MPI Collective Algorithm Selection in the Presence of Process Arrival Patterns

Majid Salimi Beni
Department of Computer Science
University of Salerno
Salerno, Italy
msalimibeni@unisa.it

Biagio Cosenza
Department of Computer Science
University of Salerno
Salerno, Italy
bcosenza@unisa.it

Sascha Hunold
Faculty of Informatics
TU Wien
Vienna, Austria
hunold@par.tuwien.ac.at

***Abstract*—The Message Passing Interface (MPI) is a programming model for developing high-performance applications on large-scale machines. A key component of MPI is its collective communication operations. While the MPI standard defines the semantics of these operations, it leaves the algorithmic implementation to the MPI libraries. Each MPI library contains various algorithms for each collective, and selecting the best algorithm typically relies on performance metrics obtained from micro-benchmarks. In such micro-benchmarks, processes are typically synchronized using an MPI_Barrier before invoking a collective operation. However, in real-world scenarios, processes often arrive at a collective in diverse patterns, often due to resource contention. The performance of collective algorithms can vary significantly depending on the arrival pattern type.**

In this work, we address the challenge of selecting the most efficient algorithm for a given collective, taking into account process arrival patterns. First, we demonstrate through a simulation study that arrival patterns significantly influence the choice of the optimal collective algorithm for specific communication instances. Second, we conduct a comprehensive micro-benchmark analysis to illustrate the sensitivity of MPI collectives to these arrival patterns. Third, we show that our innovative micro-benchmarking methodology is effective in selecting the best-performing collective algorithm for real-world applications.

***Index Terms*—Message Passing Interface (MPI), Algorithm Selection, Library Tuning, Process Arrival Patterns, Collective Communication Operations**

I. Introduction

The Message Passing Interface (MPI) and its collective operations play a pivotal role in the domain of high performance computing. MPI provides an efficient way to exchange data between processes distributed across a network in an HPC cluster. MPI collectives encompass essential communication patterns such as one-to-all (broadcast), all-to-one (reduce), all-to-all, scatter, and gather.

The MPI standard defines the semantics of collective operations, but leaves their algorithmic implementations to MPI libraries. Hence, MPI libraries provide several algorithms for each collective operation, and a decision logic selects one of these algorithms. Each algorithm has its distinct characteristics in terms of message size, network usage, and scalability. Based on the scenario, one algorithm may outperform the others for each collective operation, and therefore, selecting the right algorithm highly improves the overall scalability, communication efficiency, or resource utilization of a parallel application.

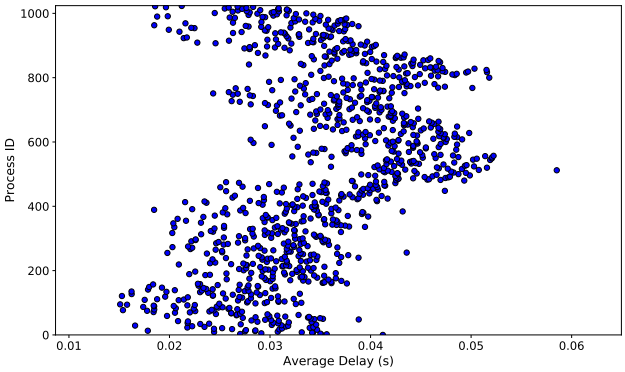


Fig. 1: Avg. process delay (skew) across all `MPI_Alltoall` calls in FT on *Galileo100* with 32×32 processes.

For that reason, a large body of related work exists, which explores different ways of performing the algorithm selection process, considering parameters such as the message size, the number of processes, the network topology, or the cluster utilization [1], [2], [3], [4], [5], [6].

In many MPI applications, processes typically do not enter collective operations simultaneously due to system noise or performance variability of HPC systems [7], [8], imbalanced workloads [9], [10], or deficiencies of the current synchronization methods [11].

Figure 1 demonstrates the actual process arrival patterns as observed for the FT application of the NAS Parallel Benchmarks [12]. To generate this plot, we recorded the timestamp of each process upon entering a collective, specifically `MPI_Alltoall`. Each collective call is assigned a sequence number, enabling us to calculate the delay of each process relative to the first process to enter a collective. We then calculate the average delay across all sequences. We can observe that the average delay is not uniformly distributed, indicating significant optimization potential.

The MPI process arrival imbalance is also shown to be impactful when designing a new collective algorithm or selecting an existing algorithm for MPI collective operations [13], [14]. Thus, a well-performing collective algorithm under a balanced process arrival pattern may show poor performance under an imbalanced process arrival pattern. Detecting actual

arrival patterns, however, is time-consuming and sometimes infeasible since the arrival times may vary depending on the cluster, network, number of nodes, network contention, or application [14], [15]. Therefore, selecting robust algorithms for MPI collectives, capable of accommodating various arrival time imbalances and ensuring high performance across different scenarios, is crucial.

In this paper, we show how arrival patterns can impact the algorithm selection for collective operations, first in a simulated environment using the SimGrid toolkit [16] and second on a real parallel machine using micro-benchmarks. We then present our novel strategy for selecting collective algorithms based on their robustness against different arrival patterns and show how this strategy can reduce the runtime of parallel applications, such as FT from the NAS Parallel Benchmarks [12].

Overall, we make the following contributions:

- We present a comprehensive simulation study demonstrating that several MPI collective operations, particularly rooted collectives like `MPI_Reduce`, are sensitive to process-arrival patterns.
- We propose an innovative algorithm selection technique that evaluates the performance of different collective algorithms under various arrival patterns to identify the most robust algorithm for specific message sizes and process counts.
- We introduce a specialized MPI tracing tool designed to accurately capture process arrival patterns in MPI applications.
- Through a case study using the FT application, we show that our algorithm selection technique, informed by benchmarking under different arrival patterns, can enhance application runtime across three different machines.

The rest of the paper is organized as follows: In Section II, we state our working hypothesis and introduce the notation used throughout the paper. Section III details our simulation study to support our hypothesis and guide our real-world experiments. Section IV presents our experimental findings for real-world benchmark experiments, which align with the simulation study. In Section V, we demonstrate that micro-benchmarking using process-arrival patterns can indeed enhance application performance. We summarize related approaches in Section VI and conclude the paper in Section VII.

II. Background and Notation

Here, we define our target metrics and provide a brief overview of the scientific background.

