMPI), and evaluated in a typical HPC environment using a specially developed benchmark as well as a real application: FFT. The experimental results show a significant advantage of the proposed approach over the default OpenMPI implementation, showing good scalability and high performance with the FFT acceleration for the communication run time: 16.7% and for the total application execution time: 3.3%.

Keywords Scatter · Gather · Process arrival pattern · MPI · PAP-aware algorithm

1 Introduction

Collective operations, in a non-trivial case, require participation of three or more processes, which are supposed to synchronize their activities or exchange data. The usual assumption in designing such algorithms is that all processes join the operation at the same time [9]. In reality process arrival time (PAT) differs for each process, implying the occurrence of the so-called imbalanced process arrival patterns (PAP), which sometimes can cause a significant delay in the performed computations. Thus, it is desirable to provide mechanisms for imbalanced PAP detection and design algorithms exploiting such information to compensate for the above imbalance.

The scatter collective operation is usually used for split and distribution of the data between the cooperating processes. As input it accepts a vector of data (usually numerical values, e.g. float) provided by an arbitrary chosen root process, and as a result it returns the corresponding data partition to each of the processes participating in the operation. The gather is the opposite operation, where all processes provide input data vectors and the root process receives their concatenation. Both of these operations are defined in the Message Passing Interface (MPI) standard [12] and are provided in its implementations.

The contribution of this paper is eight new algorithms for scatter/gather collective operations exploiting the imbalanced PAPs to increase the efficiency of communication. For scatter operation we propose: (i) Sorted LINear tree (SLIN), (ii) Sorted BiNomial tree (SBN), (iii) Background Sorted LiNear tree (BSLN) and (iv) Background Sorted BiNomial tree (BSBN). Similarly for the gather operation we propose: (v) Sorted Linear Synchronized tree (SLS), (vi) Sorted Binomial tree (SBN), (vii) Background Sorted Linear Synchronized tree (BSLS) and (viii) Background Sorted Binomial tree (BSBN).

For each algorithm we provide the description including its pseudocode, complexity analysis for communication (using Hockney model [13]) and computation, as well as we present the results of the experiments performed in a real compute cluster environment, showing a performance

✉ Jerzy Proficz
   j.proficz@task.gda.pl

1   Centre of Informatics - Tricity Academic Supercomputer & networK (CI TASK), Gdansk University of Technology, 11/12 Gabriela Narutowicza Street, 80-233 Gdansk, Poland

gain of the scatter/gather operations in comparison to the default (state-of-the-art) OpenMPI implementation. Finally we prove the usability of the approach by providing a practical use case improving the performance of the Fast Fourier Transform parallel implementation, with the acceleration of communication by 16.7% and total application execution by 3.3%.

The following section describes the already existing works related to PAP-aware algorithms and scatter/gather collective operations, Sect. 3 provides background information about the subject and the next section presents the proposed PAP-aware algorithms. Section 5 presents the developed benchmark and the experimental results of its performance, followed by a section showing a real-life application: improved parallel FFT processing and its evaluation in a real HPC environment. In the last section, conclusions and planned future works are described. Finally, in Appendix, we present extended results of the experiments, showing additional measurement parameters.

2 Related works

The following subsection presents the works related to the scatter/gather algorithms used in the currently available open-source MPI implementations, and the next subsection describes the current state-of-the-art of the PAP-aware algorithms for various collective operations.

2.1 Scatter/gather algorithms

Scatter and gather collectives are often used together, the typical example can be spotted in the master–slave processing, when the scatter operation distributes data to the slaves, where the actual computing is performed, and the gather operation is used for transferring the results to the master process. However, the above schema is not mandatory, e.g. the result gathering can be performed by another operation e.g. reduce.

In the state-of-the-art implementations, the following algorithms are used: (i) binomial (BNOM) tree, (ii) linear (LIN) tree and, for gather only, the modification of the latter: (iii) linear synchronized (LS) tree. In the case of the binomial tree, in each step of scatter operation, any process, which already received the data vector, splits it into two equal parts and sends one of them to a process which is still waiting, thus the communication is finished after $\lceil \log_2(P) \rceil$ steps, where P is the process number. The gather operation works similarly, but the data flow is performed in the opposite way.

In scatter linear tree algorithm, the root process sends the split data vector directly to each process one by one, thus the communications is finished after P steps. The

gather version of this algorithm works similarly, with the opposite data flow direction. In the linear synchronized tree gather algorithm the above mechanism is extended by segmentation of the gathered vector pieces, where each non-root process sends short part of the data vector and waits for the synchronization message from the root. This mechanism enables coordination of the order of the received messages by the root process. Table 1 presents the summary of the algorithms used in two most popular open-source MPI implementations: OpenMPI [10] and MPICH [11].

There is a number of studies related to the scatter/gather optimization in which we can distinguish two directions of the research. The first, where the irregular forms of the operations are studied (ScatterV/GatherV) and the second, considering different homogeneous models of the architecture. In both cases the authors are usually focused on different communication tree construction to decrease overlap in the differences in communication times, e.g. [7, 27, 28] or introduce some hierarchical structures, e.g. [14].

2.2 Optimization of collectives with imbalanced PAPs

The first PAP definition with its theoretical and experimental analysis, showing the ubiquity of imbalanced PAPs was provided in [9]. The authors proposed to use their STAR-MPI [8] framework, equipped with a vast collection of various collective algorithms, to improve the performance of all-to-all MPI operations, executed for typical HPC benchmarks, i.e. LAMMPS [17] and NAS [2] kernels. The solution assumed the imbalanced PAPs to occur repeatably in the same code/process location allowing their slow evolution. Thus, only rare PAT monitoring data

Table 1 Scatter/gather algorithms implemented in OpenMPI [10] and MPICH [11]

Data size (N)	Process No (P)	OpenMPI	MPICH
Scatter			
Large	Any	LIN	
Small	Large	BNOM	BNOM
Small	Small	LIN	
Gather			
Large	Any	LS	
Medium	Large/medium	BNOM	
Medium	Small	LIN	BNOM
Small	Large	BNOM	
Small	Medium/small	LIN	

BNOM binomial tree, *LIN* linear tree, *LS* linear synchronized tree

exchange between cooperating processes is required for determining the PATs.

In [21] Patarasuk et al. presented two broadcast algorithms: *arrival_b* and *arrival_nb*, optimized for imbalanced PAPs and used for large data vectors, the former is dedicated for blocking and the latter for non-blocking message-passing systems. Both solutions sort the cooperating processes by their arrival times and transfer broadcasted data to the faster processes first, so the processes do not need to wait for data delivery to the slower ones. The monitoring data, required for sorting the PATs, are exchanged at the beginning of the operation using non-blocking (*arrival_nb*) or blocking (*arrival_b*) send/receive operations. The authors performed the positive algorithms' evaluation using a developed benchmark, which enabled comparison to other broadcast implementations using a 16-nodes compute cluster.

A new PAP-aware all-to-all and all-gather algorithms exploiting a specific feature of the InfiniBand [26] interconnecting network used for HPC clusters was proposed in [25]. The feature enabled the faster processes to be aware about the current status of the slower ones. Thus, the data could be exchanged between the earlier participants first. The proposed solution also introduced a hierarchical data flow in the case when a subset of the processes was placed in the same node. The results of the experiments performed in a test environment (4 nodes InfiniBand cluster) showed the performance improvement in comparison to the typical all-to-all and all-gather algorithms.

In [20] and [19] Marendic et al. presented work on reduce algorithms considering imbalanced PAPs. The solution covers both cases: when the PATs are initially known and when they need to be discovered during exchange of the reduced data between the cooperating processes. The algorithms were tested using a specially developed mini-benchmarks comparing their performance with other typically used reduction algorithms. The results showed a significant improvement in performance, especially in the case when the reduced data could be partitioned into segments and the PATs were known a priori (the Clairvoyant algorithm).

A hardware-based multicast improving a recursive doubling algorithm for imbalanced PAPs was proposed in [1]. The approach assumed introduction of additional tagging of the exchanged messages with some kind of vector clocks, enabling the receiving process to be aware about a communication path the data had already performed, what could help in relaxing tight dependencies between received and sent messages. The approach was implemented for all-reduce operation and used a NetFPGA [18], open-source, programmable Ethernet-based device. The experiments performed on 8-node cluster showed up to 26% improvement over the standard recursive doubling algorithm.

In [23] we proposed two new, hardware agnostic, all-reduce algorithms optimized for imbalanced PAP occurrence, the solution included a PAP detection mechanism based on progress monitoring by an additional background thread placed in every process participating in the collective operation. A benchmark evaluating the performance of the algorithms was described and experimental results comparing with other typically used algorithms were provided. Finally a real case: machine learning of a deep neural network was practically examined, showing the performance advantage of the method: 21% acceleration of the communication phase.

To the best knowledge of the author there are no prior works covering PAP-aware algorithms for the scatter or gather operations. Thus, the comparison of the proposed solution is performed against typically used algorithms described in Sect. 2.1.

