The Journal of Supercomputing
https://doi.org/10.1007/s11227-020-03499-1

Improving Clairvoyant: reduction algorithm resilient to imbalanced process arrival patterns

Jerzy Proficz[1] · Krzysztof M. Ocetkiewicz[1]

Accepted: 28 October 2020

Abstract
The Clairvoyant algorithm proposed in "A novel MPI reduction algorithm resilient to imbalances in process arrival times" was analyzed, commented and improved. The comments concern handling certain edge cases in the original pseudocode and description, i.e., adding another state of a process, improved cache friendliness more precise complexity estimations and some other issues improving the robustness of the algorithm implementation. The proposed improvements include skipping of idle loop rounds, simplifying generation of the ready set and management of the state array and an about 90-fold reduction in memory usage. Finally an extension enabling process arrival times (PATs) prediction was added: an additional background thread used to exchange the data with the PAT estimations. The performed tests, with a dedicated mini-benchmark executed in an HPC environment, showed correctness and improved performance of the solution, with comparison to the original or other state-of-the-art algorithms.

Keywords Reduce · Clairvoyant · Process arrival pattern · MPI

1 Introduction

Current trends in the high performance computing (HPC), stimulated by the rapid growth of the Artificial Intelligence (AI), Internet of Things (IoT), or Big Data analysis methods and tools, show massive development of compute cluster architectures, where most supercomputers consist of independent nodes connected by a fast interconnecting network, usually InfiniBand or Ethernet. In such environment, a natural approach

✉ Jerzy Proficz
j.proficz@task.gda.pl

Krzysztof M. Ocetkiewicz
k.ocetkiewicz@task.gda.pl

[1] Gdansk University of Technology Centre of Informatics-Tricity Academic Supercomputer and NetworK (CI TASK), 11/12 Gabriela Narutowicza Street, 80-233 Gdańsk, Poland

Published online: 20 November 2020

to provide data exchange and synchronization mechanisms is the message passing paradigm, with the usually used Message Passing Interface (MPI) standard [24], and its support for both point-to-point and collective operations.

Thus, we can observe a rapid progress in parallelization of currently existing compute methods and applications, especially ones requiring the large compute resources. We perceive that the improvements of HPC communication algorithms and protocols, including our works, will enable faster optimization of the above crucial areas (IoT, AI, Big Data), especially in such topics like parallelization of hybrid parallel FTDT methods [22], intelligent home systems supported by neural networks [26], Big Data related programming models and systems [1], and voice evaluation mechanisms involving such complex mechanisms like bio-inspired algorithms or spiking neural network [14].

In the aforementioned cluster environment, every compute node, in fact a separate computer, has its own RAM memory, processor(s), I/O devices and possibly accelerators (especially GP GPU cards), data storage etc. Such a device is managed by an independent operating system, having a specific process/thread scheduler, power management mechanisms and a number of other system functions usually performed periodically or randomly. Even for the homogeneous cluster, all these features cause that the communication and computations are not performed in the same pace for all of the nodes.

Such desynchronization is especially visible for collective communication, where each participating process can start the operation in its own arrival time (PAT), which in many cases can create high time differences, called imbalanced process arrival patterns (PAP). The comprehensive analysis of a collection of HPC benchmarks showed that the mentioned imbalances can have a great impact on the application performance [6]. Moreover, the recently observed parallelization of hardware and system components, e.g. introduction of more compute cores sharing the same memory channels, which performance behavior in many cases is non-deterministic, can multiply such effects.

The MPI standard defines the operations, but their exact implementation depends solely on the providers. Usually each collective operation has several possible algorithms realizing its functionality, within MPI standard boundaries. They can be tuned for different constraints such as data size, number of processes, connection speed or network architecture. In our case we optimize collective operations by providing some algorithms (partially) resilient to imbalanced PAP environment, e.g. [15,16].

A novel PAP-aware algorithm was presented in [9] by Marendic et al. It is dedicated to the MPI reduce operation, which makes the data vectors initially placed in the cooperating processes, to be reduced into one value by a defined math operation, e.g. sum, and placed in a root process. The aforementioned algorithm seemed to be very promising, the provided experimental results showed a positive impact on the overall performance; however, during an independent implementation a number of issues occurred.

In this article we provide a collection of comments and improvements to the above algorithm, which increase its efficiency and usability, while keeping its ability of the imbalanced PAPs mitigation. Our contribution is as follows:

- Precise specification of algorithm's behavior in edge cases not handled in the original algorithm pseudocode and description,
- Introduction of improvements to the algorithm, increasing its performance,
- Extension to the algorithm covering a PAP prediction method, ported from our other works: [15,16], enhancing its usability,
- Practical verification of the proposed improvements and extension, based on the experiments performed in a typical HPC environment.

Thus, the above modifications were introduced in two phases: in the first one, we implemented the updated algorithm with the proposed improvements and performed the practical verification on one CPU, showing the better performance in comparison to the base algorithm, see the results in Sect. 5.7. In the second phase, we introduced the extension, increasing the usability, such the algorithm can be used without *a priori* PAP related knowledge, and showed, trough the series of the experiments executed in an HPC cluster environment (up to 48 compute nodes), that this approach is superior to the currently sate-of-the-art algorithms (binomial tree, Rabenseifner, Radix-K and parallel ring), see the results in Sect. 6.

The paper has the following structure: the next section describes the original algorithm including its pseudocode, Sect. 3 presents the main works related to the imbalanced PAP subject, Sect. 4 presents analysis of the Clairvoyant algorithm and necessary modifications we had to apply to perform code implementation. Then, in Sect. 5, some improvements leading to increased performance and the experimental results showing their effectiveness are presented. In Sect. 6, an extension, realized by additional background thread, enabling PAP estimation, to feed the algorithm with the PAT data, along with a practical verification of the proposed method are described. Finally, some conclusions and future works are presented.

2 Original algorithm

This paper concerns an article [9], provided by Marendic et al. Its main subject is an MPI reduction algorithm whose main feature is the ability to mitigate an impact caused by imbalanced PAPs. The algorithm is dedicated for segmentable data and based on greedy scheduling of the segment transfers between the cooperating processes as soon as they are available for the reduce operation; the pseudocode is presented in Listing 1.

The algorithm can be summarized as follows. On input P processes $p_1, \ldots, p_P$ are given. The time process p_i arrives is given by a_i. The data to be gathered is divided into N segments. The algorithm produces two queues for every process: input and output. Both queues hold a list of pairs. For process p, every pair describes a communication event. A (q, s) pair in the input queue denotes segment s being sent from process q to process p and combined by p. Analogously, (q, s) pair in the output queue represents process p sending segment s to be combined by process q. All nodes partaking a reduce operation execute the algorithm in parallel and perform communication as directed by the contents of their communication queues.

Every (process, segment) pair has a state, kept in M array. The states belong to one of the following types: A - the process has not yet sent its part of the segment, E - the process has sent its part of the segment, P - the process holds a partially combined segment. When process p_a sends its part of segment s to process p_b, state $M(p_a, s)$ is updated to E while state $M(p_b, s)$ is set to P. The algorithm works in rounds of length d time units. Processes are kept in a queue Q, ordered by their arrival times. In every round, a sequence G of processes that arrive no later than d units of time after the first process in the Q is prepared. These processes participate in the round. Then, a sink process is designated. When algorithm's $root$ process is in G, it is selected as the sink. Otherwise, the first process in G becomes the sink. The sink is moved to the front of G. This ensures that, in the end, all segments will be gathered in $root$ process. After the sink is selected, for every process i in G the algorithm tries to find a peer process z and segment j such that z can transmit j to i. Process z is then marked ($S(z) = \top$) as it can no longer send segments in this round. After all processes in G have been considered, these processes that hold unsent segments, i.e. these that have a segment in state different than E, are returned to the queue Q with arrival time incremented by d.

The virtual topology depends on the arrival patterns and segment number. Reduction of every segment will have its own connection topology. In the best case a binomial tree will be constructed for a segment. However, if there is at least two segments, the root node will transmit some of it's segments only to receive it later: the reduction will follow a tree but in one of steps, the ad-hoc root of the segment's reduction will transmit the result to the root node. Therefore, it can be said that the topology for a single segment is a graph with circuit rank not greater than one and the whole virtual topology will be a sum of a number of such graphs.

The algorithm was implemented in C++ and the series of experiments in emulated, imbalanced PAP environment showed its advantage over state-of-the-art algorithms i.e. binomial tree, parallel ring, butterfly and Radix-K. The algorithm assumes *a prori* knowledge about the arrival times of all the cooperating processes, that is why it is called Clairvoyant (CLV). To allow a real-world implementation the authors proposed usage of statistical techniques such as Simple Moving Average (SMA), assuming the repetition of the PATs over the cooperating processes.

3 Related works

Apart of the article: [9] being the subject of this paper, there is a number of other related works. The first comprehensive study over imbalanced PAPs was performed by Faraj et al. in [6]. They presented a definition according to which PAP is a tuple $(a_0, a_1, \ldots, a_{P-1})$ where a_i is the time (PAT) when an i-th process arrive to the operation and P is a number of all cooperating processes. Similarly, there was defined a process exit pattern (PET) as a tuple $(f_0, f_1, \ldots, f_{P-1})$ where f_i is the time when an i-th process finishes the operation. An imbalanced PAP occurrence was defined as if its imbalance factor: a ratio between the largest difference between the arrival times of the processes and time of the simple, point-to-point message delivery between each other, is larger than 1.