A. Process Arrival Patterns

In the present work, we examine the performance dependency of MPI collective operations on process arrival patterns. The process arrival time is the time at which a process enters a collective MPI operation. An arrival pattern emerges from the difference in process arrival times between participating processes. We consider a system with p MPI processes $P_0, P_1, ..., P_{p-1}$. Each process P_i has its own arrival time a_i and exit (or finish) time e_i. Figure 2 shows an example of imbalanced process arrival times when executing MPI programs, which illustrates that processes may arrive in a collective at different times also finish at different times. In case all processes enter a collective call simultaneously, only the last process to exit defines the running time of the operation. This is, in general, what a typical MPI micro-benchmark, such as the OSU Micro-Benchmarks [17] or ReproMPI [18], is measuring. The total delay d^* denotes the time difference between the first process arriving and the last process exiting a collective call, i.e.,

$$d^* = \max_{0 \le i < p}(e_i) - \min_{0 \le i < p}(a_i) \quad . \tag{1}$$

On the contrary, the total delay is less informative if arrival patterns are present. For example, if a process was severely delayed, the total delay would include the waiting time induced by the process imbalance. Since the arrival pattern is an external factor that we can hardly control, our primary goal should be to minimize the time between the last process entering and the last process exiting a collective call. The last delay $\hat{d}$ is defined accordingly, i.e.,

$$\hat{d} = \max_{0 \le i < p}(e_i) - \max_{0 \le i < p}(a_i) \quad . \tag{2}$$

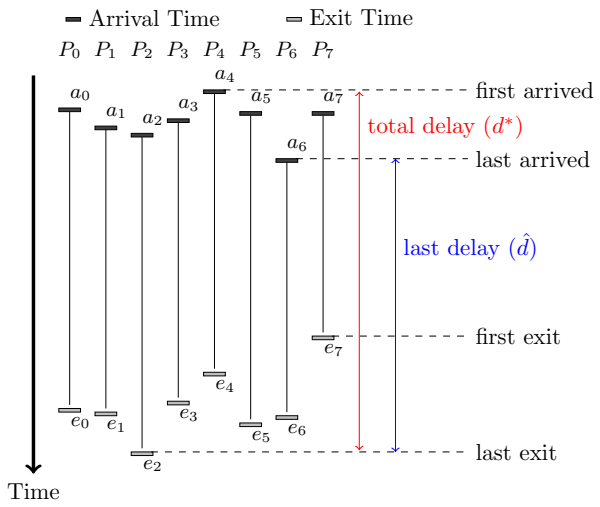


Fig. 2: Example of process arrival pattern with 8 processes.

B. Clock Precision and Clock Synchronization

In a typical parallel compute cluster, each compute node has its own local clock, and these clocks are periodically synchronized using some protocols to compensate for drifting clocks. However, the accuracy of these clocks is often too low for fine-grained measurements [19]. For inspecting the impact of process arrival patterns on different collective algorithms, a precise clock synchronization mechanism, called HCA3 [20], is used in this paper to obtain a precise, logical global clock. HCA3 can be used to obtain a precise logical, global clock by synchronizing the clocks of MPI processes in a logarithmic number of rounds. The global clock's accuracy is less than one microsecond [11], making it adequate for our purposes.

C. SimGrid and SMPI

SimGrid [16] is a versatile framework that facilitates the simulation of HPC applications in virtual environments, enabling detailed analysis and performance modeling of distributed systems. Its SMPI module [21] can simulate actual MPI applications on virtual, simulated platforms. It accomplishes this by intercepting communication calls and emulating their actions while allowing the computations to execute as they would on real systems. SMPI provides tracing features, allowing users to monitor MPI collectives, to observe their behavior and performance under different circumstances. It also enables precise comparison of different algorithms for collective operations, providing insights into their efficiency and behavior on any arbitrary network configuration.

III. Assessing the Optimization Potential of Collectives via Simulation

To support our hypothesis that a specific process arrival pattern impacts the performance of MPI collectives, we conducted a simulation study using SimGrid. There are two main benefits of using a simulator for such a study. First, the simulations are free of system noise, and runtimes are accurately reproducible. Second, all clocks in the simulator are synchronized by design. Therefore, we do not need to employ any clock synchronization algorithm for the simulations.

It is important to emphasize that the simulation results are not meant to be a one-to-one representation of real-world performance. Instead, they provide insights into the optimization potential of MPI collectives when exposed to different process arrival patterns.

A. Simulation Environment and Parameters

The SMPI module of SimGrid provides a large variety of implementations of MPI collectives. For example, SMPI contains implementations of some collective algorithms found in Open MPI [22] or MVAPICH [23]. We obtained our simulation results with SimGrid 3.35.

To demonstrate the performance of different algorithms for collective operations under various process arrival patterns, we utilized a simulation platform representing a cluster with 32 nodes. Each node comprises 32 cores and is connected to a switch, forming a typical two-level hierarchical cluster. The intra-node network has a bandwidth of 10 Gbps and a latency of 1 µs. The inter-node network also has a bandwidth of 10 Gbps but a latency of 2 µs, approximately reflecting the values of our local Omni-Path-based cluster.

We experimented with several other platforms featuring different bandwidths and latencies. However, the choice of bandwidth or latency did not significantly impact the overall outcomes. The fastest algorithm in one setting generally remained the fastest even when latency and bandwidth were varied. Additionally, we simulated several rooted and non-rooted collectives, anticipating that rooted algorithms would exhibit greater sensitivity to arrival patterns compared to non-rooted collectives. For the sake of conciseness, we only present results for one rooted (`MPI_Reduce`) and two non-rooted (`MPI_Allreduce`, `MPI_Alltoall`) collectives.

Listing 1: Applying an artificial process arrival pattern.

```
for (i=0; i<NREP; i++) {
#ifdef SIMULATOR
    double wait_time = get_arrival_pattern_delay();
    usleep(wait_time);
#else
    MPIX_Harmonize();
    double skew_time = MPI_Wtime() +
     get_arrival_pattern_delay();
    while( MPI_Wtime() < skew_time );
#endif
    double start_time = MPI_Wtime();
    MPI_COLLECTIVE(...);
    double end_time = MPI_Wtime();
}
```

It is important to state how a specific process-arrival pattern can be established. Several other works relied on `MPI_Barrier` to synchronize processes and then waited for a specific amount of time. However, synchronizing processes using an `MPI_Barrier` may be very inaccurate for small message sizes [11]. Therefore, we synchronize the processes also in time by calling `MPIX_Harmonize` [11] and then wait for a given amount of time. This method can accurately replay certain process arrival patterns in a micro-benchmark scenario. Listing 1 illustrates how an arrival pattern is established and how the runtime of a collective call is measured.