3 Background

The proposed model is an extension of a model described in [23], which considers parallel processing in a homogeneous compute cluster environment and focuses on the process arrival and exit patterns. The aforementioned work is extended by a definition of operation run time, which is helpful in evaluation of non-symmetric collective algorithms, i.e. broadcast, scatter, gather etc.

We assume the compute cluster consists of a set of homogeneous compute nodes interconnected by a fast network. The communication and synchronization between the nodes is accomplished using the typical message-passing model, in contrast to the intranode parallelism which utilizes shared memory. Each node runs one process and each process can contains multiple control threads, the processes use direct and collective communication operations (e.g. MPI [12]) and the threads use shared memory with some synchronization primitives (e.g. POSIX threads [4] or OpenMP [6]).

In the proposed model, the processes cooperate to solve some problem, we assume their algorithm is iterative, i.e. it consists of the consecutive phases: computation and communication repeated multiple times. During the computation phase the threads of a process can cooperate with each other, however the data exchange between the processes (or nodes) is performed only during the communication phase. Thus, we assume that the processor load is higher during the former phase and the network traffic is more intensive in the latter. A typical examples of such behavior can be observed in many machine learning applications where each iteration causes the underlying model to better approximate the reality. Moreover, we assume that the computations are deterministic, i.e. it is possible to indicate

a point during the computation phase, where the some specific part (e.g. 50%) of the involved calculations is already finished.

Process arrival time (PAT) is the time when the process joins the communication phase after finishing the computations, for a process i we denote its PAT as a_i. Furthermore, we define a process arrival pattern (PAP) as a tuple $(a_0, a_1, \ldots a_{P-1})$, where P is the total number of processes participating in the collective operation. Additionally we can also define process exit pattern (PEP) as a tuple $(f_0, f_1, \ldots f_{P-1})$, where f_i is the time when process i finishes the communication phase [9]. An example of the above patterns is presented in Fig. 1.

Imbalanced PAPs, as described in [9], are ubiquitous in many HPC systems, especially in clusters. Even for highly homogeneous environments they appear very often, being rather norm than exception. The expected, natural source of the imbalances is the non-equal distribution of the computations to the nodes, where, even for perfectly balanced task assignment, the PATs are not equal. The supposed cause of the imbalances is the non-deterministic behavior of both computation and communication parts of the processing, which seems to be beyond the control of application developers. We assume that the low level causes of this behavior are related to such phenomena as computational noise [22], asymmetric placement of nodes in the network topology or specific architecture features of the involved communication devices.

For collective algorithm evaluation, performing a given operation in iteration i with a measured pair of PAP and PEP, we can define the following measurements: *run time* [19]:
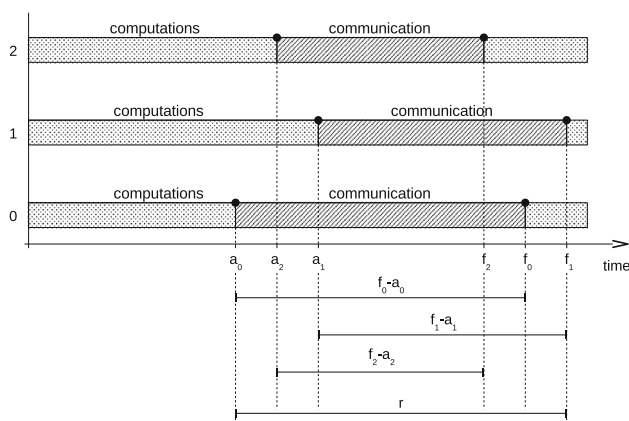
$$r_i = \max_j(f_j) - \min_j(a_j) \tag{1}$$

and *average elapsed time* [9]:

$$\bar{e}_i = \frac{1}{P}\sum_{j=0}^{P-1}(f_j - a_j) \tag{2}$$

where $j \in \langle 0, 1 \ldots P-1 \rangle$. For the sake of simplicity, in the rest of the paper, we drop the i index, with the assumption that r and $\bar{e}$ represent corresponding mean values over all iterations in a particularly executed program.

The former shows how long it takes from starting communication in the first process to finish it in the last one, and the latter shows how much time is used for communication by each process, see example in Fig. 1.

Thus, using Hockney model [13], where α is a startup time of sending a single message, and β is a fraction depending of the sent data size, in the case of a perfectly balanced, flat PAP (all the PATs are equal: $a_0 = a_1 = \cdots = a_{P-1}$), the run time and the elapsed time of the scatter/gather LIN algorithms can be estimated as in the following equations:

$$r^{LIN} = (P-1)\left(\alpha + \frac{N}{P}\beta\right) \tag{3}$$

$$\begin{aligned}\bar{e}^{LIN} &= \frac{1}{P}(P-1)\left(\alpha + \frac{N}{P}\beta\right) + \frac{1}{P}\sum_{i=1}^{P-1}i\left(\alpha + \frac{N}{P}\beta\right)\\ &= \frac{(P+2)(P-1)\left(\alpha + \frac{N}{P}\beta\right)}{2P}\end{aligned} \tag{4}$$

where N is the size of a data vector to be scattered/gathered. For the LS gather algorithm, under the same assumptions, the run time and the elapsed time are as follows:

$$r^{LS} = (P-1)\left(3\alpha + \frac{N}{P}\beta\right) \tag{5}$$

$$\begin{aligned}\bar{e}^{LS} &= \frac{1}{P}(P-1)\left(3\alpha + \frac{N}{P}\beta\right) + \frac{1}{P}\sum_{i=1}^{P-1}i\left(3\alpha + \frac{N}{P}\beta\right)\\ &= \frac{(P+2)(P-1)\left(3\alpha + \frac{N}{P}\beta\right)}{2P}\end{aligned} \tag{6}$$

For the BNOM scatter algorithm, with additional assumption that the process number is a power of 2: $P = 2^k, k \in \mathbb{N}$, the run time and the elapsed time have the same estimation, and can be denoted as follows:

$$r^{BN} = \bar{e}^{BNs} = \log_2(P)\alpha + \frac{(P-1)N}{P}\beta \tag{7}$$

Finally, for the BNOM gather algorithm the run time is the same as for the scatter one (Eq. 7), however the elapsed time is as follows:

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

$$
\begin{aligned}
\bar{e}^{BNg} &= \frac{1}{P}\left\{ P\left(\alpha + \beta\frac{N}{P}\right) + \frac{P}{2}\left(\alpha + 2\beta\frac{N}{P}\right) + \cdots \right. \\
&\quad \left. + 2\left(\alpha + \frac{P}{2}\beta\frac{N}{P}\right)\right\} \\
&= \frac{1}{P}\sum_{i=0}^{\log_2(P)-1}\frac{P}{2^i}\left(\alpha + 2^i\beta\frac{N}{P}\right) \\
&= \frac{2(P-1)\alpha + \log_2(P)\beta N}{P}
\end{aligned}
\tag{8}
$$

Since the Hockney model [13] does not take into consideration a possible contingency over the underlying interconnecting network, which can have limited bandwidth, the above estimations should be perceived as the lowest bound. Thus, although the BNOM algorithms show the lowest communication complexity, they also require the highest bandwidth and in the case of the large data vector size they can have worse performance than the linear trees. Thus, some MPI implementations use different algorithms for specific data vector sizes and cooperating process numbers, see Table 1 for more details related to MPICH [11] and OpenMPI [10].

We can notice, that in the case of symmetrical processing, where all cooperating processes perform the same send, receive and compute activities (e.g. all-reduce operation) the elapsed time seems to be more accurate for the evaluation, however for operations where one process is emphasized (e.g. the root process for scatter operation), the run time seems to be more correlated with the total application execution time. Thus, for scatter/gather evaluation we rather use the run times. However, in all performed experiments, the trends of both elapsed and run times are similar, but for the sake of the research scrupulousness, we provide the elapsed time results in Appendix.

In [23] we proposed an iterative model of computations along with an additional, background thread for monitoring purposes. The thread performs the data exchange during the computation phase (when the network is usually underused), and provides the information about the progress of the computations to all cooperating processes. The computation progress is reported by the computation threads using a special callback function: PAT_Edge(), called after reaching a specific point of processing, e.g. when 50% of the computations is finished. The background thread can also be used for some additional activities like network warmup before the communication phase, or as we propose in this paper, to exchange the messages with the actual collective data, if they are already available for a given process, even during the computation phase, see Fig 2.