There are several PAP-aware algorithms concerning other collective operations. In [12], Patarasuk et. al. presented two new broadcast algorithms: one designed for block-

Listing 1 Original Clairvoyant algorithm pseudocode. Source: [9]

```
Let $M$ be a state matrix of size $P \cdot N$. Set $M(*, *) = $ A, i.e.. mark all segments as available.
Set $S$ : bool be an array of size $P$
For $\forall i \in \{0 \dots P-1\}$ insert process rank $i$ into a priority queue $Q$, where priority is given to process
  rank $i$ with smaller arrival time $a_i$
while size($Q$) > 1 do
    Pop processes $Q_0 \dots Q_k \in Q$ to form a sorted vector $G$, so that $\forall i \in G, a_i \leq a_{Q_0} + d$
    for $\forall i \in G$ do
        let $S(i) = \bot$                                   ▷ no ready process in $G$ has yet sent a segment
    end for
    if $r \in G$ then
        insert $r$ at first position in $G$
    end if
    for $\forall i \in G$ do
        let $j = 0$, let $z = \phi$
        if $r \in G$ then
            let $r^* = r$                                   ▷ sink is the root
        else
            let $r^* = Q_0$                                 ▷ sink is the earliest to arrive process
        end if
        while $z \in \phi$ do
            if $i \neq r^*$ then
                starting from $j$, find first segment $x$, such that $M(i, x) \in \{A, P\}$. Let $j = x$
            else
                starting from $j$, find first segment $x$, such that $M(i, x) \in \{A, P, E\}$. Let $j = x$
            end if
            perform linear search through group $G$ to find the first process $z \neq i$ such that $M(z, j) \in$
              $\{A, P\} \wedge S(z) = \bot$
            if $z \in \phi$ then
                let $j = j + 1$
            end if
        end while
        enqueue to $I(i)$ the pair $(z, j)$
        enqueue to $O(z)$ the pair $(i, j)$ and let $S(z) = \top$
        $M(z, j) = E$
        $M(i, j) = P$
    end for
    for $\forall i \in G$ do
        if $\exists j \in \{0 \dots N-1\} : M(i, j) \neq E$ then
            $a_i = a_i + d$
            push process $i$ into $Q$
        end if
    end for
end while
```

ing (*arrival_b*) and the other for nonblocking (*arrival_nb*) message-passing model. In [10], Marendic et al. proposed another reduce algorithm preserving atomicity of the data and being based on the binomial tree. In [18], Qian et al. described three all-gather and all-to-all related algorithms, which based on the InfiniBand RDMA [20] features used to check the receiving process arrival time estimation. In [15–17], we provided a collection of PAP-aware algorithms for all-reduce, scatter and gather collectives, based on binomial, flat trees and parallel ring approaches.

Every aforementioned solution requires PAT related information to perform communication activities mitigating the PAP imbalance. In [6], Faraj et al. used their STAR-MPI [5] tuning platform assuming that a PAP does not change over consecutive executions of the the same operation (the operations were identified by the their code location and calling arguments). In papers [10,12,18] the authors included the detection mechanisms into their algorithms, which required additional communication or special hardware features. In [15,16] we proposed usage of an additional background thread to detect, predict and distribute PAT information over all cooperating processes while the computations are performed. As described above, Clairvoyant did not provide any out-of-the-box method of PAP estimation [9], which was our motivation to extend it with the background thread, see Sect. 6.

Recently, apart of the imbalanced PAPs subject [9], the focus of MPI reduce algorithms were mainly scoped on hierarchical optimization, where the processes are distributed over a compute cluster, and more than one process is assigned to the same node. In [7], Hasanov et al. proposed an algorithm categorizing the processes into the local (in the same node) and global (distributed over a compute cluster) communicators, where the reduce can be performed in two phases, increasing the performance of the operation. In [21], Shan et al. proposed utilization of idle threads on manycore processors and data compression to boost MPI reduce performance, and in [28], Zhao et al. showed that usage of k-nomial tree algorithms has the advantage over a typical binomial solution. These approaches exploit a priori knowledge about the hardware architecture and, therefore anticipate larger network latencies for the remote processes (assigned to different nodes); in comparison our solution uses its own mechanisms to retrieve knowledge about delays during the runtime, using the auxiliary, background thread.

The other, recent improvements of MPI reduce performance, were based on the additional hardware support, namely offloading the execution of the operation into FPGA (Field-Programmable Gate Array) [23,27]. The proposed solution moved the reduction calculations from the compute node into a network switch, where the FPGA is deployed, providing significant performance boost (up to 10$\times$ speedup). In comparison, our solution does not interfere the hardware layer and is network independent, however we can perceive that such support can be complementary to our approach and is taken under consideration in the future works.

4 Analysis and necessary modifications to the algorithm

The authors of [9] did not provide a reference implementation, and shared only a few implementation details like using pairing heaps to improve the performance. Therefore we attempted to implement the Clairvoyant algorithm ourselves. However, our implementation meet several edge cases neither handled by the pseudocode nor discussed in the article. In this section we present details of how we solved these issues.

4.1 Available processes and the state array

The referenced paper proposes to use a set of pairing heaps holding process indices, one for each segment, to optimize the linear search in the innermost loop of the algorithm. This set is accompanied by an array of handles to every (process, segment) pair in the heaps. This allows for a fast search for the process with the smallest rank that satisfies the condition $M(z, j) \in \{A, P\} \wedge S(z) = \perp$. Such a setup is efficient for handling changes in M array, as the changes translate to a single insert or erase into a pairing heap. After a send operation is scheduled for a process, its state is updated to $\top$. This means it should not be considered during following searches in the same round. However, the rank of this process may appear in every pairing heap. Therefore, it either must be removed from $O(N)$ pairing heaps (as in the worst case the process may be inserted into every segment's pairing heap). Alternatively, the search for a process with the smallest rank must filter out the processes that do not satisfy the condition $S(z) = \perp$. This, however, will require delete-min operation instead of find-min on pairing heap which requires $O(\log n)$ amortized time instead of $O(1)$ in case of delete-min. Additionally, at the start of the next round, the values in array S for every ready process are reset to $\perp$, which means reinserting the process rank to $O(N)$ pairing heaps.

Lemma 1 *There exist instances that require $O(PN^2)$ inserts and $O(PN^2)$ erases from the pairing heap during the entire execution of the Clairvoyant algorithm, where N is the number of segments and P is the number of processes.*

Proof There are N pairing heaps used in the algorithm—one per message segment—containing processes available for choosing. Process p will appear in pairing heap s if $M(p, s) \in \{A, P\}$ at the start of the round. Every process transmits every of its segments at least once. Since a process is allowed only one transmission per round, every process takes part in at least N rounds. The transmission of a segment is followed by setting $S(p) = \top$ for the process p that sends the data and makes the process unavailable for choosing in this round. If we choose to remove such process from pairing heaps immediately, then we must remove it from as many paring heaps as the process has segments in state A or P, which is N for the first transmitted segment, $N - 1$ for the second, and so on, for the total of

$$\text{erasures-per-process}_{\text{immediately}} = \sum_{i=1}^{N} i = \frac{(N+1)N}{2} = O(N^2)$$

erasures for process p. At the beginning of the next round, the process must be reinserted to all heaps for whose it still holds segment in A or P state. There will be one insertion less than there were deletions in previous round, because the transmission changes state of one segment. That leads to:

$$\text{insertions-per-process}_{\text{immediately}} = \sum_{i=1}^{N} i - 1 = \frac{N(N-1)}{2} = O(N^2)$$

insertions per process for a total of $O(PN^2)$ operations.

If we take path of filtering out processes during extraction, consider an instance with processes arriving one per round, i.e. $a_i = i$ for $i = 0, \dots, P-1$ and $d = 1$ and $P > N$. According to the pseudocode, after $O(N)$ initial rounds there will be $O(P-N) = O(P)$ rounds with regular communication structure (see Fig. 1). In every such round, N processes will transmit a segment and every segment will be sent by some process.

Last process considered in such round will have $N-1$ segments in state P or A: it became available in this round and immediately sent his segment 0 to process 0. At least $N-1$ processes were already considered in this round and all of them sent some segment thus being marked with $S[p] = \top$. These processes have $N-2, N-3, \dots, 1$ segments in state P or A so they were present in respectively $N-2, N-3, \dots, 1$ paring heaps. The last considered process will not find a peer, and to realize this it will iterate over $N-1$ pairing heaps filtering out all of disabled processes from them, erasing

$$\text{erasures-per-round}_{\text{filtering}} = \sum_{i=1}^{N-1} i - 1 = \frac{(N-1)(N-2)}{2} = O(N^2)$$

nodes.

At the start of the next round, $N-1$ processes transition back to $S[p] = \bot$ and must be added to respectively $1, 2, \dots, N-1$ pairing heaps, according to the number of segments in P or A state for a process. Additionally, a new process will arrive that must be added to N pairing heaps. Therefore, at the start of this round $O(N^2)$ insertions will occur:

$$\text{insertions-per-round}_{\text{filtering}} = N + \sum_{i=1}^{N-1} i = \frac{(N+1)N}{2} = O(N^2)$$

There will be $O(P-N)$ rounds following this schema so the total number of insertions and erasures will be $O(PN^2)$.

Thus we have shown that in either strategy, the number of insertions and erasures from paring heaps is in the order of $O(PN^2)$. $\square$

These operations are not only time-consuming but also require a significant amount of memory bandwidth. For example, when $N = P = 512$ and assuming 56 bytes per (process, segment) pair (see calculations in the Sect. 4.5), the total sum of memory transfer would be in the range of tens of gigabytes.

4.2 Additional state P'

Take an instance with $P = 4$ processes with arrival times of $a_0 = a_1 = a_2 = 0$, $a_3 = 1.1$, $r = p_0$, $N = 4$ segments and round time $d = 1$. In the first round at $t = 0$, processes p_0 and p_1 will exchange messages with p_0 sending segment 1 to p_1 and p_1 sending segment 0 to p_0. Notice that p_2 will neither send nor receive a segment in this

	p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}	...
s_0		1	2	3	4	5	6	7	8	9	10	**11**		
s_1	1	2	4	6	8	10	**11**							
s_2	2		3	4	5	6	7	8	9	10	**11**			
s_3	3	3		5	6	7	8	9	10	**11**				
s_4	4	4	5		7	8	9	10	**11**					
s_5	5	5	6	7		9	10	**11**						

Fig. 1 Example of first 11 rounds of a scenario producing $O(PN^2)$ erasures from pairing heaps. $N = 6$. Column indexes are process numbers, row indexes are segment numbers. Value r in cell (s, p) signifies that process p sends segment s in round r. This also means that the cell (s, p) transitions to state E in round r. Empty cells are in A or P state

round. During the next round, starting at $t = 1$, p_3 finally arrives. The process p_0 will be considered first and will be scheduled to receive segment 0 from p_2. Next step will determine that p_1 should receive segment 1 from p_3. The next two steps will cause a difficult situation: p_2 will be scheduled to receive segment 1 from p_1 and then p_3 should receive segment 0 from p_0, causing both p_0 and p_1 to send the same segment they receive in one round.

This is an issue. If we want the algorithm to remain deterministic, we must synchronize these two transmissions: either the segment is sent after the receive or is received after the send. This, however, requires doubling the round time for the entire algorithm as some transmissions can no longer be be asynchronous.

We propose to solve this issue by introducing an additional state P' in the M array. $M(i, j) = P'$ represent the case when a process j already received segment i in this turn. When a segment is in this state, it must not be sent in the same round this state was set.