Since all processes in the simulation already share the same global clock, there is no need to synchronize using `MPIX_Harmonize`. Instead, we can just wait until the exact target time is reached before entering the collective call.

B. Artificial Process Arrival Patterns

To generate realistic process arrival patterns, we have recorded a large number of program traces that capture the start and end times of collectives (cf. Section V-A). We have observed that identifying typical arrival patterns is challenging because each application produces distinct patterns based on the parallel machine, number of processes, or input size. Therefore, we created eight artificial process arrival patterns that encapsulate the general trends observed from our tracing experiments. The various configurations of these patterns are depicted in Figure 3. The magnitude of the *maximum process skew*, displayed on the y-axis, consistently varies. For short-running collectives, the maximum process skew will be smaller compared to long-running collectives.

To generate a concrete process arrival pattern, we select one of the shapes from Figure 3 and define a *maximum process skew* (s) for this pattern, which defines the time between the first and the last process exiting the pattern. Therefore, the delay experienced by each process due to a specific pattern ranges from 0 to s time units.

In the simulations and in the experiments shown later, all process arrival patterns are generated before starting a micro-benchmark. The generator takes the shape type, the number of processes, and the maximum process skew as inputs and produces a file with p lines, where each line i

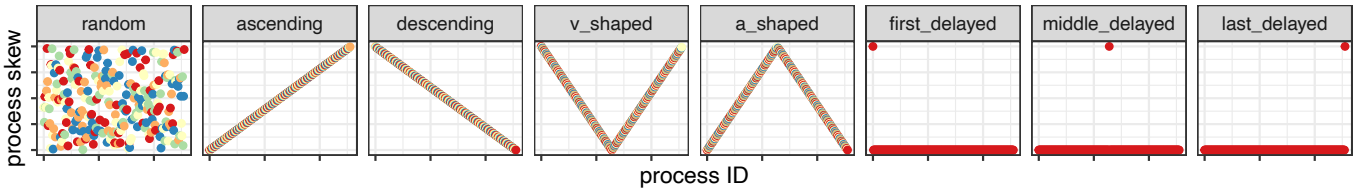


Fig. 3: Visual representation of the shapes of various artificially created process arrival patterns.

denotes the process skew of process P_i. In Listing 1, function `get_arrival_pattern_delay()` returns the process skew for process P_i.

For the simulations, we chose a specific maximum process skew to create diverse process arrival patterns that affect the collective call's outcome. If the process skew in the generated pattern is too small, there will hardly be a difference in the runtime of various collective algorithms. In contrast, if the process skew is too large, the runtime of a collective has little impact on the overall execution time. For that reason, we generated the arrival patterns with the following maximum process skew s as follows: We ran all k algorithms of a specific collective in the simulator and recorded their execution times t_i when there is no process skew, i.e., processes enter the collective simultaneously. We compute the average runtime over all algorithms for this collective call, which we call $t^a = \frac{1}{k}\sum_{i=0}^{k} t_i$. We then multiplied this average runtime t^a with three factors: $0.5, 1.0$, and 1.5. Thus, we generated patterns with three different maximum process skews for each case, which are $0.5t^a, 1.0t^a$, and $1.5t^a$. These three different skews help us evaluate the impact of increasing or decreasing this artificial process skew. In the present paper, we show only results for the $1.5t^a$ factor, as it had the strongest influence on the results.

C. Simulation Results

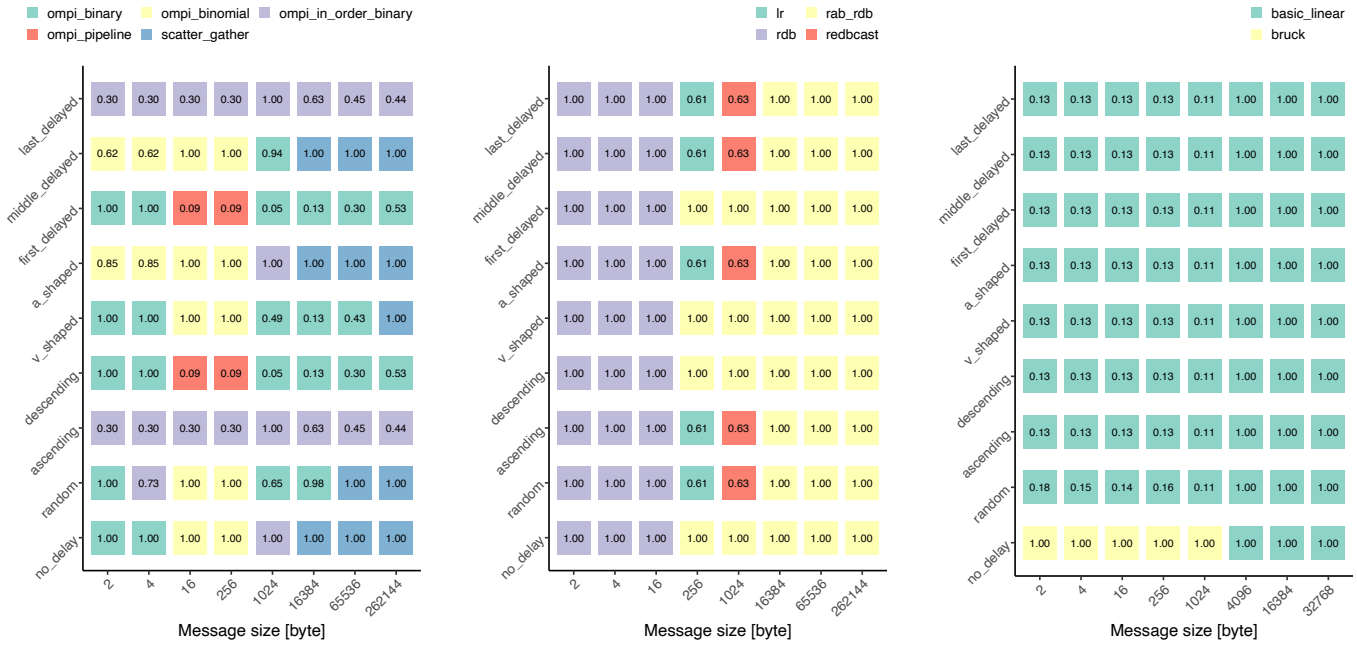
Figure 4 presents our simulation results for three collectives `MPI_Reduce`, `MPI_Allreduce`, and `MPI_Alltoall`. We selected these collectives for presentation because they best illustrate the optimization potential. The relative runtimes displayed in each cell are based on the last delay metric ($\hat{d}$).