The background thread pseudo-code is presented in Fig. 3. The thread uses the *working* variable as an indicator
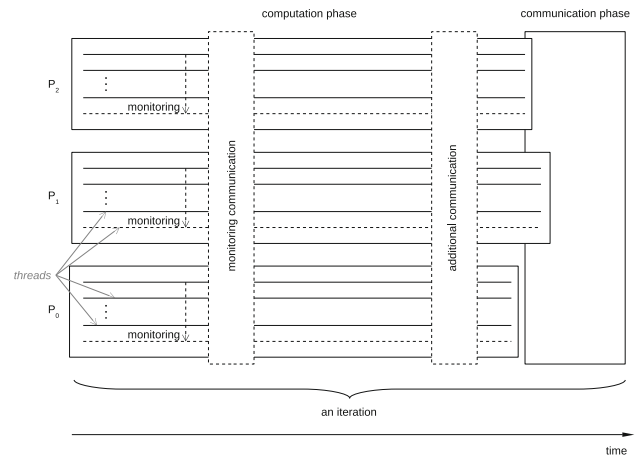

Fig. 2 Iterative model of computations enhanced with the auxiliary communication and thread (marked by dashed line) for monitoring and additional data exchange

```
variables:
working – true as long as the thread is operating

1.  while working
2.     wait for PAT_Edge() function call
3.     measure time since processing start
4.     estimate the PAP for the current process
5.     exchange PAPs with other processes using MPI_Allgather()
6.     perform additional algorithm related activities
```

Fig. 3 Pseudo-code of the background thread

of its activity, it is set up in the PAT_Init() function call and switched off in PAT_Finalize(). The main thread loop is executed in parallel with the main algorithm iterations (see Fig. 4 for a pseudo-code of typical PAP-aware operation usage), where the computation and communication phases are constantly repeated. The thread starts its main activities after the PAT_Edge() function call, when a significant (usually 50%) part of the computations is already performed. It estimates the computation phase finish time, i.e. process arrival time (PAP) for the current iteration, and exchanges it with other processes using MPI_Allgather

```
1.  MPI_Init()
2.  PAT_Init()
3.  PAT_UseAlg(PAT_Gather_XXXX)
4.  for i := 1 to K  // algorithm iterations
5.     MPI_Scatter(. . .)
6.     PAT_ProcessingStart()
7.     computations: first 50%
8.     PAT_Edge(50%)
9.     computations: second 50%
10.    PAT_ProcessingEnd()
11.    PAT_Gather_XXXX(. . .)
12. PAT_Finalize()
13. MPI_Finalize()
```

Fig. 4 Pseudo-code of typical usage of a PAP-aware gather operation, with an XXXX algorithm implementation

```
input parameters:
P − number of processes/nodes (one process per node)
a_r − arrival time of process r
d_x − input data segments to be scattered, an x segment is to
be sent to the rank x process
rank − the rank of the process
root − the rank of the root process

variables:
result − a result data segment
pt_r − an array with rank identifiers, r = 0, 1, . . . , P − 1

background:
1.  if rank ≠ root then
2.     result := receive()

foreground:
3.  if rank = root then
4.     sort(pt_r) according to a_r
5.     for i := 0 to P − 1
6.        if i ≠ root then
7.           send(segment: d_{pt_i}, to: pt_i)
8.        else
9.           result := d_{rank}
10. else
11.    wait() for the background actions to be finished
12. return result
```

Fig. 6 Pseudo-code of scatter Background Sorted LiNear (BSLN) tree algorithm

Fig. 6), where additionally the background thread of a receiving process handles the incoming messages (lines 1–2) despite the fact that the computation phase can still go on. The code of the root process remains the same as in the SLN algorithm (lines 4–9), and foreground actions of a leaf process are limited to waiting for the background receive of the data (line 11). Such approach enables the delayed processes to not block the root if it already finished the computation phase.

The communication complexity of the SLIN and BSLN scatter algorithms, for the perfectly balanced, flat PAP is the same as for LIN (see Eqs. 3 and 4), and the computation complexity can be estimated as $O(P \log P))$, due to sorting the processes by their PATs. However, potentially both algorithms work much faster in case of an imbalanced PAP, where the sorted and background send-receive operations can accelerate the scatter in the earlier (according to their PATs) processes.

Let's analyze run times of the above algorithms considering a situation when one process, either the first receiver (id: 1) or the root (id: 0), is delayed. In the first case $a_1 > a_0$ and $a_0 = a_2 = \cdots = a_{P-1}$, and the run times can be estimated as in the following equations:

$$r^{LIN(id:1)} = a_1 - a_0 + r^{LIN} \tag{9}$$

$$r^{SLIN(id:1)} = \max\left(a_1 - a_0 + \alpha + \frac{N}{P}\beta, \ r^{LIN}\right) \tag{10}$$

$$r^{BSLN(id:1)} = \max\left(a_1 - a_0, \ r^{LIN}\right) \tag{11}$$

where r^{LIN} is defined in Eq. 3. In the latter case, when the root process is delayed: $a_0 > a_1$ and $a_1 = a_2 = \cdots = a_{P-1}$, regardless of the used algorithm, all other processes need to wait. Thus the run times are equal and can be estimated as in the following equation:

$$r^{LIN(id:0)} = r^{SLIN(id:0)} = r^{BSLN(id:0)} = a_0 - a_1 + r^{LIN} \tag{12}$$

The next proposed algorithm: scatter Sorted BiNomial (SBN) tree (see the pseudo-code in Fig. 7) is based on the regular binomial tree, extended by sorting the processes by their PAPs (lines 1–3), in such a way that the faster processes are involved in the earlier phases of the algorithm. This approach requires swapping and shuffling the segments of the data vector, according to the PAT order (lines 7–8). Afterwards the typical binary tree operations are executed (lines 11–18).

The scatter Background Sorted BiNomial (BSBN) tree algorithm (see the pseudo-code in Fig. 8) extends SBN by

```
input parameters:
P − number of processes/nodes (one process per node)
a_r − arrival time of process r
d_x − input data segments to be scattered, an x segment is to
be sent to the rank x process
rank − the rank of the process
root − the rank of the root process

variables:
nrank − a new rank of the process after the arrival sorting
result − a result data segment
pt_r − an array with rank identifiers, r = 0, 1, . . . , P − 1
received − true if the process already received the data

1.  swap(pt_0, pt_root)
2.  sort(pt_r) according to a_r
3.  shuffle(pt_r) according to bit reverse, e.g. rev(0111) = 1110
4.  nrank := index of rank in pt_r
5.  k := 2^{⌈log_2 P⌉}

6.  if rank = root then
7.     shuffle(d_r) according to pt_r
8.     shuffle(d_r) according to bit reverse, e.g. rev(0111) = 1110

9.  i := k/2
10. received := false

11. while i > 0 do
12.    if received ∨ rank = root then
13.       send(segments: (d_i, . . . d_{2i−1}), to: pt_{nrank+i})
14.    if (¬received) ∧ (nrank & i ≠ 0) then
15.       d := receive()
16.       received := true
17.    i := i/2
18. return d_0
```

Fig. 7 Pseudo-code of scatter Sorted BiNomial (SBN) tree algorithm

```
input parameters:
P – number of processes/nodes (one process per node)
a_r – arrival time of process r
d_x – input data segments to be scattered, an x segment is to
be sent to the rank x process
rank – the rank of the process
root – the rank of the root process

variables:
nrank – a new rank of the process after the arrival sorting
result – a result data segment
pt_r – an array with rank identifiers, r = 0, 1, ..., P − 1
received – true if the process already received the data

1.  swap(pt_0, pt_root)
2.  sort(pt_r) according to a_r
3.  shuffle(pt_r) according to bit reverse, e.g. rev(0111) =
1110
4.  k := 2^⌈log_2 P⌉

background:
5.  if rank ≠ root then
6.     nrank := index of rank in pt_r
7.     i := k/2
8.     received := false
9.     while i > 0 do
10.      if received then
11.        send(segments: (d_i, ... d_2i−1), to: pt_nrank+i)
12.      if (¬received) ∧ (nrank & i ≠ 0) then
13.        d := receive()
14.        received := true
15.      i := i/2

foreground:
16. if rank = root then
17.    shuffle(d_r) according to pt_r
18.    shuffle(d_r) according to bit reverse, e.g. rev(0111) =
1110
19.    i := k/2
20.    while i > 0 do
21.      send(segments: (d_i, ... d_2i−1), to: pt_nrank+i)
22.      i := i/2
23. else
24.    wait() for the background actions to be finished
25. return d_0
```

Fig. 8 Pseudo-code of scatter Background Sorted BiNomial (BSBN) tree algorithm

moving receive operations of non-root processes into the background thread (lines 5–15), what, in case of delay in the processes, can accelerate the sending of the data segments—the delayed receiving processes do not block the ones which already started sending the data. The activities of the root stay in the foreground (lines 17–22) and the results are returned after the the background thread (in the case of a leaf) finishes its activities (lines 24–25).

Similarly to scatter linear trees, the SBN and BSBN, for the perfectly balanced, flat PAP, do not improve the communication complexity in comparison to their base algorithm: the binomial tree (see Eq. 7), and the computational complexity can be estimated as $O(P \log P + N)$

(because of the process sorting and data shuffling and swapping). However, in the case of an imbalanced PAP, some early message exchange (in the background during the computation phase) with the processes ordered by PATs can speed up the data flow of the collective operation.