Since any process can receive only one segment in one round, only one cell per process in the state array can be in P' state. Moreover, this state is equivalent to state E, except for the case when the algorithm searches for a process that will send a segment to the sink (i.e. the root or the first ready process in the round). In this case P' should be treated as a distinct state. We will be able to properly differentiate states P' and E, if we implement P' state by replacing it with state E in M array and storing, in additional per-process variable, the segment index of P' state. Essentially, setting $M(s, p) = P'$ is equivalent to setting $M(s, p) = E$ and setting an additional array $P'_{\text{states}}(p) = s$. A test for a segment to be in E state becomes $M(s, p) = E \wedge P'_{\text{states}}(p) \neq s$. After the round is completed, state P' becomes state P again, so $M(P'_{\text{states}}(p), p) = P$ must be executed for every p for which $P'_{\text{states}}(p)$ was set.

4.3 Issues with pseudocode

A few cases are not properly handled in the algorithm's pseudocode. The innermost loop does not recognize the case where a (process, segment) pair cannot be found. There will be processes that will not communicate in a round despite being able to, e.g. see the example in the original paper, or the first round of the example in Sect. 4.2. Therefore, the loop must terminate not only when a segment has been found but also when all segments have been considered. This change also has consequences in the following steps. Since such a pair may not be found, modifications of array M and enqueueing to the result queues must be done only when the search was successful.

Yet another consequence of possibility of not participating in communication is that the result of the algorithm, i.e. the input and output queues, must include not only the segment number and the process to communicate with but also the round number. Otherwise, in presence of rounds without communication, processes may not properly recognize time to communicate. For example, consider process p_0 that should receive segment s from process p_1 in round $k+1$ and process p_1 that sends segment t to p_2 in round k and segment s in round $k+1$. In round k process p_0 sees (p_1, s) in its input queue, while process p_1 sees (p_2, t), (p_1, s) in its output queue. Without knowing the round number, p_0 cannot determine whether to communicate with p_1 in round k or $k+1$ and in round k will try to receive a segment that will not be sent until round $k+1$.

4.4 Deterministic priority queue for Q

It should be noted that the priority queue used for Q must be deterministic. Specifically, handling of keys of the same value must be identical across all machines, which rules out such techniques as random priorities or using value-of-pointer for symmetry breaking.

4.5 Cache friendliness impact on complexity

The authors compute the complexity of their algorithm to be $O(RPN)$, where N is the number of segments, P is the number of processes and R denotes the number of rounds, estimated in the article to be $\log_2 P + N + 1$ in the balanced PATs case. Additionally, the authors ignore the N component in the complexity on the basis of small value and beneficial cache effects, ending up with $O(PN + P\log^2 P)$ steps. However, the algorithm is far from being cache-friendly. The suggested implementation accompanies state matrix M with a number of `pairing_heap` queues. In the worst case, every cell of M will have an accompanying node in some pairing heap. Additionally, a handle to such a node will also be stored for faster erases. Assuming x86-64 machine this sums up to 56 bytes per single state in the M matrix: 48 bytes for `pairing_heap` node with value type of `std::size_t` and another 8 bytes for handle (using gcc 7.3.0 with -O3 and boost 1.70.0). This size is close to the typical size of a cache line (64 bytes) for x86-64 architecture so different nodes will occupy different cache lines. Moreover, any operation that changes the pairing heap requires access to at least two nodes of the structure, which puts even more pressure on the cache memory.

Note that to account for the required additional operations described in the Sect. 4.1, a $O(N^2 P \log P)$ component should be added to the above complexity. An additional component of $O(RN \log P)$ must be added to account for the idle loops (see Sect. 5.1) if the round time is significantly shorter than the PAT differences. Moreover, in the worst case, the number of idle rounds can grow exponentially with the input size as it will be at least $R \geq \frac{\max_i a_i}{d}$.

Therefore, we believe that the complexity of the Clairvoyant algorithm should be stated as $O(N^2 P \log P + RN)$, with $N^2 P \log P$ being the number of steps required to manage the pairing heaps (and this number dominates other operations in the algorithm) and RN component accounting for the possible large number of idle rounds.

4.6 The algorithm with fixes applied

Listing 2 presents a pseudocode for the Clairvoyant algorithm after applying the above fixes. The original line numbering have been preserved in most cases, but the required condition to check if the peer has been found has moved line 25 to line 26 lines 30–33 to lines 33–36 and lines 35–41 to lines 39–45.

5 Improvements to the algorithm

The benchmarks presented by Marendic et al. show an impressive speedup achieved by the Clairvoyant algorithm over other reduce algorithms in an imbalanced PAP environment. We have implemented two versions of Clairvoyant with our fixes applied. The base version replicates the pseudocode. The pairing version uses `pairing_heaps` from the boost library. Unfortunately, neither implementation was able to match the performance claimed in the original article. While [9] cites a time of about $0.125s$ for generating a schedule for an instance with $P = N = 512$ and unspecified arrival times on a Xeon E5-2680@2.7 GHz CPU.[1] With our implementation we were able to reach schedule generation time of about $10s$ for the same sized instances with randomly generated arrival times. Our tests were run on a different CPU, a Xeon E5-2670 v3 @2.3 GHz, but such a large run time difference cannot be explained by neither hardware nor compiler or implementation differences alone. We assume that the main reason for this disparity is the addition of the updates of the pairing heaps after disabling and enabling processes.

We were also able to identify a few places where the algorithm can be improved. Therefore, we implemented yet another version of the algorithm that employs the improvements, i.e. uses a reduced set of segment states, stores the state array in segment tree, has simplified handling of set of processes that are active during a round and skips unnecessary computations in rounds with only one active process. These improvements allowed our implementation to reach the run times comparable to the ones presented in [9]. Moreover, the changes caused the algorithm's run time to be much less sensitive to process arrival time patterns. The results of computational experiments supporting these claims are presented in Sect. 5.7.

5.1 Skipping idle rounds

The original Clairvoyant algorithm does not handle well cases where the round time is significantly smaller than the differences between processes arrival times. An exam-

[1] As mentioned in [9], this is the CPU on which the experimental results were obtained, but authors mention also a system with Xeon X5660@2.8 GHz CPU, used as a prototyping and testing environment.

Listing 2 Clairvoyant algorithm with applied fixes from Sect. 4. In line 19 an additional termination condition has been added to guarantee the loop terminates, the original line 25 (now line 26) is guarded by a check if the previous search was successful, the original lines 30–33 (now lines 33–36) are executed only when a proper segment has been found, and finally, in line 36 (original line 33) M array indices have been fixed. The addition of the conditional instruction moved the original lines 35–41 to lines 39–45.

```
1:  Let $M$ be a state matrix of size $P \cdot N$. Set $M(*, *) = $ A, i.e.. mark all segments as available.
2:  Set $S$ : bool be an array of size $P$
3:  For $\forall i \in \{0 \dots P-1\}$ insert process rank $i$ into a priority queue $Q$, where priority is given to process
    rank $i$ with smaller arrival time $a_i$
4:  while size($Q$) $> 1$ do
5:      Pop processes $Q_0 \dots Q_k$ from $Q$ to form a sorted vector $G$, so that $\forall i \in G, a_i \leq a_{Q_0} + d$
6:      for $\forall i \in G$ do
7:          let $S(i) = \bot$
8:      end for
9:      if $r \in G$ then
10:         insert $r$ at first position in $G$
11:     end if
12:     for $\forall i \in G$ do
13:         let $j = 0$, let $z = \phi$
14:         if $r \in G$ then
15:             let $r^* = r$
16:         else
17:             let $r^* = Q_0$
18:         end if
19:         while $z \in \phi$ and $j < N$ do
20:             if $i \neq r^*$ then
21:                 starting from $j$, find first segment $x$, such that $M(i, x) \in \{A, P\}$. Let $j = x$.
22:             else
23:                 starting from $j$, find first segment $x$, such that $M(i, x) \in \{A, P, E\}$. Let $j = x$.
24:             end if
25:             if such segment $j$ has been found then
26:                 perform linear search through group $G$ to find the first process $z \neq i$ such that $M(z, j) \in$
                    $\{A, P\} \wedge S(z) = \bot$.
27:             end if
28:             if $z \in \phi$ then
29:                 let $j = j + 1$
30:             end if
31:         end while
32:         if $j < N$ then
33:             enqueue to $I(i)$ the tuple $(round, z, j)$
34:             enqueue to $O(z)$ the tuple $(round, i, j)$ and let $S(z) = \top$
35:             $M(z, j) = E$
36:             $M(i, j) = P$
37:         end if
38:     end for
39:     for $\forall i \in G$ do
40:         if $\exists j \in \{0 \dots N-1\} : M(i, j) \neq E$ then
41:             $a_i = a_i + d$
42:             push process $i$ into $Q$
43:         end if
44:     end for
45: end while
```

ple of such a case would be a single segment, $1ms$ round time and arrival times $a_0 = 0s, a_1 = 1s, a_2 = 2s, a_3 = 3s$. After each process arrival, a few rounds of computation will be spent to communicate with this newly arrived process. When all required messages are passed, the algorithm will begin idling with group G of size 1. However, during such idle loops, entire M array will be scanned for a candidate process. Therefore, a lot of time will be wasted in a search for a process that does not exist.

We propose to update the algorithm in such a way that when vector G is calculated and its size is equal to one, instead of the usual loop, the algorithm looks for the next process P to arrive and jumps ahead to the round number in which process P finally arrives. The arrival of a new process is the only event that can change the size of G in this situation, so the jump will not interfere with the flow of the algorithm. Looking at the above example, at $t = 0s$ process p_0 arrives. Instead of doing 1000 of idle loops, the algorithm notices the case when $|G| = 1$ and jumps ahead $(a_1 - a_0)/1ms = 1000$ rounds. In the next step, $G = \{p_0, p_1\}$. In a single step p_1 sends its segment to p_0 and leaves G. In the following step the algorithm can again notice the single element in vector G and instantaneously skip another 999 rounds.

5.2 Generating vector *G* without priority queue

Let us say that process p is active when its arrival time has been reached and p has not yet sent out all its segments. Let the availability time of a process p be the time the process arrives if the process did not become active yet. After the process became active, let the availability time denote the time p becomes available for the next round. During the execution of the algorithm the availability time of p starts with the value of its arrival time and gets increased by d every round the process is active.