We begin with the results of `MPI_Reduce`, displayed in Figure 4a. The last row presents the relative performance results for the `no_delay` case, corresponding to scenarios where all processes are perfectly synchronized. The color indicates the best algorithm found for a specific message size. For example, `ompi_binomial` is the fastest for message sizes from 2 B to 256 B, while `scatter_gather` works best for larger message sizes. We now applied the eight different process arrival patterns from Figure 3. Similar to the `no_delay` case, we plot the best algorithm found for each pattern and message size using the respective color. Let us examine the row for `last_delayed`, where `ompi_in_order_binary` is found to be the best. The value inside each box denotes the relative performance of this algorithm compared to the best algorithm from the `no_delay` case. For example, for a message size of 2 B, we know that a decision logic based on the `no_delay` case would select `ompi_binomial`. However, our experiments show that `ompi_in_order_binary` leads to a smaller value of $\hat{d}$. Thus, the cell shows the relative performance $\frac{\hat{d}_{\texttt{ompi\_in\_order\_binary}}}{\hat{d}_{\texttt{ompi\_binomial}}}$, which for 2 B tells us that the `ompi_in_order_binary` only requires about 30 % of the time of `ompi_binomial`. This simulation outcome aligns with our expectations, as a delay in the last process directly results in an immediate delay in the root process during the first communication round of a binomial tree. This characteristic makes the binomial tree algorithm particularly sensitive to process skew. Generally, it is evident that the optimal algorithm for `MPI_Reduce` varies with different message sizes and process arrival patterns, demonstrating the strong optimization potential of `MPI_Reduce` in real-world scenarios.

Figure 4b presents the simulation results for `MPI_Allreduce`, which is one of the most commonly used collective operations [24]. We have tested the following algorithms available in SimGrid: `lr` (logical ring reduce-scatter followed by logical ring allgather.), `rdb` (recursive doubling), `rab_rdb` (Rabenseifner's algorithm using recursive doubling), `ompi_ring_segmented` (ring algorithm), and `redbcast` (reduce+bcast). When these algorithms are exposed to different process arrival patterns, the best algorithm is often the same as in the `no_delay` case. This aligns with our intuition, as the reduction step in an Allreduce is a strongly synchronizing sub-task, i.e., all processes must receive all buffers from every other process before the result can be propagated back. However, process skew can be absorbed during the reduction step if the reduction algorithm can leverage the skew. This is evident for certain process-arrival patterns (e.g., `ascending` or `last_delayed`) with medium message sizes such as 256 B or 1024 B.

Finally, Figure 4c shows the simulation results for `MPI_Alltoall`. Here, the `bruck` algorithm consistently outperforms the other algorithms in the `no_delay` case for message sizes from 2 B to 1024 B. However, if `bruck` is exposed to different process arrival patterns, the best algorithm changes. For example, the `last_delayed` pattern favors the `basic_linear` algorithm with small message sizes, e.g., 2 B or 4 B. This indicates that the `bruck` algorithm is sensitive to process-arrival patterns, and there is optimization potential for `MPI_Alltoall` in real-world scenarios.

In summary, our simulations reveal significant optimization



(a) `MPI_Reduce` (b) `MPI_Allreduce` (c) `MPI_Alltoall`

Fig. 4: Simulation results of the best algorithm under various process-arrival patterns for different collectives, tested with 1024 processes on a standard cluster system consisting of 32 nodes, each equipped with 32 cores.

potential for most collective operations, particularly rooted ones like `MPI_Bcast` and `MPI_Reduce`, and even some non-rooted collectives such as `MPI_Alltoall`. However, for `MPI_Allreduce`, the impact of process-arrival patterns in real-world scenarios suggests minimal potential for improving the current algorithms in MPI libraries.

IV. Assessing the Robustness of Collectives to Arrival Patterns

We now present the results of our experimental study, which assesses the robustness and performance of collective algorithms under varying process arrival patterns, using the same micro-benchmarking approach as our simulation study.

A. Hardware and Software Setup

Our experiments were conducted on three distinct parallel machines, as detailed in Table I. We used the ReproMPI benchmark suite [18] and HCA3 for clock synchronization in our micro-benchmark experiments. The artificial process arrival patterns employed are those shown in Figure 3. To reproducibly create a specific process arrival pattern in the micro-benchmark, we followed the method outlined in Listing 1. Specifically, we first synchronized the processes using `MPIX_Harmonize` [11], and then each process waited until its designated skew time.

Table II shows the mapping of existing Open MPI collective algorithms to their IDs in our experiments. We omitted some algorithms that only function with two processes or that consistently underperformed across various message sizes.

B. Impact of Process Arrival Patterns on Choice of Collective Algorithm

In the first set of experiments, we assess whether the simulation findings hold true in practice. Specifically, we evaluate whether the optimal algorithm for a given message size under ideal synchronization remains the best under specific process arrival patterns.

Figure 5 shows our experimental results with different process arrival patterns on *Hydra*. For all real-world experiments, we only present a subset of the original eight process arrival patterns and message sizes for the sake of clarity. We selected the most distinct process arrival patterns for the figures. The values in each cell represent the last delay runtime ($\hat{d}$) of each algorithm for a specific arrival pattern.

Figure 5a presents the experimental results for `MPI_Reduce`. Each row contains the runtime (in ms) for one specific process arrival pattern. Consider, for example, the top left plot, which shows the runtime results for 8 B. The maximum process skew s for these experiments is determined as done in the simulation study. We first measured the runtime of all algorithms for the `No-delay` case. Then, we computed the average runtime across all algorithms (for a specific message size) and used this value as input for the process skew generator. The colored boxes separate the algorithms into good (light blue) and less efficient (light red) algorithms. A good algorithm is either the fastest or one that is at most 5% slower than the fastest algorithm (which we consider indistinguishable). Since experimental results may vary a little, it is generally not useful to seek only a single

TABLE I: Characteristics of parallel machines used in the experiments.

Machine	#Nodes	Interconnect	CPU/Cores per Node	MPI Version
Hydra	36 (Dual-Socket Dell PowerEdge)	Intel Omnipath (100 Gbit/s)	2 x 16-core Intel Xeon Gold 6130F	Open MPI 4.1.5
Galileo100	554 (Dual-Socket Dell PowerEdge)	Mellanox Infiniband HDR100	2 x 24-core Intel CascadeLake 8260	Open MPI 4.1.1
Discoverer	1128 (Atos BullSequana XH2000)	Infiniband HDR (Dragonfly+)	2 x 64-core AMD Epyc 7H12	Open MPI 4.1.4

TABLE II: Algorithm IDs and their names in Open MPI 4.1.X.