Similarly to the linear case we can analyze run times of the binomial-based algorithms considering a situation when one process, either the first receiver (id: 1) or the root (id: 0), is delayed. In the first case $a_1 > a_0$ and $a_0 = a_2 = \cdots = a_{P-1}$, and the run times can be estimated as in the following equations:

$$r^{BN(id:1)} = a_1 - a_0 + r^{BN} \tag{13}$$

$$r^{SBNs(id:1)} = \max\left(a_1 - a_0 + \alpha + \frac{N}{P}\beta, \ r^{BN}\right) \tag{14}$$

$$r^{BSBNs(id:1)} = \max\left(a_1 - a_0, \ r^{BN}\right) \tag{15}$$

where r^{BN} is defined in Eq. 7. In the latter case, when the root process is delayed: $a_0 > a_1$ and $a_1 = a_2 = \cdots = a_{P-1}$, just like for the linear-based algorithms, regardless of the used algorithm modifications, all other processes need to wait. Thus the run times are equal and can be estimated as in the following equation:

$$r^{BN(id:0)} = r^{SBNs(id:0)} = r^{BSBNs(id:0)}$$
$$= a_0 - a_1 + r^{BN} \tag{16}$$

4.2 Gather algorithms

The gather Sorted Linear Synchronized (SLS) tree algorithm (see the pseudo-code in Fig. 9) is based on linear synchronized tree (see Sect. 2.1), with the extensions related to the order of the performed message exchange, where the data from the faster leaf processes can be received before the data from the slower ones (line 3). The other operations seem to stay the same, i.e. the data vectors are received in two segments (lines 3–12) and the leaf processes wait for receiving the empty, synchronization message before sending the data (lines 16–18).

The gather SLS algorithm can be extended to Background Sorted Linear Synchronized (BSLS) tree (see the pseudo-code in Fig. 10), where the receiving the data in the root process is moved into the background thread (lines 1–9). Thus, in case when the root process is delayed, it still can manage the receiving of the gathered data sent by the leaves (lines 17–19), even in the ongoing communication phase, leaving to the foreground only merging its own data (lines 12–13).

For a perfectly balanced, flat PAP the communication complexity of SLS and BSLS algorithms is the same as for

```
input parameters:
P − number of processes/nodes (one process per node)
a_r − arrival time of process r
rank − the rank of the process
root − the rank of the root process
inp − the data to be sent to the root by the current process

variables:
inp^A, inp^B − the variable to be used for splitting the input
data to two segments
pt_r − an array with rank identifiers, r = 0, 1, . . . , P − 1
d_x^A, d_x^B − the data segments to be gathered by the root
process, x segments are to be received from the rank x
process

1.  split(inp, into: inp^A, inp^B)
2.  if rank = root then
3.    sort(pt_r) according to a_r
4.    for i := 0 to P − 1
5.      if i ≠ root then
6.        d_{pt_i}^A := ireceive() // non-blocking
7.        send(NIL, to: pt_i) // an empty message
8.        d_{pt_i}^B := ireceive() // non-blocking
9.        wait() for receiving segment d_i^A
10.     else
11.       d_{rank}^A := inp^A
12.       d_{rank}^B := inp^B
13.    wait() for receiving all segments d_r^B
14.    return (d_0, . . . d_{P−1})
15.  else
16.    receive() // an empty message
17.    send(segment: inp^A, to: root)
18.    send(segment: inp^B, to: root)
```

Fig. 9 Pseudo-code of gather Sorted Linear Synchronized (SLS) tree algorithm

```
input parameters:
P − number of processes/nodes (one process per node)
a_r − arrival time of process r
rank − the rank of the process
root − the rank of the root process
inp − the data to be sent to the root by the current process

variables:
inp^A, inp^B − the variable to be used for splitting the input
data to two segments
pt_r − an array with rank identifiers, r = 0, 1, . . . , P − 1
d_x^A, d_x^B − the data segments to be gathered by the root
process, x segments are to be received from the rank x
process

background:
1.  if rank = root then
2.    sort(pt_r) according to a_r
3.    for i := 0 to P − 1
4.      if i ≠ root then
5.        d_{pt_i}^A := ireceive() // non-blocking
6.        send(NIL, to: pt_i) // an empty message
7.        d_{pt_i}^B := ireceive() // non-blocking
8.        wait() for receiving segment d_i^A
9.    wait() for receiving all segments d_r^B

foreground:
10. split(inp, into: inp^A, inp^B)
11. if rank = root then
12.   d_{rank}^A := inp^A
13.   d_{rank}^B := inp^B
14.   wait() for the background actions to be finished
15.   return (d_0, . . . d_{P−1})
16. else
17.   receive() // an empty message
18.   send(segment: inp^A, to: root)
19.   send(segment: inp^B, to: root)
```

Fig. 10 Pseudo-code of gather Background Sorted Linear Synchronized (BSLS) tree algorithm

LS tree, see Eqs. 5 and 6. However, when some leaf processes are delayed, the SLS/BSLS can accelerate the whole operation, and for BSLS it is possible even in the case of the delayed root. The additional sorting of the processes by their PATs introduces a computation overhead estimated as $O(P \log P)$.

Below we analyze run times of the proposed LS-based algorithms, considering a situation when one process, either the first receiver (id: 1) or the root (id: 0), is delayed. In the first case $a_1 > a_0$ and $a_0 = a_2 = \cdots = a_{P-1}$, and the run times can be estimated as in the following equations:

$$r^{LS(id:1)} = a_1 - a_0 + r^{LS} \tag{17}$$

$$r^{SLS(id:1)} = r^{BSLS(id:1)}$$
$$= \max\left(a_1 - a_0 + 3\alpha + \frac{N}{P}\beta, \ r^{LS}\right) \tag{18}$$

where r^{LS} is defined in Eq. 5. In the latter case, when the root process is delayed: $a_0 > a_1$ and $a_1 = a_2 = \cdots = a_{P-1}$, for SLS algorithm, the sending processes need to wait for the root, thus the run time is the same as for LS:

$$r^{SLS(id:0)} = r^{LS(id:0)} = a_0 - a_1 + r^{LS} \tag{19}$$

However, BSLS algorithm uses the background thread for preliminary data exchange and the root can collect the data even before the computation phase is finished. Thus the run time can be estimated as follows:

$$r^{BSLS(id:0)} = \max\left(a_0 - a_1, r^{LS}\right) \tag{20}$$

The gather Sorted BiNomial (SBN) tree algorithm (see the pseudo-code in Fig. 11) extends a regular binomial tree by introducing the PAT related order (lines 1–2) of the message exchange, causing the faster processes to send their data at the beginning, without waiting for the slower ones (lines 11–13). After the above procedure, the root process needs to shuffle the received data vector back to its proper order (lines 19–20).

The last proposed gather algorithm: Background Sorted BiNomial (BSBN) tree modifies the SBN, by moving the loop with the receiving operations into the background

```
input parameters:
P − number of processes/nodes (one process per node)
a_r − arrival time of process r
rank − the rank of the process
root − the rank of the root process
inp − the data to be sent to the root by the current process

variables:
nrank − a new rank of the process after the arrival sorting
pt_r − an array with rank identifiers, r = 0, 1, . . . , P − 1
d_x − the data segments to be gathered by the root process,
an x segment is to be received from the rank x process
sent − true if the process already sent the data

 1. swap(pt_0, pt_root)
 2. sort(pt_r) according to a_r descending
 3. nrank := index of rank in pt_r
 4. k := 2^⌈log_2 P⌉

 5. d_0 := inp
 6. i := k/2
 7. sent := false
 8. s := 1

 9. while i > 0 do
10.   if ¬sent then
11.     if nrank&i ≠ 0 then
12.       send(segments: (d_0, . . . d_{s−1}), to pt_{rank−i})
13.       sent := true
14.     else
15.       segments: (d_{s/2}, . . . d_{s−1}) := receive()
16.   i := i/2
17.   s := s × 2
18. if rank = root then
19.   shuffle(d_r) according to bit reverse, e.g. rev(0111) = 1110
20.   shuffle(d_r) according to pt_r
21.   return (d_0, . . . d_{P−1})
```

Fig. 11 Pseudo-code of gather Sorted BiNomial (SBN) tree algorithm

```
input parameters:
P − number of processes/nodes (one process per node)
a_r − arrival time of process r
rank − the rank of the process
root − the rank of the root process
inp − the data to be sent to the root by the current process

variables:
nrank − a new rank of the process after the arrival sorting
pt_r − an array with rank identifiers, r = 0, 1, . . . , P − 1
d_x − the data segments to be gathered by the root process,
an x segment is to be received from the rank x process
sr − indicates where to send a partially gathered vector data

 1. swap(pt_0, pt_root)
 2. sort(pt_r) according to a_r descending
 3. sr := −1
 4. s := 1

background:
 5. nrank := index of rank in pt_r
 6. k := 2^⌈log_2 P⌉
 7. i := k/2

 8. while i > 0 do
 9.   if sr = −1 then
10.     if nrank&i ≠ 0 then
11.       sr := pt_{rank−i}
12.     else
13.       segments: (d_{s/2}, . . . d_{s−1}) := receive()
14.   s := s × 2
15.   i := i/2

foreground:
16. d_0 := inp
17. if rank ≠ root then
18.   send(segments: (d_0, . . . d_{s−1}), to sr)
19. else
20.   shuffle(d_r) according to bit reverse, e.g. rev(0111) = 1110
21.   shuffle(d_r) according to pt_r
22.   return (d_0, . . . d_{P−1})
```

Fig. 12 Pseudo-code of gather Background Sorted BiNomial (BSBN) tree algorithm

thread (lines 8–15), but keeping the sending operations in the foreground (line 18) (Fig. 12). This approach can accelerate the operation in case the non-leaf processes are delayed in their computation phase. Similarly to the SBN, there is performed sorting of the processes (line 1–2) and shuffling of the received data (lines 20–21).