The original algorithm uses a priority queue Q to create a vector of processes that will be considered during the next turn (the vector G). This queue contains all processes. Every round a process is active it is removed from the queue and reinserted into Q, except the last round the process is active, when it will be only removed from Q. This gives the total cost of $O(R_\Sigma \log P)$, where $R_\Sigma = \sum_{p=0,\dots,P-1} R_i$ and R_p is the number of rounds in which process p is active. We propose an alternative solution that reduces this cost to $O(R_\Sigma + P \log P)$ and does not require any complex data structures.

The problem of generating the set of active processes can be modeled as follows. There are given P processes with availability times a_p for $i = 0, \dots P - 1$ and round time d. A computation is performed in R rounds. During each round of computation a GET-ACTIVE query is issued. The result of this query must be the list of processes that have availability time $a_p \leq a_{min} + d$ where a_{min} is the smallest availability time among not completed processes. The query must return processes ordered by their availability time. Next, every process on this list will perform a COMMUNICATE operation, and some of them will also perform COMPLETE:

– COMMUNICATE adds d to its availability time,
– COMPLETE removes the process.

Every process is guaranteed to eventually perform COMPLETE.

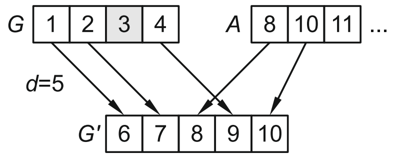

Fig. 2 Example of generating a new vector (G') from the previous one (G) and the awaiting queue A. The round time is $d = 5$. The squares represent processes and values in squares denote the availability time of every process. The gray background marks a process that became inactive and should be removed from G

We propose the following solution to the problem. Let G be the result of GET-ACTIVE from the previous round. Let A (awaiting) be a queue of processes ordered by their arrival time with ties broken by a process rank. Initially, all processes occupy A queue and G queue is empty.

Next, consider list G—the result of GET-ACTIVE query from the previous round. Processes in G are ordered by their availability times (again, with ties resolved by process ranks). Every one of them either should be removed from G (due to call to COMPLETE) or had their availability time increased by d as a result of calling COMMUNICATE. Let G' be the result of the next GET-ACTIVE query. The relative order of processes that were in G and will be included in G' will not change as all of them called COMMUNICATE. However, we may need to weave into the G' some processes from the A queue that will become available (see Fig. 2). This can be done by a simple merge operation, not unlike the one used in the merge-sort algorithm. First, assume that looking at the head of G means removing from G's head any process that should be removed from G due to calling COMPLETE, until we find a process that is to stay. This process becomes the observed head. Now all we need to do is select either the head of G or the head of awaiting queue, depending on the smaller availability time, as the first element of the G'. Call this process the *head*. Now we repeatedly select the head from the G or A queue based on the earlier availability time, as long as the selected head's availability time is no later than $a_{head} + d$. This produces G' of processes, ordered by their availability time for the next round, i.e. the result of the following GET-ACTIVE query.

The COMMUNICATE and COMPLETE operations can be easily implemented by holding the availability time and completion flag of processes in an array indexed by the process rank.

Lemma 2 *The computation with P processes and R rounds, where each round calls* GET-ACTIVE *once, a process is active in R_p rounds, calling* COMMUNICATE *in every round and calling* COMPLETE *after R_p rounds takes $O(R_\Sigma + P \log P)$ time, where $R_\Sigma = \sum_{p=0,\dots,P-1} R_i$.*

Proof Every process will be added to the awaiting queue exactly once and will be removed from awaiting queue at most once during entire computation. The initial setup of awaiting queue requires ordering all processes which takes $O(P \log P)$ time. If queues are implemented as lists the construction of the queue in the given order consumes $O(P)$ time. The removal of process from queue takes $O(1)$ time as it is always removed from the head. Both COMMUNICATE and COMPLETE require

$O(1)$ time as each of them require one update of an array and there is a total of R_Σ calls to COMMUNICATE and P calls to COMPLETE. The GET-ACTIVE query takes constant time per process. One query handles all processes from list G and some from A queue. As no process is re-inserted to the A queue, GET-ACTIVE will handle P processes from A queue during the entire computation. Because process p will be inserted and removed from list G R_p times, GET-ACTIVE will handle a total of R_Σ processes from $LAST-ACTIVE$ queue.

Therefore, in the presented computation the number of performed steps will be $O(P \log P + P + R_\Sigma) = O(R_\Sigma + P \log P)$. If R is the total number of rounds of the algorithm and in every round only P processes can be active, and the total number of rounds will be $R = \Omega(\log P)$ we can also state this time as $O(RP)$. □

5.3 Compressing the state array

The Clairvoyant does not differentiate between states A and P. Therefore, only states P and E must be encoded in the state array. Since, as was mentioned before, additional state P' can be encoded in an external variable (see Sect. 4.2), state array M requires only a single bit for every (process, segment) pair.

5.4 Segment trees

In this section, the process index will denote its zero-based number in a sequence obtained by ordering processes according to certain criteria. We call this number an index because it is used to index the state array. Additionally, wherever we mention a specific order of processes, ties in ordering will be broken by process ranks.

In the inner-most loop the Clairvoyant algorithm tries to find a peer for every process in the active set. This peer must be free, i.e. it has not sent a segment in the considered round. Additionally, both the process and its peer must have a common segment such that none of them have sent—the state of this segment for both of the processes must be A or P. When there are multiple processes that can become a peer, the one that provides a segment with the smallest index is chosen. When tied, the peer with the earlier availability time is selected. There is an exception for the head of G, as it accepts any segment except the one in P' state. Its peer, however, must still have the segment in A or P state.

The authors of the original paper propose to solve peer finding by a linear search (in the pseudocode) or by using pairing heaps. Our computational experiments suggest that this route is inefficient, especially for larger instances. Our tests show that the implementation using pairing heaps requires more time to complete than the naive implementation with linear search. We propose another approach to this problem that employs segment trees [25].

A segment tree is a binary tree where leaf nodes store data and internal nodes hold the aggregate information from all its children. For example, when the aggregation function is the minimum, each internal node contains the minimum of its two children. Since these children contain minima of their children and so on, every internal node knows the minimum of the whole subtree rooted at the node. In consequence, the root stores the minimum of the entire range of data, its left (right) child stores the minimum of the left (right) half of the data and so on.

Many operations on a segment tree, such as finding an aggregate information over a continuous subset of data or updating a single value, can be done in the number of steps proportional to the height of the tree, i.e. in $O(\log n)$ time.

Although the segment tree is a tree, it can be efficiently stored in memory without using pointers. Details of such implementation can be found in e.g. [4]. We place the nodes of the tree in an array, starting with the root at index 1. Next we place its left and right child, then the children of the root's left child, the children of the root's right child, then the children of the previous nodes, from left to right, and so on. Such an order allows us to calculate the position of the parent of the node at index i as parent $= i/2$ while the left and right children of i are given by left-child $= 2i$, right-child $= 2i + 1$. The root is always at the index 1. If a tree has n leaf nodes, let $m = 2^k \geq n$ for the smallest integer k. If leaves are numbered from 0, a leaf number can be translated to a node number as follows: leaf-node$(number) = m + number$.

5.4.1 Finding peers with bitwise operations

Let $M(s, p)$ be state array, where p is the process index and s is the segment index. As we mentioned earlier, only one bit for every state in M array is needed, therefore M is an array of single bits. Let the value of 1 represent state P and 0 denote state E.

The problem of finding a peer for the considered process p can be expressed as finding the leftmost bit set to 1 in the row s with the smallest index in M for which $M(s, p) = 1$. More formally, the problem to be solved is as follows. There is a two-dimensional array with bit variables $M(s, p) \in \{0, 1\}$ for $p = 0, \ldots, P - 1$, $s = 0, \ldots, N - 1$. Our goal is to answer two types of queries:

- FIND-PEER$(M, q) = \min_{M(s,p)=1, p\neq q, M(s,q)=1}(s, p)$ i.e. find a lexicographically smallest pair (s, p) such that $M(s, p)$ and $M(s, q)$ are set and $p \neq q$,
- FIND-PEER-ROOT$(M, q) = \min_{M(s,p)=1, p\neq q}(s, p)$ i.e. find a lexicographically smallest pair (s, q) such that $M(s, p)$ is set and $p \neq q$.

Between queries the array M can be updated in a following way:

- CLEAR(M, p) sets $M(s, p) = 0$ for $s = 0, \ldots, P - 1$ (corresponds to disabling process with $S(p) = \top$),
- SET(M, p, V) sets $M(s, p) = V(s)$ for $s = 0, \ldots, P - 1$ (corresponds to re-enabling process with $S(p) = \bot$ where V represents previous state of segments),
- CLEAR(M, p, s) sets $M(s, p) = 0$ (corresponds to changing single segment state to E or P'),
- SET(M, p, s) sets $M(s, p) = 1$ (corresponds to restoring segment to state P from state P'),

Let us split FIND-PEER query into two steps. We will find an appropriate row (segment) first, then we will find the required process, see Fig. 3.

Segment s (or a row s of M) is eligible for selection when there is a process p' different from p that has $M(s, p') = 1$ and $M(s, p) = 1$. We can check the existence of such a process p' by calculating a logical sum over all elements of the row s, except

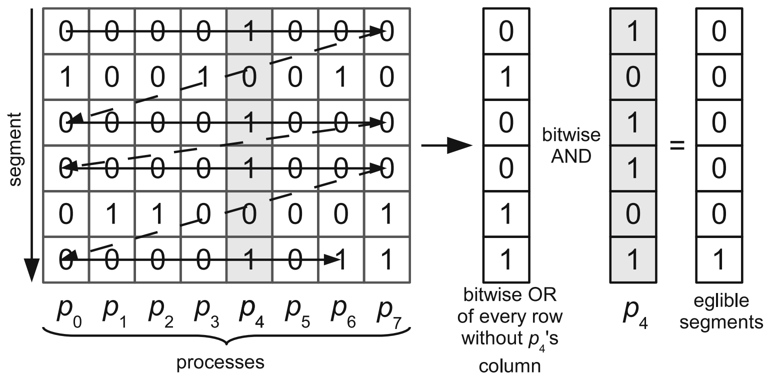
segment
processes
p_0 p_1 p_2 p_3 p_4 p_5 p_6 p_7
bitwise OR of every row without p_4's column
bitwise AND
p_4
=
eglible segments

Fig. 3 Order of iteration over the state array during a search for a peer for a process p_4. 0 encodes E state while 1 represent state P. Rows (segments) for which p_4 has state E set are skipped

the element at index p, i.e. $\bigvee_{i=1,\ldots,P,i\neq p} M(s,i)$. Let us calculate these sums for every row and perform an element-wise logical product of the result with the column p of M, i.e.

$$S = M(*,p) \wedge \bigvee_{i=1,\ldots,P,i\neq p} M(*,i).$$

This gives us a bit vector S of eligible rows where element $S(s)$ decides whether segment s can be selected. Now find the smallest index i such that $S(i) = 1$. This index corresponds to the segment that should be selected.