Collective	Algorithm IDs, Names and Abbreviations
Allreduce	2 Non-overlapping (Non-ovlp), 3 Recursive Doubling (Rec-Dbl), 4 Ring (Ring), 5 Segmented Ring (Seg-Ring), 6 Rabenseifner (Raben)
Alltoall	1 Linear (Lin), 2 Pairwise (Pair), 3 Modified Bruck (M-Bruck), 4 Linear with Sync (L-Sync)
Reduce	1 Linear (Lin), 2 Chain (Chain), 3 Pipeline (Pipe), 4 Binary (Bin), 5 Binomial (Binom), 6 In-order Binary (In-Bin), 7 Rabenseifner (Raben)

very best algorithm. In Figure 5a (left figure), we see that Algorithm 5 (binomial tree) is found to be the best algorithm in the `No-delay` case and 8 B with on average a runtime of 0.01 ms. For the same message size, we can also observe that in the `Last-delayed` case, Algorithm 6 (in-order binary) is the fastest algorithm, and unlike the binomial tree algorithm, it can absorb some of the process skews. The experimental results match our simulation results and our expectations.

For `MPI_Allreduce` in Figure 5b, however, a different behavior can be observed, which also matches our simulation results. Here, we see that the fastest algorithm in the `No-delay` case consistently remains the fastest in most of the other cases. We can also observe that several algorithms perform equally well for the message size of 8 B, which explains the more frequent occurrence of blue rectangles. Additionally, the runtimes of each algorithm across various patterns are more consistent than those for `MPI_Reduce`, showing little variation. Thus, similar to simulation results, `MPI_Allreduce` is less sensitive to arrival patterns compared to `MPI_Reduce`, demonstrating that most available algorithms are robust against process imbalances.

For `MPI_Alltoall`, the process arrival patterns have less impact on the algorithms for a small message size (8 B), as shown in Figure 5c, and the runtime of all algorithms remains relatively stable across various patterns. For a message size of 1024 B, however, there is more variance across different arrival patterns, and the runtimes of the algorithms considerably change by changing the arrival patterns. Here, the maximum process skew is relatively high, as Algorithms 1 and 4 are much slower than the other two in the `No-delay` case, which translates into a larger process skew in the experiments. In the Alltoall experiments, algorithms are very sensitive to the shape of the process arrival pattern. For example, Algorithm 4 (Linear with Sync) is the fastest for three delay scenarios but is very slow for the `First-Delayed` pattern. For `MPI_Alltoall` and a message size of 1 048 576 B, a very similar behavior is observed, where the runtimes of the algorithms significantly vary based on the type of pattern. Another observation is that choosing the fastest algorithm when processes are synchronized (i.e., `No-delay` pattern) might be totally misleading. For this message size, the fastest algorithm for the `No-delay` scenario (Algorithm 2) is, in fact, the worst choice for all other process arrival patterns.

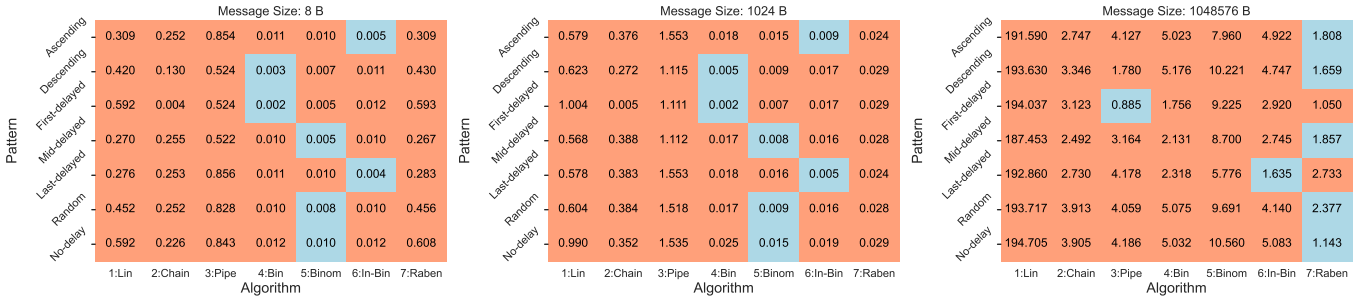
As demonstrated by previous experiments, selecting algorithms for MPI collectives is a challenging task for several reasons. First, the optimal algorithm without process arrival imbalances may not be the best under specific arrival patterns; selection depends on the pattern encountered in the application. This necessitates broadening our selection strategy beyond just choosing the fastest algorithm based on synchronized micro-benchmarks. Second, predicting actual process arrival patterns for a specific application is challenging due to factors like the application type, number of nodes, network and topology, and network congestion. Therefore, assessing the *robustness* of collective algorithms against various arrival patterns, alongside their runtimes, is crucial to ensure overall performance.

C. Assessing the Robustness of Collective Algorithms

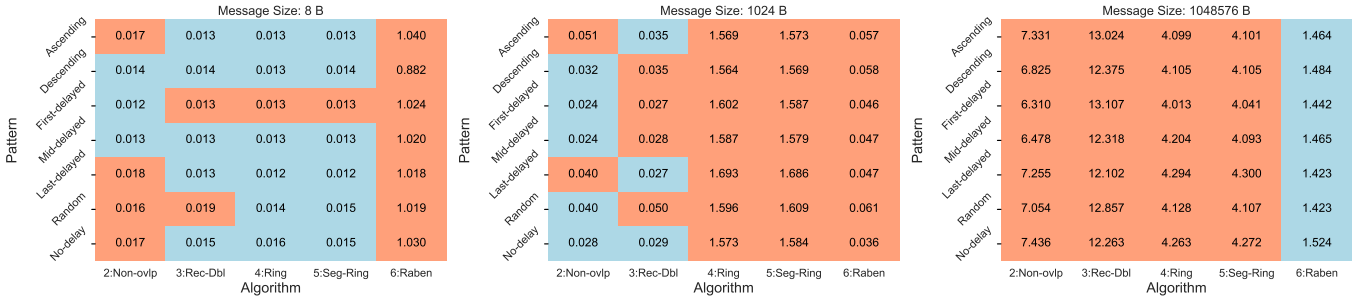
Until now, we have demonstrated that choosing an efficient algorithm for a specific collective heavily depends on the arrival pattern type. Next, we aim to evaluate the robustness of each algorithm against particular patterns, diverging slightly from our previous analyses. Our focus is on identifying algorithm classes that maintain high performance and efficiency despite variations in process arrival patterns, defining robustness as the ability to consistently deliver strong results under these conditions.