The communication complexity of the SBN and BSBN algorithms, for the perfectly balanced, flat PAP is similar to the regular BNOM and is denoted by Eqs. 7 and 8. The improvements in performance are possible, when some cooperating processes are delayed, and the introduced order of message flow and/or the background activities cause the faster participants to act earlier than the delayed ones. Due to process sorting and data shuffling, the compute complexity can be estimated as $O(P \log P + N)$.

We can analyze run times of the above algorithms considering a situation when one process, either the first receiver (id: 1) or the root (id: 0), is delayed. In the first case $a_1 > a_0$ and $a_0 = a_2 = \cdots = a_{P−1}$, and the run times can be estimated as in the following equations:

$$r^{BN(id:1)} = r^{SBNg(id:1)} = a_1 - a_0 + r^{BN} \tag{21}$$

$$r^{BSBNg(id:1)} = \max\left(a_1 - a_0 + \alpha + \frac{(P-1)N}{2P}\beta, \; r^{BN}\right) \tag{22}$$

where r^{BN} is defined in Eq. 7. Similarly to the LS-based algorithms, in the latter case, when the root process is delayed: $a_0 > a_1$ and $a_1 = a_2 = \cdots = a_{P−1}$, in SBN algorithm, the sending processes need to wait for the root, thus the run time is the same as for BN:

$$r^{BN(id:0)} = r^{SBNg(id:0)} = a_0 - a_1 + r^{BN} \tag{23}$$

However, BSBN algorithm uses the background thread for preliminary data exchange and the root can collect the data even before the computation phase is finished. Thus the run time can be estimated as follows:

$$r^{BSBNg(id:0)} = \max\left(a_0 - a_1, \; r^{BN}\right) \tag{24}$$

5 Experimental evaluation

A benchmark evaluating the proposed algorithms emulates a typical iterative application (e.g. machine learning), where the input data with a given size are exchanged between the cooperating processes, which some of them are delayed according to a given, randomly generated PAP. Each such process uses the usleep() function calls to indicate the progress of the emulated computations to the background thread, including their start, edge point (at 50% of computations) and finish, see Fig. 13 for the benchmark pseudo-code.

The implementation uses C language (v. C99, compiled by GCC v. 7.3.0 with –O$_3$ optimization), with OpenMPI

```
input parameters:
size – number of elements (floats) in scattered/gathered data
N – number of iterations
maxDelay – maximal delay of the process(es)
algorithm – tested algorithm, e.g. SLS, BSBN
P – number of processes
id – process id – MPI rank: 0 . . . P − 1

output:
rtResults – vector of measured average run times
etResults – vector of measured average elapsed times

variables:
halfTime – 50% of the emulated computation time
startTime – start time of measurement
endTime – end time of measurement
myET – elapsed time measured in the current process
sumET – sum of the average elapsed times of all processes
data – vector of data to be scattered/gathered

1.  MPI_Init()
2.  PAT_Init()
3.  PAT_UseAlg(algorithm)
4.  for i := 1 to N
5.    data := generateRandomData(size)
6.    halfTime := 100 ms + random(0 . . . maxDelay)/2
7.  MPI_Barrier()
8.  MPI_Barrier()
9.  PAT_ProcessingStart()
10.   usleep(halfTime)
11.   PAT_edge(50%)
12.   usleep(halfTime)
13.   PAT_ProcessingEnd()
14.   startTime := MPI_Wtime()
15.   makeOperation(algorithm, data)
16.   endTime := MPI_Wtime()
17.   checkCorrectness(data)
18.   MPI_Allreduce(minAT, endTime, MPI_MIN. . .)
19.   MPI_Allreduce(maxAT, endTime, MPI_MAX. . .)
20.   rtResults[i] := maxAT − minAT
21.   myET := endTime − startTime
22.   MPI_Allreduce(sumET, myET, MPI_SUM. . .)
23.   etResults[i] := sumET/P
24. PAT_Finalize()
25. MPI_Finalize()
```

Fig. 13 Pseudo-code of the performance benchmark

[10] (v. 3.0.0) for processes/nodes message exchange, POSIX Threads [4] (v. 2.12) for intranode communication and synchronization, and GLibc (v. 2.0) for dynamic data structures' management. The similar approach was used in [23].

5.1 Test environment and configuration

The benchmark was executed using a typical HPC cluster: Tryton, located in Centre of Informatics – Tricity Academic Supercomputer and networK (CI TASK) at Gdansk University of Technology, Poland. The supercomputer consists of 40 racks with 1600 nodes intraconnected by FDR 56 Gbps InfiniBand [26] and 1 Gbps Ethernet networks, and has in total 1.48 PFLOPS of theoretical compute power. The typical node contains 2 processors (Intel Xeon Processor E5 v3, 2.3 GHz, Haswell architecture), with 12 physical cores (24 cores per node) and 128 GB RAM [16].

The tests were performed in a separated rack containing 48 typical nodes connected by 1 Gbps Ethernet switch (HP J9728A 2920-48G). The benchmark was executed for both scatter (LIN, BNOM, SLIN, SBN, BSLN, BSBN) and gather (LS, BNOM, SLS, SBN, BSLS, BSBN) operations, including the proposed algorithms and, for comparison purposes, the typical ones. The range of data size covered: 128 K, 256 K, 512 K, 1 M, 2 M of floats (4 bytes long). The above values do not exceed the cache size of the used processors, thus we avoided the additional noise caused by the unpredictable intranode data transfers, a similar approach was taken for the internode communication, where we focused on sizes covering the rendezvous send-receive protocol.

The PAPs were generated randomly, with uniform distribution, and the following maximum delays (PATs) were used: 0, 1, 5, 10, 50, 100, 500 ms. The above values were set up experimentally, we performed the tests with increasing delays, until the changes of the absolute measured time values stabilized on the same level, i.e. an introduction of a lager delay gave the same improvement (in ms), in comparison to the base algorithm, e.g. LIN, as the previous one.

The benchmark performed 128–256 iterations, depending on the maximum PAT (more for lower delays). Eventually the tests were executed for different sizes of process/nodes set: 4, 6, 8, 10, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48.

5.2 The results

Fig. 14 presents the results of the benchmark execution for different scatter algorithms regarding the changing maximum arrival time-delay of the processes. We can observe the larger the delay in arrival times (PATs) the better the
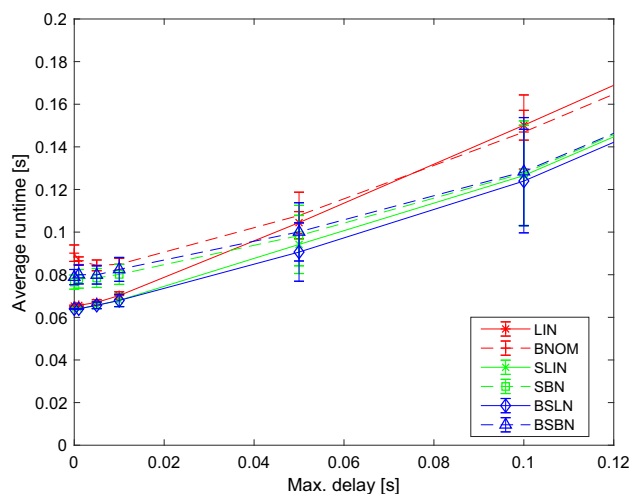
Fig. 14 Benchmark results of the scatter algorithms' run times for increasing maximum delay. The experiments were performed on 48 nodes connected by 1 Gbps Ethernet network, the processes were delayed randomly (uniform distribution), and the total data size: 2 M of floats (8 MB). The error bars are set to $\pm\sigma$ (68% of the measurements for the normal distribution)

PAP-aware algorithms' behavior: the measured run times are shorter. For the assumed configuration (48 nodes with 2 M of floats data size) BSLN achieves the best results, stabilizing the gained advantage with more imbalanced PAPs (with the maximum delay over 100 ms).