5.4.2 Finding a peer in a segment tree

The naive implementation of computing logical sums would be no more efficient than the linear search. However, here we can take advantage of instruction level parallelism of bitwise instructions in modern processors. Using SIMD instructions we can calculate up to 256 or 512 logical sums of single bits in one CPU instruction (depending on the instruction set). Moreover, using segment trees, we are able to reduce the number of operations required to compute the sums by a factor of $P/\log P$.

We do this by building a segment tree over the columns of the matrix M. Columns of M are leaf nodes of such a tree. Internal nodes hold an element-wise logical sum of both of its children. An example of such tree can be seen on Fig. 4. Having such a tree constructed, the logical sum of all columns except one can be computed using $O(\log P)$ element-wise logical sums of columns. We start in the leaf we want to omit, with the result initialized to a neutral value (all zeros in our case). Now, we climb up to the root, in each step combining (performing element-wise logical sum) the current result with the value in the sibling node (see Listing 3).

all-except-one performs $O(\frac{N}{\mathrm{SIMD}} \log P)$ steps. $\log P$ comes from the fact that the outer loop iterates over the height of our segment tree (which is $O(\log P)$). In every iteration we perform an bitwise-OR of two vectors of length N, which can be done in

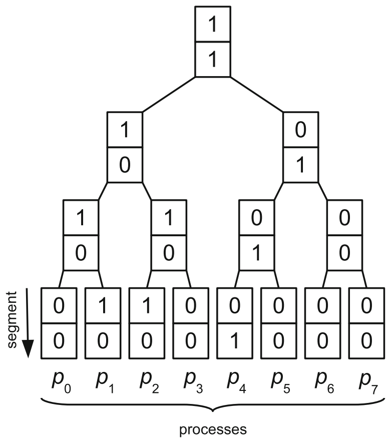

Fig. 4 Segment tree built over a state matrix. There are 8 processes and 2 segments. Every internal node holds a bitwise OR of its children

Listing 3 all-except-one(T, $node$, $initial$) function. Given segment tree T, selected leaf $node$ and $initial$ value calculate the bitwise-OR of $initial$ value and all nodes except the $node$.

```
$value = initial$
while $node \neq root$ do
    $sibling = sibling(T, node)$
    if $sibling$ is the right-hand child of its parent then
        $value$ = bitwiseOR($value$, $T[sibling]$)
    else
        $value$ = bitwiseOR($T[sibling]$, $value$)
    end if
end while
return $value$
```

$\frac{N}{\text{SIMD}}$ instructions where SIMD denotes the width of a single bitwise operation on a chosen CPU.

Finally, we perform the element-wise logical product of the result with the column of process p and find the smallest index of a non-zero element in the result. The last step requires a linear number of steps but can be sped up with specialized hardware instructions that count leading or trailing zeros (e.g. `BSF` or `TZCNT` on x86 architecture).

$$\begin{aligned}&\text{FIND-PEER-SEGMENT}(M, p)\\&\quad = \text{index-of-first-bit-set}(\text{bitwiseAND}(\text{all-except-one}(T_M, p, [0, \ldots, 0]), M(*, p)))\end{aligned}$$

where T_M is the segment tree build for array M.

As we want to employ instruction-level parallelism during these computations, we must ensure that columns occupy consecutive bits in memory. This can be achieved by simply storing the M array in process-major order i.e. storing all segments for a given process in N consecutive bits and enforcing data alignment required by used SIMD instruction set.

The FIND-PEER-ROOT query can be handled in a similar way. When we are looking for the peer for the G's head, we must consider processes in P or E states, not only in P state. Fortunately, it is enough not to perform the final bitwise AND to not exclude the segments in E state. However, we must check if the root process has a segment in P' state and exclude this segment, as such segment cannot be transmitted.

$$\begin{aligned}&\text{FIND-PEER-ROOT-SEGMENT}(M, p)\\ &\quad = \text{index-of-first-bit-set}(\text{bitwiseAND}(\text{all-except-one}(T_M, p, [0, \dots, 0]), \text{MASK}_q))\end{aligned}$$

where

$$\text{MASK}_q(i) = \begin{cases} 0 & \text{if } i = P'_{\text{states}}(q) \\ 1 & \text{for } i = 0, \dots N-1, i \neq P'_{\text{states}}(q) \end{cases}$$

When the proper segment (or row) is selected, we must find the process with the smallest index that will become the peer. This means we must scan the row from left to right looking for the first bit that is set. Since we organized the table in a column-row order, this search is more complicated than the previous one and cannot be sped up by hardware instructions. However, we can use the already built segment tree to perform the search in $O(\log P)$ steps. The algorithm takes advantage of the fact that the internal nodes hold the logical sum of its children. Consider a single index i in a bit vector. If there is one on i-th position in an internal node then there exists a leaf in the subtree rooted at this node that has one at i-th index.

Start at root. The root must have 1 on s-th position, where s is the selected segment. By applying the above reasoning, if the left child has one on s-th position then the leftmost leaf in the segment tree with 1 on s-th position must be in the left subtree, otherwise it will be in the right subtree. Repeat the process until a leaf is reached (see Listing 4).

Listing 4 find-leftmost(T, $root$, $position$) algorithm. Given segment tree T, node $root$ and bit index $position$ within the node value, find the leftmost (i.e. the one with the smallest index) leaf node that has the bit at index $position$ set.

```
if not $root.bit[position]$ then
    return $\phi$
end if
$node = root$
while $node$ is not leaf do
    if left-child$(T, node).bit[position] = 1$ then
        $node$ = left-child$(T, node)$
    else
        $node$ = right-child$(T, node)$
    end if
end while
return $node$
```

We will also need a modification of the Listing 4 that finds the leftmost leaf equal to 1, starting from (and excluding) a given leaf. This can be easily solved by backtracking from the leaf to the root and performing `find-leftmost` on the right sibling of the considered node if the sibling equals to 1.

Listing 5 find-leftmost-from(T, $from$, $position$) function. Given segment tree T, leaf node $from$ and bit index $position$ within the node value, find the leftmost (i.e. the one with the smallest index) leaf node that has the bit at index $position$ set and its node number is greater than $from$.

```
node = from
while node is not root do
    if is-left-child(node) and right-sibling(T, node).value[position] then
        return find-leftmost(T, right-sibling(T, node), position)
    end if
    node = parent(T, node)
end while
return $\phi$
```

The algorithm presented in Listing 5 will become useful in cases when an application of Listing 4 will find the same process we are finding a peer for. In this case we must perform another search, looking for the process that is to the right from the process found in the previous search.

Finally, we obtain:

$$\text{FIND-PEER}(M, q) = \begin{cases} \text{leftmost} \;\; \text{if } q \neq \text{leftmost} \\ \text{find-leftmost-from}(T_M, q, \text{segment}) \text{ if } q = \text{leftmost} \end{cases}$$

where

$$\text{segment} = \text{FIND-PEER-SEGMENT}(M, q)$$

and

$$\text{leftmost} = \text{find-leftmost}(T_M, 0, \text{segment}))$$

5.4.3 Proper order of processes

There are still two problems to solve. The first one is to properly order the processes. The algorithm requires that the leftmost process will be the one with the earliest ready time. The second problem is updating the tree. Processes lose their ready state after sending a message and become unavailable for selection as a peer. Moreover, we have introduced P' states that are represented by 0 in M array, but revert to state P (1 in M) after the round ends. Both of these cases require updates in the segment tree.

For the first problem, recall that processes are already ordered by their ready times in vector G. Moreover, their relative order will not change in time because the ready

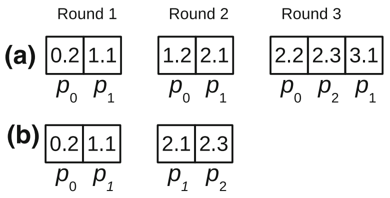

Fig. 5 Order of processes in vector G depends on the head of G. Squares represent processes and numbers within squares denote their availability times. Arrival times are given by $a_0 = 0.2$, $a_1 = 1.1$, $a_2 = 2.3$ and round time $d = 1$. In case a, p_0 stays active during all three rounds. In case b, p_0 becomes inactive after round 1 and leaves vector G. Consequently, while p_1 follows p_2 in case a, in case b p_2 follows p_1. Notice that in every case the order of processes is the same as some circular shift of the order of their fractional times, i.e. $p_1(0.1)$, $p_0(0.2)$, $p_2(0.3)$

times are advanced in steps of an equal size. However, processes may change their position in vector G. Take 3 processes with arrival times $a_0 = 0.2$, $a_1 = 1.1$, $a_2 = 2.3$ and round time $d = 1$. In the first round vector G contains p_0, p_1 in that order. Assume that after round 1 process p_0 leaves group G. In the next round G contains p_1, p_2, but if p_0 had stayed in G, p_1 would be placed after p_2 (see Fig. 5). This means that process placement in G depends on the head of G.

To resolve this issue, order the processes with respect to their "fractional" arrival times, that is by f_i such that $a_i = d * k + f_i$ where a_i is the process arrival time, d is the round time, k is a non-negative integer and $0 \leq f_i < d$. Essentially, f_i is the remainder of dividing a_i by d. Now every process is assigned two indexes: i and $i + P$, where i is the position in the above order. When a ready process in group G has a smaller index i than the head of G, we use index $i + P$ for that process instead of i. This operation doubles the size of the segment tree, but it adds just one step to the cost of operations, since the cost depends on the height of the tree.

The Fig. 6 shows the final shape of the segment tree, along with a sample search for a peer.