In the previous section, we analyzed the runtime of different algorithms for various process arrival patterns, and each algorithm was exposed to the same magnitude for the process skew. In the following robustness experiments, we create a specific process arrival pattern for an algorithm as follows. We obtain the running time t_i for each algorithm i in the `No-delay` case. For each algorithm i, we generate a specific process arrival pattern based on t_i, i.e., the shape is the same but the magnitude changes. The idea is that an algorithm that requires X ms should be given a process arrival pattern with a maximum skew of X ms, while an algorithm that needs $10X$ ms should also be given a maximum skew of $10X$ ms.

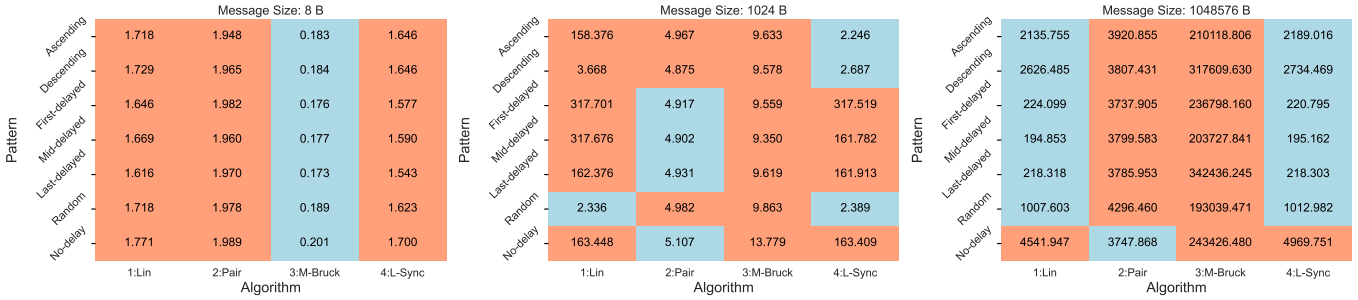
The robustness results are given in Figure 6, which were obtained on *Hydra* with 32×32 processes. We measure the last delay $\hat{d}$ and present normalized values for each arrival pattern. The normalized performance is defined as $\frac{\hat{d}^k}{\hat{d}^{\texttt{No-delay}}} - 1$, which



(a) Reduce



(b) Allreduce



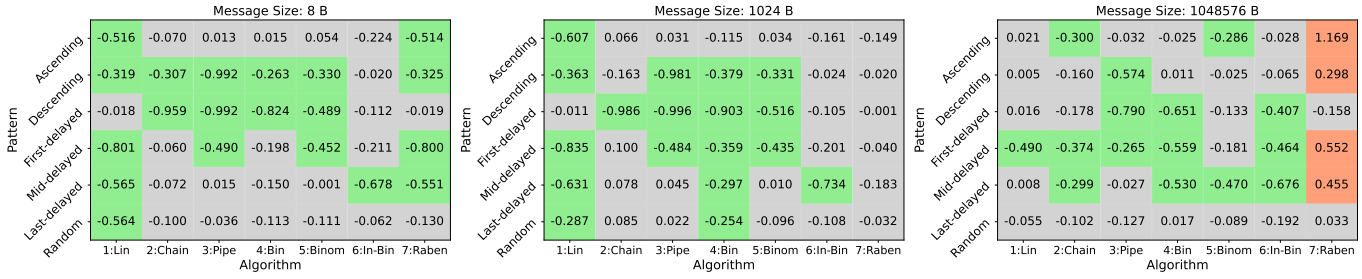
(c) Alltoall

Fig. 5: Impact of arrival patterns on run-times of collective algorithms. Runtimes (in milliseconds) of MPI collectives for various message sizes on *Hydra* and with 32×32 processes. For each arrival pattern, algorithms within 5% of the fastest (per row) are highlighted in light blue, while all others are in light red.

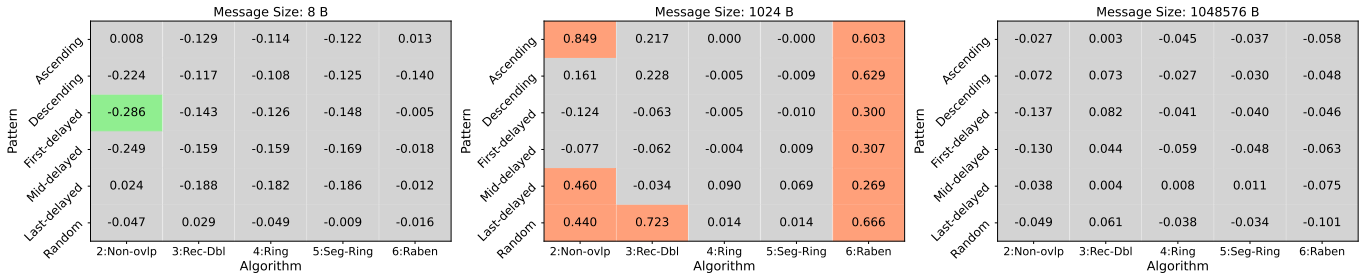
denotes the speedup or slowdown of a collective algorithm under pattern k compared to the `No-delay` case.

Consider the plot for 8 B in Figure 6a. Negative values indicate that the last delay metric for a pattern was smaller than in the `No-delay` case, whereas positive values mean the last delay metric was larger than in the `No-delay` scenario. Values that are within 25 % (either negative or positive) are colored gray to signify that there is no significant impact on this algorithm when being exposed to a process arrival pattern. However, if values are smaller than -0.25, we use green boxes to denote that the algorithm could significantly absorb process skew for a particular arrival pattern. Red boxes represent cases where algorithms get significantly slower (more than 25 %) in the presence of an arrival pattern. For example, we see in Figure 6a that Algorithm 1 makes the biggest improvement (-0.564) in the `Random` case for 8 B, hence it is colored green.