Let's analyze scatter results for 0 ms and 50 ms maximum delay as an example, the distribution is uniform, thus in the latter case the mean delay is 25 ms. The average run times are as follow: for 0 ms LIN 65 ms, SLIN 64 ms, BSLN 64 ms and for the 50 ms LIN 104 ms, SLIN 94 ms, BSLN 91 ms. So, the BSLN run time for 50 ms delay is greater than maximum and mean delays, as well as base result for 0 ms delay. Thus it alleviates imbalances for 14 ms in comparison to LIN (default MPI). The interesting observation is that in some limited range LIN algorithm itself alleviates the imbalances, its run time is lower than a sum of the base result (for 0 ms delay) and maximum delay (50 ms in this case). The above phenomenon is true for other scatter and gather results.

The detailed results comparison between the PAP-aware scatter algorithm: background sorted linear (BSLN) tree and the regular linear (LIN) tree is presented in Table 3. Apart from the mentioned delay, we can also notice that the BSLN works better with larger data size, where the gained improvement can be estimated up to 21% (faster by factor 1.27, for maximum delay 50 ms and data size 1 M of floats). As we expected in the theoretical analysis (see Sect. 4.1), the aforementioned algorithm does not provide significant performance increase in the case of the balanced PAPs.

The performance results of the gather algorithms for 48 processes/nodes and data vector size 2 M of floats are presented in Fig. 15. The chart shows the advantage (shorter run times) of the PAP-aware algorithms (SLS, SBN, BSLS and BSBN) in the case of larger delays—arrival times of the processes (PATs). For the more balanced PAPs the typical approach (LS and BNOM) shows better behavior, which is compliant with the theoretical analysis presented in Sect. 4.2. We can notice that for the provided conditions, BSLS presents the best performance, showing the advantage over other algorithms.

An interesting phenomenon can be observed for SLS and BSLS algorithms: the average run time is decreasing with the increasing delays, in interval 0–50 ms. We assume the reason is related to the diminishing contingency: linear sync based algorithms (LS, SLS and BSLS) start with sending messages (the first segments) from the leaves to the root node, what can cause a collision, which has to be resolved by the network switch, leading to some additional latency in the data transmission. However, when the processes are sorted according to their arrival times (PATs) and spread due to the introduced random delays, the above collision does not occur or, at least, is less significant, causing the observable performance improvements.

Thus, Table 4 presents the detailed comparison between the BSLS and the regular LS run times. We can notice, that even for shorter data size the algorithm performs quite well, and the results seem to be better for the higher maximum delays, up to 60% time saving (faster by factor 2.52, for maximum delay 50 ms and data size 2 M of floats).

Figure 16 presents the algorithms' behavior in the case of increasing scattered data size with the constant maximum delay and node number. Analyzing the chart, we can observe that the longer messages, the larger benefit of using the PAP-aware algorithms, however the absolute gains decrease with the size. Thus, for small data size, where the network latency is more important, the proposed algorithms are not so efficient in comparison with their non-PAP-aware counterparts, but with the longer messages where the bandwidth is more important, the algorithms provide greater performance improvements. Eventually, in the case of the largest data sizes, due to the constant maximum delay, the benefits of the algorithms usage stabilize.

Finally we can asses the scalability of the PAP-aware algorithms, Fig. 17 shows the measurements of the run times of the scatter algorithms regarding the increasing number of processes/nodes (up to 48). We can observe that although the times increase for larger configurations, the growth is moderate and the PAP-aware algorithms show their advantage for the whole range of the performed tests.

Table 3 Comparison of BSLN and LIN scatter algorithms for 1 Gbps Ethernet and 48 processes/nodes

Max delay → Size↓	0 Mean run time	1	5	10	50	100	500
128	0.0	− 0.1	0.0	− 0.1	− 0.1	− 0.1	0.0
	1.02	1.20	1.01	1.01	1.00	1.00	1.00
256	0.2	− 0.1	− 0.1	− 0.1	− 0.1	− 0.1	− 0.1
	0.85	1.12	1.01	1.01	1.00	1.00	1.00
512	− 0.1	0.1	− 0.1	− 0.1	− 0.1	− 0.1	− 0.1
	1.06	0.94	1.02	1.01	1.00	1.00	1.00
1024	0.2	− 0.8	− 0.6	− 2.9	− 15.6	− 15.3	− 16.9
	0.99	1.03	1.02	1.08	1.27	1.15	1.04
2048	− 1.3	− 1.6	− 1.4	− 2.3	− 13.8	− 26.2	− 37.0
	1.02	1.03	1.02	1.03	1.15	1.21	1.08

Maximum delay is measured in ms and size in K of floats (4×KB). Each entry consists of two values: a difference of the run times in ms ($r^{BSLN} - r^{LIN}$) and acceleration: a quotient of the run times ($\frac{r^{LIN}}{r^{BSLN}}$). The bold values indicate better performance in comparison with the BSLN algorithm
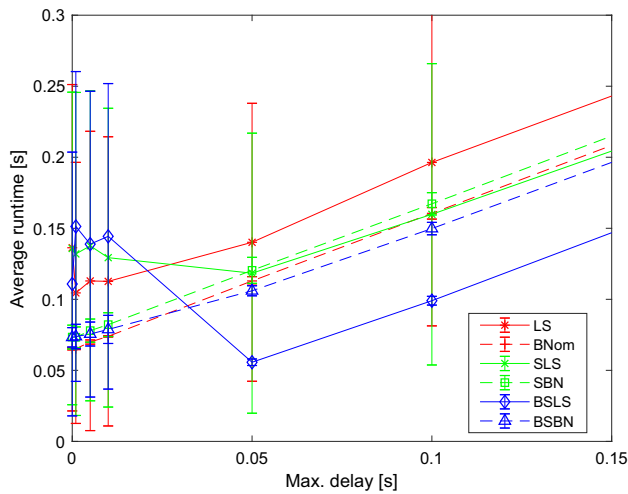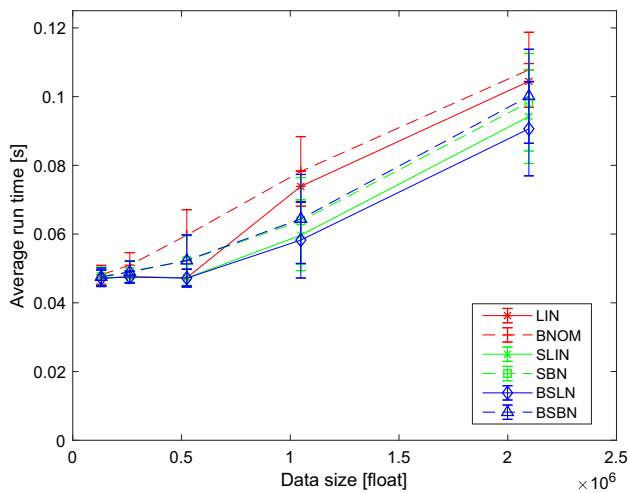

Fig. 15 Benchmark results of the gather algorithms' run times for increasing maximum delay. The experiments were performed on 48 nodes connected by 1 Gbps Ethernet network, the processes were delayed randomly (uniform distribution), and the total data size: 2 M of floats (8 MB). The error bars are set to $\pm\sigma$ (68% of the measurements for the normal distribution)

We can conclude that the experimental results show a clear improvement of the scatter/gather operations' performance while executed in imbalanced PAP environment, in comparison to the default OpenMPI (LIN/LS) and MPICH (BNOM) algorithms. The analogous results of the same experiments, but presenting the average elapsed times instead of run times, are presented in Appendix, showing the similar advantages of the proposed PAP-aware algorithms.

6 Practical use case: parallel FFT

As the use case of typical usage of the HPC cluster we propose Fast Fourier Transform (FFT) parallel implementation, with hierarchical partitioning of the processed data under the master–slave programming paradigm. We use a typical Radix-2 algorithm with Decimation-In-Frequency approach enabling easy distribution of preprocessed data to the slave processes deployed in separated computation nodes [3]. At the higher, internode level the communication is performed by MPI [12] calls, using both point-to-point (for data distribution to the slaves) and collective (for data gathering to the master) operations. At the intranode level the computations are performed using OpenMP [6] where the shared memory is used for data exchange and thread synchronization.

The implementation uses up to 24 threads per node for the computation purposes, managed by the OpenMP framework [6]. The underlying hardware (2×Intel Xeon CPUs per node) provides matching 24 physical cores with the Hyper Threading mechanism switched off—a typical configuration used in HPC computations. For the PAP-aware algorithms, the background thread is implemented using a different approach of parallelization, namely POSIX Thread library [4]. Thus, in this case, the background thread does not have a dedicated core, and causes processor oversubscription, this overhead is perceived as a computational cost of the proposed solution, however when we compare compute times measured in the performed experiments we can observe that it is negligible: the differences do not seem to depend on the algorithm used and they are smaller than 0.5%.