5.4.4 Updates to the segment tree

All CLEAR and SET operations modify the content of the M array and require updating the segment tree. Is updated as follows. When process p becomes unavailable, i.e. after sending a message when CLEAR(M, p) is issued, the contents of its column is saved in a separate location and the column in the segment tree is zeroed. This requires updating nodes of the segment tree on path from the updated leaf to the root, which takes $O(\log P)$ element-wise logical sums on columns. When the process becomes ready again (SET(M, p, V), where V is the saved column), the content of its column is restored and the tree is updated again, following the same path. Bear in mind that all processes become ready at the same point in the algorithm so we can rebuild the entire segment tree at once instead of updating after every change, achieving $O(P\frac{N}{\text{SIMD}})$ steps for the entire batch of SET operations, instead of $O(P\frac{N}{\text{SIMD}}\log P)$. Replacing state P' with P also leads to the tree update but since only one bit is changed we need only $O(\log P)$ logical operations on single bits per every update, for a total of $O(P \log P)$ steps as no more than P processes can have a segment in P' state.

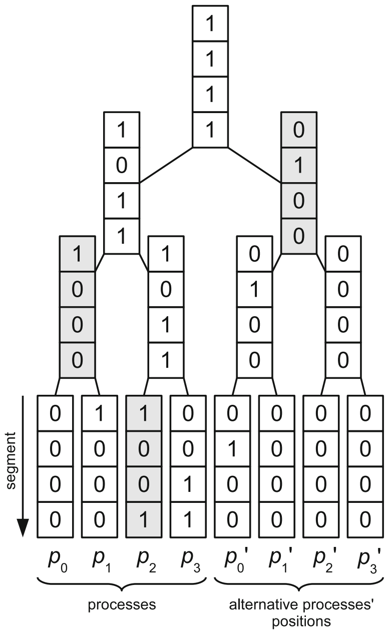
segment
p_0 p_1 p_2 p_3 p_0' p_1' p_2' p_3'
processes
alternative processes' positions

Fig. 6 Example segment tree T for processes p_0, p_1, p_2, p_3 with 4 segments. In this example, process p_1 has the lowest activation time so p_0 uses alternative position (denoted by p_0') in the segment tree. All internal nodes store bitwise OR of its children. In the example, we look for a peer for p_3. Nodes with gray background are the nodes that will be OR-ed while calculating all-except(T, p_3) resulting in a vector [1, 1, 0, 1]. This vector will be masked by p_3's vector ([0, 0, 1, 1]) resulting in [0, 0, 0, 1] which tells us that segment 3 will be selected. To find the peer process for p_3 start at root and in each step descend left if the left child has the last bit set (this is the bit corresponding to segment 3). This will lead us to the left, right and finally left to process p_2. Therefore, the peer for p_3 will be p_2 with segment 3

5.5 Complexity

Lemma 3 *The total complexity of the algorithm can be given by $O(RP\frac{N}{\text{SIMD}}\log P)$, where R is the number of rounds and SIMD is the available width of bitwise vector instructions (e.g.* 256 *in case of AVX2 instructions or* 64 *when using* 64*-bit registers).*

Proof Let R be the number of rounds performed by the algorithm. In every round a loop considers every active process once. Since there are P processes, the loop will iterate at most P times.

Before the loop, active processes are re-enabled. As this can be performed by a simple copy operation on a set of bits and there are at most P active processes, it will take $O(P\frac{N}{\text{SIMD}})$ steps. Another action before the loop is replacing P' states with state P. Again, only up to P processes will require this action and it will take $O(\log P)$ time per process as it is a simple update of a segment tree. To sum up, the loop preparation takes $O(P\frac{N}{\text{SIMD}} + P\log P)$ steps.

In every iteration (for every process) $O(\frac{N}{\text{SIMD}}\log P)$ steps are required to execute operation Find-peer-and-segment and another $O(\frac{N}{\text{SIMD}}\log P)$ to update the segment tree after disabling the process. The preparation of the set of active processes will take

$O(RP)$ steps over entire run of the algorithm (Lemma 2). Therefore, the complexity of the algorithm is $O(R(P\frac{N}{\text{SIMD}} + P\log P + P\frac{N}{\text{SIMD}}\log P) + RP)$ or $O(RP\frac{N}{\text{SIMD}}\log P)$ after removing dominated terms. □

Replicating the estimation from [9] where $R = O(\log P + N)$ in the case of balanced PATs leads to the complexity of $O(\frac{N}{\text{SIMD}}NP\log P)$ which is the same as the complexity of the Clairvoyant version employing the pairing heaps. However, in this implementation, we are able to reduce the required time by a factor of $\frac{1}{\text{SIMD}}$ by using instruction-level parallelism.

The main component of the memory complexity is M array which has PN cells. Taking into account alternative indices (that double the number of cells), the segment tree which, again, doubles the memory usage and the save-space for enabling/disabling processes, we end up with a total of 5 bits per cell of M. Compared to the original implementation, where every cell used 56 bytes (or 448 bits), we achieved an almost 90-fold reduction in memory usage.

5.6 Final algorithm

Here we present a pseudocode for the improved version of the algorithm. First, we extract the round initialization to a separate function. Function `merge-and-clean` creates vector G for the next round based on G from the previous round and the awaiting queue A. The function also removes from G all processes that have completed their part in the algorithm (hence the `-clean` suffix).

Listing 6 Merge-and-clean(G, A) algorithm. Given vector G from the previous round and the awaiting queue A, calculate vector G' of processes that will take part in the next round.

```
1: let next candidate be the head of $G$ when $a_{G_{head}} < a_{Q_{head}}$ and head of $A$ otherwise
2: let $G' =$ empty list
3: while $G$ is not empty or head of $A$ has $a_{Q_{head}} < a_{G_{head}} + d$ do
4:     extract next candidate $c$ from its queue
5:     if $c$ has all segments in state $E$ then
6:         continue                                  ▷ do the cleanup: do not append to $G'$
7:     end if
8:     if $(a_c \mod d) < (a_{G'_{head}} \mod d)$ then
9:         let $index = c + P$
10:    else
11:        let $index = c$
12:    end if
13:    append $c$ at the end of $G'$ using $index$ as $c$'s index in $M$
14: end while
15: return $G'$
```

Function `find-peer-and-segment` is used to find an appropriate candidate process for communication and a segment to transmit.

Listing 8 presents the final algorithm. The pseudo-r process requires an explanation. Since we order processes according to their fractional arrival times, the process r (i.e.

Listing 7 Find-peer-and-segment(T, G, p) function. Given segment tree T over the M set, and vector G, find a peer for the process with index p to communicate with and a segment index that will be transmitted.

```
let $column$ = all-except-one($T$, leaf-node($p$), $[0, \dots, 0]$)
if $p$ is not at the head of $G$ then
    $column$ = bitwiseAND($column$, $M(p, *)$)
else
    if $\exists j (p, j) \in P'$ then
        set $column[j] = E$
    end if
end if
$segment$ = index-of-first-bit-set($column$)
if $segment$ was not found then
    return $\phi$
end if
$peer$ = find-leftmost($T$, 1, $segment$)
if $peer = p$ then
    $peer$ = find-leftmost-from($T$, leaf-node($p$), $segment$)
end if
return $peer$, $segment$
```

the root of reduce operation) may not be at the beginning of the G group, as the Clairvoyant requires. We solve this by inserting another process: the pseudo root. If the root process is active and is not the first process in G, then pseudo-r is inserted at the beginning of G. pseudo-r is handled as if it was the r process. Moreover, if pseudo-r was inserted to G, then the real root, r, is ignored when extracted from G. Since it is a single process, we do not need to include it in any data structures and we can handle it as a special case.

5.7 Performance tests

Here we compare our implementations of the Clairvoyant algorithms. Clairvoyant(linear) is the the pseudocode version with the fixes applied. Clairvoyant(pairing heaps) employ pairing heaps from the boost library but also include pairing-heap specific fixes: changing process state in S array removes or inserts the process into the appropriate pairing heap (based on our internal small-scale tests this solution was more performant than filtering during extraction). Finally, the Clairvoyant(segment tree) algorithm is based on segment trees and uses our improvements.

The goal of these tests was to compare these implementations. All implementations produce the same communication schedule and the time spent on communication should be exactly the same. Therefore, they were run on a single node to generate only a communication schedule and do not perform actual communication. As consequence, the results represent the computational overhead for every node as during the reduce operation all nodes run the algorithm in parallel and act according the produced communication schedule.

The tests were performed for two cases: (i) uniform: where all processes have random arrival times, and (ii) skewed: when only one process was delayed. We assume the former considers the situation usually occurring in the real-world compute clusters

Listing 8 Clairvoyant algorithm based on segment tree with all fixes and improvements included. P states are encoded as 1 and E states are represented by 0 in the M matrix.

```
Let $M$ be a state matrix of size $2P \cdot N$. Set $M(*,*) = $ P, i.e. mark all segments as available.
Let $T$ be a segment tree over all $2P$ columns of $M$
Create awaiting queue $A$ by sorting processes by their arrival times
let $G$ = empty list
$P'$ = empty set
while all processes but one have all segments in $E$ state and none in $P'$ do
    for $(i, j) \in P'$ do
        set $M(i, j) = P$                                  ▷ revert states $P'$ to $P$
    end for
    call merge-and-clean($G$, $A$)
    restore all saved copies of columns of $M$
    rebuild $T$
    if size($G$) $\leq 1$ then
        skip rounds until another process becomes available
        continue
    end if
    $P'$ = empty set
    if $r \in G$ and $r$ is not at the head of $G$ then
        insert pseudo-r at the head of $G$
    end if
    for $\forall p \in G$ do
        if $p = r$ and pseudo-r has been processed this round then
            continue
        end if
        $peer, seg$ = find-peer-and-segment($T, G, p$)
        if $peer \neq \phi$ then
            enqueue to $I(i)$ the tuple $(round, peer, seg)$
            enqueue to $O(peer)$ the tuple $(round, p, seg)$
            $M(peer, seg) = E$
            $M(p, seg) = E$
            update $T$ up for segment $seg$ starting from leaf $peer$
            update $T$ up for segment $seg$ starting from leaf $p$
            save a copy of $M(peer, *)$, and reset $M(peer, *) = E$
            update $T$ up starting from leaf $peer$
            append $(p, s)$ to $P'$
        end if
    end for
    for $\forall i \in G$ do
        $a_i = a_i + d$
    end for
end while
```

[6], and the latter reflects the conditions undertaken in the paper [9], where the original Clairvoyant algorithm comes from.

Input instances were randomly generated. In uniform instances arrival times were chosen from the range $[0, P + 0.1]$ with uniform distribution. A random process was selected as the root. In skewed instances, all processes but one arrived on time 0, while the process $P - 1$ arrived at time $a_{P-1} = N$. In skewed instances process 0 was designated as the root. In both cases, the round time was chosen from the range $[0.001, 1]$ with uniform distribution. The results are averaged over 250 executions.