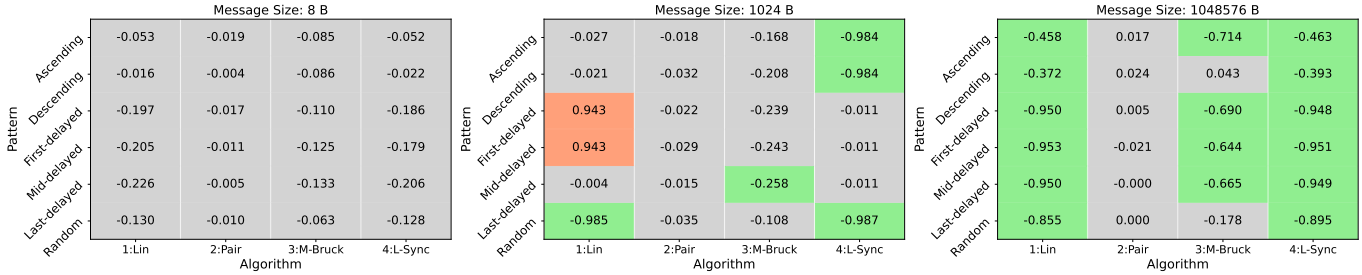
Overall, Figure 6 confirms our previous findings. For `MPI_Reduce`, most algorithms are sensitive to process arrival patterns. However, if the pattern changes, it often improves the performance rather than degrading it, as indicated by the majority of green boxes. Therefore, most `MPI_Reduce` algorithms are robust. For `MPI_Allreduce`, with 8 B and 1 MiB messages, most of the algorithms are less sensitive to different arrival patterns. In contrast, with 1024 B, Algorithms 2 and 6 show poor robustness due to severe slowdowns (red boxes). For `MPI_Alltoall`, a potential for optimization is found for medium and large message sizes. Therefore, in real-world applications dealing with such message sizes, tuning `MPI_Alltoall` in the presence of process arrival patterns should improve the overall application performance. In the next section, we will demonstrate that this hypothesis holds true in practice.



(a) Reduce



(b) Allreduce



(c) Alltoall

Fig. 6: Robustness of collective algorithms against arrival patterns. Normalized runtimes of `MPI_Reduce`, `MPI_Allreduce`, and `MPI_Alltoall` (w.r.t the `No-delay` case) on *Hydra* and 32×32 processes. Green rectangles: at least 25 % faster than `No-delay`; red rectangles: at least 25 % slower than `No-delay`.

V. Improving Application Performance by Process Arrival Analysis

We demonstrate how to enhance the performance of the FT application (problem size D) from NAS Parallel Benchmarks v3.4.2 [12] by selecting the best collective algorithm using our micro-benchmarking technique. This approach involves analyzing each algorithm's performance across various process arrival patterns.

To understand arrival patterns on real production machines, which may not match artificially generated ones, we trace and analyze collective calls within MPI applications. This approach allows us to develop arrival patterns based on real-world scenarios and evaluate the robustness of algorithms against these patterns.

A. Recording Arrival Patterns by Application Tracing

To study real-world arrival patterns, we developed a small MPI tracing library using the PMPI interface of MPI. In contrast to tracing libraries like Score-P [25], our library only focuses on MPI collectives. More importantly, it synchronizes the clocks before starting the tracer, which enables us to measure process arrival patterns more accurately. The library includes features for process and collective call sampling, allowing only a subset of processes to be traced or every kth collective call to be recorded. This approach is often sufficient to gain an overview and helps reduce the size of the trace.

We traced the `MPI_Alltoall` calls in the FT application, where Alltoall is the primary communication pattern, consuming 50–70% of the total runtime, depending on the collective algorithm used. In this application, `MPI_Alltoall` is the dominant collective operation, accounting for over 95% of the MPI operations' time with a message size of 32 768 B.

For each `MPI_Alltoall` call in the FT application, we set the arrival time of the first process as time zero and subtract the arrival times of all other processes from this value. We apply this method to all `MPI_Alltoall` calls in FT, ultimately calculating the average delay for each process across all calls. This average skew per process over all calls in

each cluster is termed the FT-Scenario. Figure 1 illustrates the average delay pattern of all MPI_Alltoall calls in FT for all 1024 processes on *Galileo100*, clearly indicating that some processes experience more delay than others.

B. Replaying Arrival Patterns from Applications in Micro-benchmarks

We demonstrate how the process arrival pattern in an application can affect the performance of collective algorithms. To this end, we conduct an Alltoall micro-benchmark using the same message size as the Alltoall calls in FT. Note that the micro-benchmark uses the No-delay scenario, where all processes enter the MPI_Alltoall function at the same time (i.e., no arrival pattern is applied). Figure 7 shows the runtime of FT and Alltoall on *Hydra*, *Galileo100*, and *Discoverer*. We ran FT 10 times and reported the average runtime. On *Hydra* (Figure 7a), when running the Alltoall micro-benchmark (the bottom figure), Algorithms 1, 2, and 4 are much faster than Algorithm 3, and for instance, Algorithm 4 is about four times faster than Algorithm 3. However, when using the same Alltoall algorithms in FT (the figure above), we see a totally different picture. Unlike the micro-benchmark, choosing Algorithms 1 and 4 would not be the best choice for FT. In fact, the behavior of algorithms varies between running in the micro-benchmark and within the application.

Likewise, on *Galileo100*, Algorithm 2 shows the best performance in a micro-benchmark, where it is 27% faster than Algorithm 4. However, Algorithm 2 is not the best choice for the FT application, where FT's runtime becomes 8% slower than choosing Algorithm 4. On *Discoverer*, however, the best algorithms in the micro-benchmark and application are the same, but the ratios of runtimes of different algorithms are still different. For example, in the Alltoall micro-benchmark, there is a 40% difference between Algorithms 1 and 2, while this difference is only 8% in the FT runtime.

Overall, the runtime trend of the fastest Alltoall algorithms is inconsistent across different machines due to changing runtime ratios of the algorithms. Consequently, the selection logic for MPI collective algorithms should not rely solely on micro-benchmarking with time-synchronized processes.

To understand why MPI_Alltoall algorithms perform differently in a micro-benchmark without process arrival imbalance compared to their performance within an application, we replicate previous experiments. We subject the MPI_Alltoall algorithms to various process arrival patterns, including both artificial ones and those derived from tracing FT in each cluster (FT-Scenario). The maximum process skew used to generate the artificial patterns is determined by the highest skew observed during tracing on each parallel machine.

Figure 8 present the runtimes of the Alltoall algorithms when being exposed to different process arrival patterns on the three different machines. The values in the heatmaps are normalized to the smallest value in each row, meaning that the fastest algorithm for that delay scenario has a normalized value of 1. In parentheses, the absolute runtimes of the algorithms based on the last delay metric are shown. In the last row, we provide the average performance ratio of each algorithm across different arrival patterns as an indicator of an algorithm's robustness against process arrival imbalances. The No-delay arrival pattern is equal to running the Alltoall micro-benchmarks in Figure 7.