The input data were randomly generated by the master process and distributed to 7 slaves (the master also performs computations), where the processing was performed

Table 4 Comparison of BSLS and LS gather algorithms for 1 Gbps Ethernet and 48 processes/nodes

Max delay → Size↓	0 Mean run time	1	5	10	50	100	500
128	− 0.7	− 0.8	− 4.1	− 4.9	− 4.1	− 3.3	− 2.6
	1.13	1.13	1.68	1.52	1.09	1.03	1.01
256	− 1.1	− 0.9	− 4.3	− 5.8	− 4.7	− 4.1	− 3.1
	1.17	1.14	1.63	1.61	1.10	1.04	1.01
512	− 0.7	− 0.3	− 3.7	− 4.9	− 4.6	− 3.8	− 3.0
	1.12	1.05	1.57	1.51	1.10	1.04	1.01
1024	29.1	− 8.1	16.6	22.2	− 69.6	− 72.2	− 68.1
	0.79	1.07	0.86	0.83	2.40	1.74	1.14
2048	− 25.5	46.8	25.9	31.7	− 84.4	− 97.3	− 88.5
	1.23	0.69	0.81	0.78	2.52	1.98	1.18

Maximum delay is measured in ms and size in K of floats (4×KB). Each entry consists of two values: a difference of the run times in ms ($r^{BSLS} - r^{LS}$) and acceleration: a quotient of the run times ($\frac{r^{LS}}{r^{BSLS}}$). The bold values indicate better performance in comparison with the BSLS algorithm

Fig. 16 Benchmark results of the scatter algorithms' run times for increasing data size. The experiments were performed on 48 nodes connected by 1 Gbps Ethernet network, the processes were delayed randomly (uniform distribution), and the maximum delay was set to 50 ms. The error bars are set to $\pm\sigma$ (68% of the measurements for the normal distribution)

Fig. 17 Benchmark scalability results of the scatter algorithms. The experiments were performed on up to 48 nodes connected by 1 Gbps Ethernet network, the processes were delayed randomly (uniform distribution), the maximum delay: 50 ms, and the total data size: 2 M of floats (8 MB). The error bars are set to $\pm\sigma$ (68% of the measurements for the normal distribution)

in iterative manner, and every iteration data vector size is 256 K of floats (1 MB), and 1000 iterations were executed for each test. The experiments were deployed with a similar configuration as the one used by the benchmark (see Sect. 5.1), except that the regular Tryton supercomputer queue system (SLURM [29]) was utilized, just like for any other compute jobs started by regular users, in contrast to the separated rack designated for the benchmark. Each experiment, consisting of the 1000 iterations for each tested algorithm, was repeated 100 times on 8 compute nodes with 1 Gbps Ethernet connection.

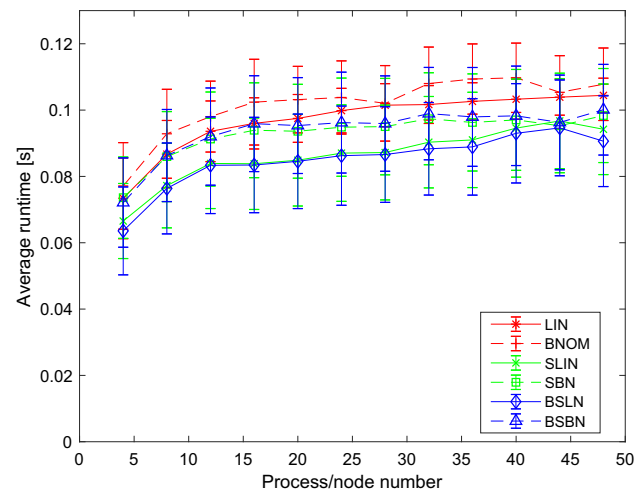Table 5 presents the results of the experiments. The PAP-aware algorithms show their advantage over the regular approach, and for this configuration, the best performance is obtained by SLS and BSLS, with 3.3% acceleration of the total application execution time (2.126 s in absolute value) over default LS, which was also used by OpenMPI [10] implementation, providing similar results. This result was achieved by optimizing gather operation only, in a parallel program, where, on average, the computations cover over 60% of the processing time (43.517 s in absolute value).

The other measurements also confirm even more the superiority of the SLS/BSLS algorithms, with the lowest run times (10 ms, 16.7% shorter than LS) and average elapsed times (4 ms, 50% shorter than LS). Finally, the

Table 5 Parallel FFT execution results for 1 Gbps Ethernet and 8 processes/nodes

Algorithm	Total time	Run time	Elapsed time
LS	64.854	0.012	0.004
BNOM	69.632	0.017	0.005
SLS	62.728	0.010	0.002
SBN	69.845	0.017	0.005
BSLS	62.813	0.010	0.002
BSBN	67.402	0.015	0.003
MPI	64.815	0.012	0.004

The row labeled MPI indicates default (LS) OpenMPI gather implementation. The measured times are presented in seconds

binomial tree solutions, both PAP-aware (BSBN/SBN) and regular (BNOM), showed the worst results, what could be expected for the given data vector sizes (binomial trees are rather designed for shorter messages).

The above results show a clear performance improvement for a real HPC program, which in turn is frequently used for many scientific applications, e.g. audio analysis, radio telescope signal correlation. We would like to emphasize that it is just one example of possible usage of this approach, which can be introduced for many other iterative, parallel programs. The achieved acceleration (3.3% of total application execution time and 16.7% of run time related to the communication operations), moderate at the first look, enables significant savings in the used infrastructure, what is important for current, large investments in the HPC industry, where every percent of budget decrease means a huge cost reduction.

7 Conclusions and future works

We presented a collection of PAP-aware scatter/gather algorithms based on typical, linear and binomial tree approaches. The performed experiments, based on the developed benchmark as well as a real case application, showed a significant improvement of the computation performance, for a typical HPC environment. Furthermore, the results proved that the solution is well scalable and can be used for a wide range of parallel applications.

We expect that the ubiquity of imbalanced PAP occurrences in HPC systems [9] will drive more focus for the research in this area and the following works are going to be performed in the future:

- introduction of new collective PAP-aware algorithms, for other collective operations: e.g. all-to-all, all-gather,
- extension of the algorithms to be used for hierarchical architecture, e.g. when more than one process work on the same node, or grid of clusters is used,
- the evaluation of the ultra-scale HPC environments for imbalanced PAPs using typical simulation tools, e.g. [5, 24],
- improving the existing PAP-aware algorithms by introduction of hardware related solutions (e.g. specific Infiniband [26] features like multicast),
- introduction of the proposed algorithms into other computing environment (besides HPC), like cloud or specific processing platforms, e.g. for machine learning,
- usage of the PAP evaluation methods for other purposes, like deadlock and time dependent errors detection in parallel programs [15],
- a dedicated framework automating PAP-aware algorithm injection into existing parallel applications.

Appendix: Benchmark results for average elapsed time

In this appendix, we provide the benchmark results presented in terms of average elapsed times for scatter (Fig. 18 and Fig. 19) and gather (Fig. 20) algorithms. We can notice, that the measurement values for the tested algorithms behave similarly to the run times presented in Sect. 5.2, showing the advantage of PAP-aware algorithms.
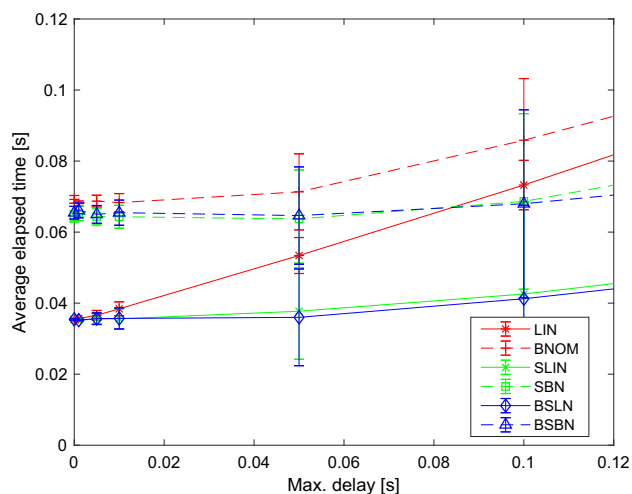
Fig. 18 Benchmark results of the scatter algorithms' average elapsed times for increasing maximum delay. The experiments were performed on 48 nodes connected by 1 Gbps Ethernet network, the processes were delayed randomly (uniform distribution), and the total data size: 2 M of floats (8 MB). The error bars are set to $\pm\sigma$ (68% of the measurements for the normal distribution)
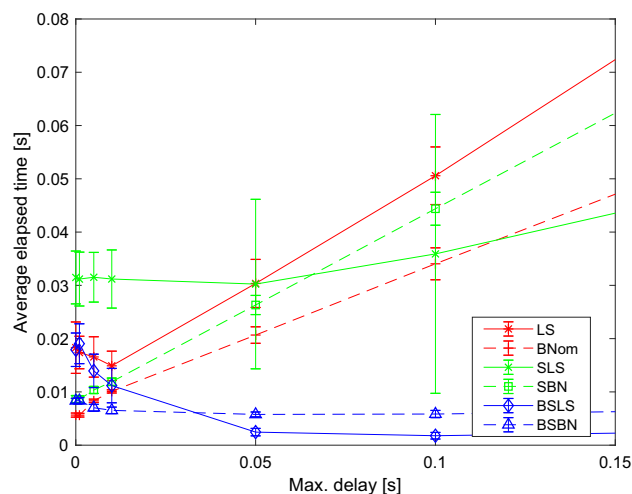