Table 1 Benchmark results showing comparison between different implementations of Clairvoyant algorithm

	Uniform		Skewed	
$P = N$	$\frac{\text{linear}}{\text{segment}}$	$\frac{\text{pairing heaps}}{\text{segment}}$	$\frac{\text{linear}}{\text{segment}}$	$\frac{\text{pairing heaps}}{\text{segment}}$
4	0.49	1.84	0.45	2.45
8	0.62	3.07	0.61	5.68
16	0.73	3.53	0.61	5.34
32	0.97	5.72	0.67	8.60
64	1.26	8.17	0.64	14.37
128	2.00	12.46	0.61	21.04
256	4.05	15.58	0.64	22.77
512	19.33	43.64	1.36	82.98

Results are averaged over 250 runs. Values in table show the average speedup of segment-tree implementation over the linear and pairing-heaps, defined as a ratio of execution time of respectively linear and pairing heap to the execution time of segment tree implementation

The obtained results are presented in Figures 7 and 8 and Table 1. In the case of uniformly distributed arrival times our improvements reduce the run time by almost two orders of magnitude. When only one process is delayed, we achieve a similar speedup over the pairing heaps version.

The ratios in Table 1 describe the relation of average run times of compared algorithms with confidence level of 99.9% for all cases except the uniform, linear, $N = P = 32$, where the confidence level is 98% (according to the Welsh's t-test). The analysis of the speedup values in Table 1 reveal causes of such differences.

The skewed case may be considered as a best or almost-best case scenario, where all processes except one partake in reduction since the first round. In such case the segment tree version takes about 67% more time than the linear version for up to 256 processes. Since this difference is independent of the problem size, we attribute it to the increased code complexity of the segment tree version. For 512 processes the gains from employing instruction level parallelism become apparent. The greater code complexity and much larger memory usage of the pairing heap implementation are apparent since the smallest instance size. The step-like changes in the speedup values suggest that the worse utilization of cache memory is the major culprit of execution time differences.

In the uniform case, not all processes are ready since the beginning of the reduce operation and may require much more work from the algorithm (see e.g. Lemma 1). The step-like speedup changes for the pairing heap algorithm are not as apparent but still are visible confirming the thesis that the memory utilization is the weak point of this implementation. In this case the sublinear time required to find a peer gives the segment tree implementation an increasing advantage over the linear one.

The error bars on the graphs depict the standard deviation. Such large deviation for linear and pairing heaps is a consequence of the "busy waiting" performed by these algorithms when the round time is relatively small. Such a bad case rarely occurred in test instances, therefore not all data points expose this large deviation, but when it occurs, the algorithm's performance drops significantly (from an average of about

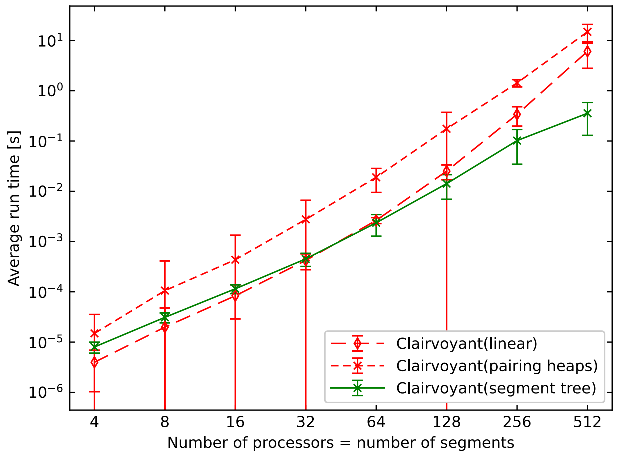

Fig. 7 Benchmark results showing comparison between different implementations of Clairvoyant algorithm. Results are averaged over 250 runs, uniform distribution of process arrival times

15 s per run to 65–90 s with very short round times for the pairing heaps version in the $N = P = 512$ case). This demonstrates that, without skipping idle rounds, the algorithm is much more sensitive to the distribution of process arrival times, where one severely delayed process can slow down the reduce operation not only by its lateness but also by causing unnecessary computations in all nodes.

As a consequence of these results, we have chosen the segment tree implementation as the best one for comparison with other reduce algorithms. While it is somewhat slower in the skewed case than the linear version, it performs better in medium and large instances with uniform arrival times distribution.

The results of all three implementations were cross-checked to ensure that the resulting schedules are the same. The tests were performed on a single Xeon E5-2670 v3 @2.3 GHz CPU under CentOS v. 6.10, with kernel v. 2.6. The code was compiled in 64 bit mode with gcc 7.3.0 and boost 1.70.0, using the following options for optimization: -O3 -mavx -flto. The completions of these tests took about 12 hours.

6 PAP prediction

The proposed algorithm assumed *a priori* knowledge of the PATs by every process in the moment of collective operation call—that is why it was named Clairvoyant. We propose an extension to provide this information using our solution, presented in [15], introducing an additional thread, monitoring computations and possibly additional actions.

The approach is dedicated to the iterative processing model, where the computation and communication phases are repeated consecutively for the whole application execution. During the computation phase, the background thread can use the unloaded network for exchanging the progress information between collaborating processes,

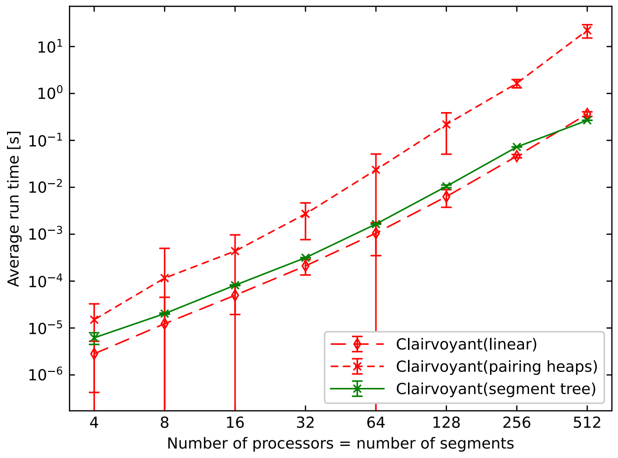

Fig. 8 Benchmark results showing comparison between different implementations of Clairvoyant algorithm. Results are averaged over 250 runs, skewed distribution of process arrival times

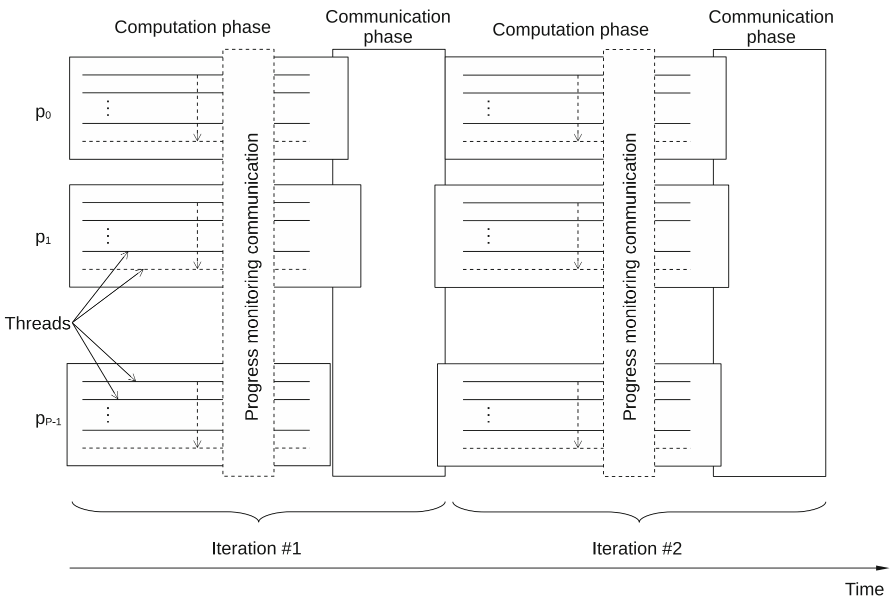

Fig. 9 Iterative processing model (solid lines), extended by the introduced progress monitoring mechanism (dashed lines)

providing PAP details before starting the communication phase, when the collective operation is called, see Fig. 9.

The proposed extension was tested using a specially developed mini-benchmark emulating the imbalanced-PAPs for MPI parallel programs employing an iterative processing model. This approach is similar to the one presented in [15,16], where it was used for all-reduce, scatter and gather operations. In each iteration, the benchmark performs the following steps: (i) random data generation, (ii) double barrier to synchronize all the processes, (iii) emulation of the computation (by `usleep()` function) with randomly generated additional time (unified distribution with a given maximum value), (iv) measurement and storing of the communication times, and (v) validation of the operation results.

Similarly to the commented paper [9], we used a collection of typical reduce algorithms for the comparison purposes:

- Default OpenMPI [11] implementation (MPI), it uses different types of structures (binomial tree or pipeline) with or without the segmentation depending on the data size and the process number,
- A typical binomial tree without segmentation (BNOM),
- An algorithm proposed by Rabenseifner in [19] (RAB),
- Radix-K with group sizes generated by trying to divide processes into groups of one of predefined sizes (RXK) [13],
- Parallel ring, also known as a bucket/cycling algorithm presented in [2] (RING).

Each of the above algorithms was executed using the mini-benchmark in the exactly same way as Clairvoyant (CLV).

There are several measures used for PAP-aware algorithms' evaluation. In the commented paper [9], there was proposed an absorption time reflecting the ability of the algorithm to mitigate the impact of the imbalanced PAP, as well as the (relative) run time which denotes the time required for completing the operation since the first process arrival to the finish of the last one. In this paper we decided to use the latter, however with the exact absolute values, instead of the relative ones normalized to CLV results.

The experiments were performed using supercomputer Tryton [8], placed in Centre of Informatics - Tricity Academic Supercomputer & networK (CI TASK) at Gdansk University of Technology in Poland. It consists of over 1600 compute nodes, interconnected by 56 Gb/s InfiniBand and 1 Gb/s Ethernet networks, with the total compute power about 1.5 PFLOPS. Each node contains 2 $\times$ Xeon E5-2670 v3 @2.3 GHz CPUs, 128 GB of RAM, network interfaces and works under CentOS v. 6.10, with kernel v. 2.6. For the experiment purposes, to keep the network and CPU noise on a lowest possible level, one rack case was separated, with 48 nodes interconnected by Ethernet.

The mini-benchmark was executed multiple times, with different configurations. To avoid the non-unified data transfer times, each node hosted exactly one MPI process, and for the testing purposes we used between 4 and 48 nodes (processes). The data size was consecutively increased from 128 K to 1 M of float numbers (`MPI_FLOAT`), and the PATs were randomly generated for a maximum delay being in 0 to 0.5 s range. Thus, the mini-benchmark was executed $12 \times 5 \times 9 = 540$ times, with 128/256 iterations (measurements) per each repetition. The code was compiled in 64 bit mode with gcc 7.3.0 and boost 1.70.0, using -O3 optimization option.

From the statistical point of view the obtained results seem to be stable. The standard deviation is mostly below 10% of the measurement values, and depends on the data size, with larger values for smaller messages, what is usually observed in a typical network traffic. The graphical representation of the error ranges ($\pm\sigma$ or 68% of the certainty for the normal distribution) is provided as error bars in Figs. 10 and 11.

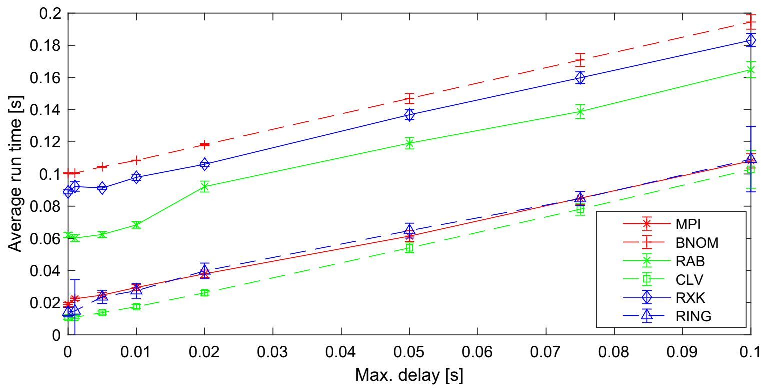

Fig. 10 Benchmark results showing influence of the increasing maximum delay on average run times of the tested algorithms. The experiments were performed on 48 nodes for 1 Gbps Ethernet network, and message size: 512 K of float numbers (`MPI_FLOAT`). The error bars are set to $\pm\sigma$ (68% of the measurements for the normal distribution)

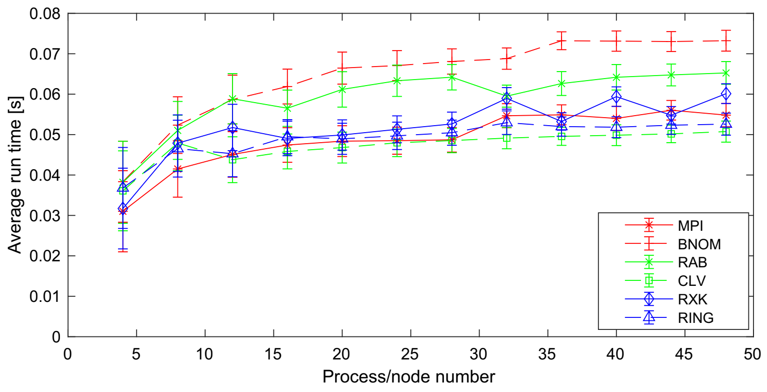

Fig. 11 Benchmark scalability results for 1 GB Ethernet network, processes randomly delayed up to 50 ms, 128 K of float numbers (`MPI_FLOAT`) data size. The error bars are set to $\pm\sigma$ (68% of the measurements for the normal distribution)

Figure 10 presents the results of the experiments, showing the influence of the increasing maximum delay time on the tested algorithms' behavior. We can notice that the Clairvoyant has the shortest run times, slightly outperforming Parallel Ring and native OpenMPI [11] algorithms. The others have much worse results for the proposed configuration.

Figure 11 presents the scalability of the tested algorithms, showing the influence of increasing number of cooperating processes on the run times. We observe that the Clairvoyant is better for a larger cooperating group having advantage over other algorithms, and the overall decrease of its performance is sublinear.

In the provided imbalanced PAP environment the reduce operations based on Clairvoyant algorithm show expected results—for some configurations, especially with the higher number of nodes involved, it outperforms its regular counterparts. Similarly to

[15,16], the background thread seems to provide the proper support for PAP detection, which is successfully used to provide necessary PAT information to the collaborating processes.

7 Final remarks

The paper presented our comments on the Clairvoyant algorithm [9]. They covered a wide range of the changes beginning from the corrections of the pseudocode, necessary for the implementation, through the improvements providing performance increase, such as skipping of idle rounds or use of segment trees instead of the linear search, and finalizing on an extension introducing a possible PAP prediction method to make the algorithm feasible for a real deployment.

We consider the following possible future works related to the imbalanced PAP subject:

– Generalization of the Clairvoyant algorithm for the all-reduce case,
– Designing of new or improved PAP-aware algorithms for other collective operations, e.g. all-gather,
– Introduction of hardware specific network support, e.g. multicast in InfniBand, for PAP-aware algorithm improvements,
– Simulation of the PAP imbalances for exascale (and beyond) supercomputers, including GP GPU support, e.g. with usage of MERPSYS simulator [3].

We believe that the ubiquity of the PAP imbalances in current and most probably future HPC environments [6] will motivate scientists to perform new researches in the field, including improvement of the existing PAP-aware algorithms and design of the new ones as well as their introduction to the modern supercomputing solutions.

References

1. Belcastro L, Marozzo F, Talia D (2019) Programming models and systems for big data analysis. Int J Parallel Emerg Distrib Syst 34(6):632–652
2. Chan E, Heimlich M, Purkayastha A, van de Geijn R (2007) Collective communication: theory, practice, and experience. Concurr Comput Pract Exp 19(13):1749–1783
3. Czarnul P, Kuchta J, Matuszek M, Proficz J, Rościszewski P, Wójcik M, Szymański J (2017) MERPSYS: an environment for simulation of parallel application execution on large scale HPC systems. Simul Model Pract Theory 77:124–140
4. Edelkamp S, Elmasry A, Katajainen J (2017) Optimizing binary heaps. Theory Comput Syst 61(2):606–636

5. Faraj A, Yuan X, Lowenthal D (2006) STAR-MPI: self tuned adaptive routines for MPI collective operations. In: Proceedings of the 20th Annual International Conference on Supercomputing, pp 199–208
6. Faraj A, Patarasuk P, Yuan X (2008) A study of process arrival patterns for MPI collective operations. Int J Parallel Progr 36(6):543–570
7. Hasanov K, Lastovetsky A (2017) Hierarchical redesign of classic MPI reduction algorithms. J Supercomput 73(2):713–725
8. Krawczyk H, Nykiel M, Proficz J (2015) Tryton supercomputer capabilities for analysis of massive data streams. Pol Marit Res 22(3):99–104
9. Marendic P, Lemeire J, Vucinic D, Schelkens P (2016) A novel MPI reduction algorithm resilient to imbalances in process arrival times. J Supercomput 72:1973–2013
10. Marendić P, Lemeire J, Haber T, Vučinić D, Schelkens P (2012) 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). vol 7484 LNCS. Springer, Berlin, pp 439–450
11. Open MPI: Open Source High Performance Computing. https://www.open-mpi.org. Accessed 23 Oct 2020
12. Patarasuk P, Yuan X (2008) Efficient MPI bcast across different process arrival patterns. In: 2008 IEEE International Symposium on Parallel and Distributed Processing, pp 1–11. IEEE, Apr
13. Peterka T, Goodell D, Ross R, Shen HW, Thakur R (2009) A configurable algorithm for parallel image-compositing applications. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09, New York, NY, USA, Association for Computing Machinery
14. Połap D, Woźniak M, Damaševičius R, Maskeliūnas R (2019) Bio-inspired voice evaluation mechanism. Appl Soft Comput 80:342–357
15. Proficz J (2018) Improving all-reduce collective operations for imbalanced process arrival patterns. J Supercomput 74(7):3071–3092
16. Proficz J (2020) Process arrival pattern aware algorithms for acceleration of scatter and gather operations. Cluster Comput
17. Proficz J, Sumionka P, Skomiał J, Semeniuk M, Niedzielewski K, Walczak M (2020) Investigation into MPI all-reduce performance in a distributed cluster with consideration of imbalanced process arrival patterns. In: Barolli L, Amato F, Moscato F, Enokido T, Takizawa M (eds) Advanced information networking and applications. AINA 2020. advances in intelligent systems and computing, vol 1151. Springer, Cham, pp 817–829
18. Qian Y, Afsahi A (2011) Process arrival pattern aware alltoall and allgather on infiniband clusters. Int J Parallel Progr 39(4):473–493
19. Rabenseifner R (2004) Optimization of collective reduction operations. In: Bubak M, van Albada GD, Sloot PMA, Dongarra J (eds) Computational science-ICCS 2004. Springer, Berlin, pp 1–9
20. RDMA Consortium. http://www.rdmaconsortium.org. Accessed 23 Oct 2020
21. Shan H, Williams S, Johnson CW (2018) Improving MPI reduction performance for manycore architectures with OpenMP and data compression. In: 2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), pp 1–11. IEEE, Nov
22. Shi Q, Zou B, Zhang L, Liu D (2019) Hybrid parallel FDTD calculation method based on MPI for electrically large objects. Wirel Commun Mob Comput 2019:1–9
23. Stern J, Xiong Q, Skjellu J, Skjellum A, Herbordt M (2017) Accelerating MPI_Reduce with FPGAs in the network extended abstract. In: Proceedings of the Workshop on Exascale MPI
24. The Standarization Forum for Messsage Passing Interface (MPI). https://www.mpi-forum.org Accessed 23 Oct 2020
25. Wang L, Wang X (2019) A simple and space efficient segment tree implementation. MethodsX 6:500–512
26. Wozniak M, Polap D (2020) Intelligent home systems for ubiquitous user support by using neural networks and rule-based approach. IEEE Trans Ind Inform 16(4):2651–2658
27. Xiong Q, Yang C, Haghi P, Skjellum A, Herbordt M (2020) Accelerating MPI collectives with FPGAs in the network and novel communicator support. In: 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), p 215. IEEE, May
28. Zhao T, Wang Y, Wang X (2020) Optimized reduce communication performance with the tree topology. In: Proceedings of the 2020 4th High Performance Computing and Cluster Technologies Conference

and 2020 3rd International Conference on Big Data and Artificial Intelligence, pp 165–171, New York, NY, USA, Jul ACM