Fig. 20 Benchmark results of the gather algorithms' average elapsed times for increasing maximum delay. The experiments were performed on 48 nodes connected by 1 Gbps Ethernet network, the processes were delayed randomly (uniform distribution), and the total data size: 2 M of floats (8 MB). The error bars are set to $\pm\sigma$ (68% of the measurements for the normal distribution)
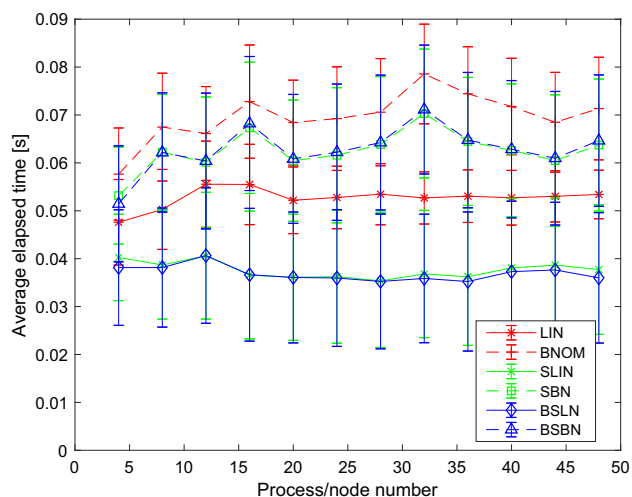

Fig. 19 Benchmark scalability results of the scatter algorithms in terms of average elapsed time. The experiments were performed on up to 48 nodes connected by 1 Gbps Ethernet network, the processes were delayed randomly (uniform distribution), the maximum delay: 50 ms, and the total data size: 2 M of floats (8 MB). The error bars are set to $\pm\sigma$ (68% of the measurements for the normal distribution)

References

1. Arap, O., Swany, M., Brown, G., Himebaugh, B.: Adaptive recursive doubling algorithm for collective communication. In: 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, pp. 121–128. IEEE (2015)
2. Bailey, D.H.: NAS parallel benchmarks. In: Padua, D. (ed.) Encyclopedia of Parallel Computing, pp. 1254–1259. Springer, Boston (2011)
3. Balducci, M., Choudary, A., Hamaker, J.: Comparative analysis of FFT algorithms in sequential and parallel form. In: Mississippi State University Conference on Digital Signal Processing, pp. 5–16 (1996)
4. Butenhof, D.R.: Programming with POSIX Threads. Addison-Wesley Professional, Boston (1997)
5. Czarnul, P., Kuchta, J., Matuszek, M., Proficz, J., Rościszewski, P., Wójcik, M., Szymański, J.: MERPSYS: an environment for simulation of parallel application execution on large scale HPC systems. Simul. Model. Pract. Theory **77**, 124–140 (2017)
6. Dagum, L., Menon, R.: OpenMP: an industry standard API for shared-memory programming. IEEE Comput. Sci. Eng. **5**(1), 46–55 (1998)
7. Dichev, K., Rychkov, V., Lastovetsky, A.: Two algorithms of irregular scatter/gather operations for heterogeneous platforms. In: Keller, R., Gabriel, E., Resch, M., Dongarra, J. (eds.) Recent Advances in the Message Passing Interface, pp. 289–293. Springer, Berlin (2010)
8. Faraj, A., Yuan, X., Lowenthal, D.: STAR-MPI: self tuned adaptive routines for MPI collective operations. In: Proceedings of the 20th Annual International Conference on Supercomputing, pp. 199–208 (2006)
9. Faraj, A., Patarasuk, P., Yuan, X.: A study of process arrival patterns for MPI collective operations. Int. J. Parallel Program. **36**(6), 543–570 (2008)
10. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: goals, concept, and design of a next generation MPI implementation. In: Kranzlmüller, D., Kacsuk, P., Dongarra, J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface, pp. 97–104. Springer, Berlin (2004)
11. Gropp, W., Lusk, E.: User's guide for MPICH, a portable implementation of MPI. Technical Report ANL-96/6, Argonne National Laboratory (1994)
12. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message-Passing Interface. The MIT Press, Cambridge (1996)
13. Hockney, R.W.: The communication challenge for MPP: Intel Paragon and Meiko CS-2. Parallel Comput. **20**(3), 389–398 (1994)

14. Kandalla, K., Subramoni, H., Vishnu, A., Panda, D.K.: Designing topology-aware collective communication algorithms for large scale InfiniBand clusters: Case studies with Scatter and Gather. In: 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and PhD Forum (IPDPSW), pp. 1–8. IEEE (2010)

15. Krawczyk, H., Krysztop, B., Proficz, J.: Suitability of the time controlled environment for race detection in distributed applications. Future Gener. Comput. Syst. **16**(6), 625–635 (2000)

16. Krawczyk, H., Nykiel, M., Proficz, J.: Tryton supercomputer capabilities for analysis of massive data streams. Polish Maritime Res. **22**(3), 99–104 (2015)

17. LAMMPS benchmarks. URL: https://lammps.sandia.gov/bench.html. Accessed 09 Dec 2018

18. Lockwood, J.W., McKeown, N., Watson, G., Gibb, G., Hartke, P., Naous, J., Raghuraman, R., Luo, J.: NetFPGA—an open platform for gigabit-rate network switching and routing. In: 2007 IEEE International Conference on Microelectronic Systems Education (MSE'07), pp. 160–161. IEEE (2007)

19. Marendic, P., Lemeire, J., Vucinic, D., Schelkens, P.: A novel MPI reduction algorithm resilient to imbalances in process arrival times. J. Supercomput. **72**, 1973–2013 (2016)

20. Marendić, P., Lemeire, J., Haber, T., Vučinić, D., Schelkens, P.: An investigation into the performance of reduction algorithms under load imbalance. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). LNCS, vol. 7484, pp. 439–450. Springer, Berlin (2012)

21. Patarasuk, P., Yuan, X.: Efficient MPI Bcast across different process arrival patterns. In: 2008 IEEE International Symposium on Parallel and Distributed Processing, pp. 1–11. IEEE (2008)

22. Petrini, F., Kerbyson, D.J., Pakin, S.: The case of the missing supercomputer performance. Proceedings of the 2003 ACM/IEEE Conference on Supercomputing—SC'03, vol. 836, p. 55. ACM Press, New York (2003)

23. Proficz, J.: Improving all-reduce collective operations for imbalanced process arrival patterns. J. Supercomput. **74**(7), 3071–3092 (2018)

24. Proficz, J., Czarnul, P.: Performance and power-aware modeling of MPI applications for cluster computing. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 9574, pp. 199–209. Springer, Berlin (2016)

25. Qian, Y., Afsahi, A.: Process arrival pattern aware alltoall and allgather on infiniband clusters. Int. J. Parallel Program. **39**(4), 473–493 (2011)

26. Shanley, T.: Infiniband Network Architecture. Addison-Wesley Professional, Boston (2003)

27. Träff, J.L.: Practical, distributed, low overhead algorithms for irregular gather and scatter collectives. Parallel Comput. **75**, 100–117 (2018)

28. Traff, JL: Hierarchical gather/scatter algorithms with graceful degradation. In: 18th International Proceedings on Parallel and Distributed Processing Symposium, 2004, pp. 80–89. IEEE (2004)

29. Yoo, A.B., Jette, M.A., Grondona, M.: SLURM: simple Linux utility for resource management. In: Feitelson, D., Rudolph, L., Schwiegelshohn, U. (eds.) Job Scheduling Strategies for Parallel Processing, pp. 44–60. Springer, Berlin (2003)

Jerzy Proficz Ph.D. is the Deputy Director of the Centre of Informatics - Tricity Academic Supercomputer & networK (CI TASK) at Gdansk University of Technology, Poland. He earned his Ph.D. (2012) in High Performance Computing (HPC) in the subject of supercomputer resource provisioning and management for on-line data processing. Author and coauthor of over 40 scientific journal and conference publications related to the HPC technologies, especially to parallel and distributed computing. He has 10 years' experience in the business environment, as a team leader and manager, as well as a software developer and designer, working for a variety of Polish and international companies. He participated in several Polish and international/European R&D projects. Taught courses on computer science, including: system software, parallel and distributed processing, Java programming, etc.