As shown in Figure 8a, in the *No-delay* scenario, Algorithms 1 and 4 are the optimal choices, with Algorithm 3 being approximately 4.6 times slower than these. Conversely, when process imbalance is present, Algorithms 1 and 4 are not the fastest. Specifically, in the Descending scenario, Algorithm 4 performs about 16 times slower than in the *No-delay* case. From this figure, it is evident that Algorithms 1 and 4 have inferior performance for most of the arrival patterns except for the No-Delay case. Therefore, selecting these two algorithms proves suboptimal for FT, as demonstrated in Figure 7a. This highlights why the fastest algorithm in a micro-benchmark may not be the quickest in application settings due to process arrival imbalances. In addition, the Average normalized values of each column (last row) serve as a useful indicator for algorithm selection, showing performance consistency across all patterns. Interestingly, these averaged values closely correlate with the runtimes observed in FT using these algorithms, as illustrated in Figure 7a.

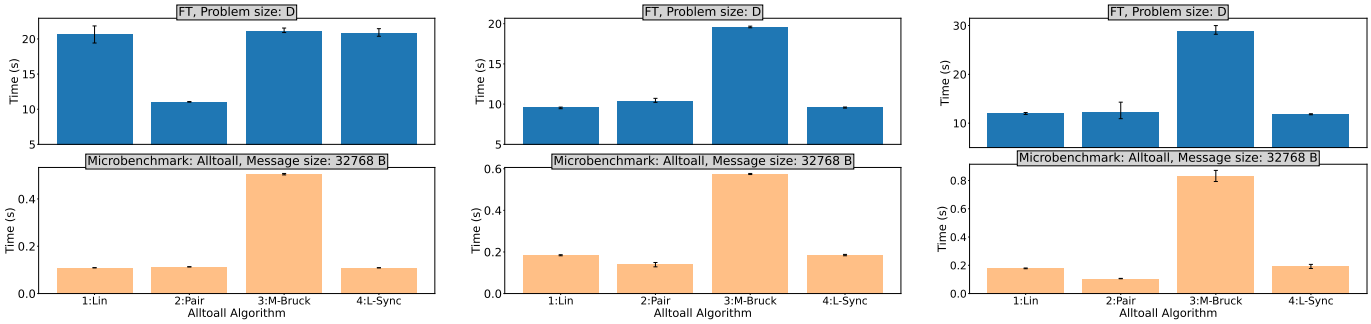
Figure 8b shows the same experiment for *Galileo100*, where we can observe a similar effect. As in Figure 7b, Algorithm 2 is the fastest in the No-delay scenario, in which it is more than 30% faster than Algorithms 1 and 4. When algorithms are exposed to process imbalance, e.g., the FT-Scenario, Algorithms 1 and 4 become around 16% faster than Algorithm 2. On *Discoverer* (Figure 8), the fastest algorithm in both the No-delay and FT-scenario cases is the same, and overall, Algorithm 2 demonstrates the most robust performance across all arrival patterns. This robustness explains why Algorithm 2 is the fastest in both the micro-benchmark and FT, as shown in Figure 7c.

Overall, the real-world arrival pattern of the application (FT-Scenario) enables us to accurately predict the best-performing algorithm within the application. Additionally, our proposed micro-benchmarking technique assists in identifying the most robust algorithm for each data size and machine.

C. Optimize for Robustness against Arrival Patterns

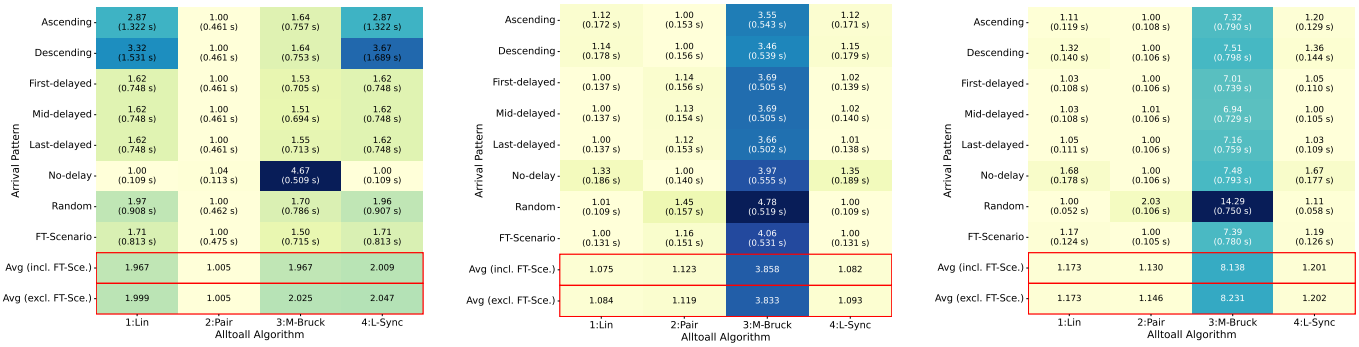
The final question to be addressed is how to perform algorithm selection in practice. In the preceding sections, we demonstrated that the runtime of MPI_Alltoall with the FT application significantly relies on the process arrival pattern. However, in this analysis, we utilized a pre-recorded arrival pattern trace. In real-world scenarios, tracing each application before deciding which collective algorithm to use is impractical. Therefore, we require an approach for optimizing collective algorithms in general cases.

Our approach to optimizing collective selection involves choosing the most robust algorithm against artificially created arrival patterns. Our rationale is as follows: an algorithm that



(a) *Hydra* (b) *Galileo100* (c) *Discoverer*

Fig. 7: Uncorrelated runtimes of FT (problem size D) and `MPI_Alltoall` benchmark on *Hydra*, *Galileo100*, and *Discoverer* with different algorithms for `MPI_Alltoall` and 32×32 processes. The `MPI_Alltoall` benchmark does not consider an arrival pattern (i.e., `No-delay` case).



(a) *Hydra* (b) *Galileo100* (c) *Discoverer*

Fig. 8: Normalized runtimes of Alltoall algorithms on *Hydra*, *Galileo100*, and *Discoverer* with message size 32 768 B and 32×32 processes. The reported runtimes are based on the *last delay* metric ($\hat{d}$).

consistently performs well across multiple arrival patterns will likely yield satisfactory results across various applications.

We show an example in Figure 9, where we compare the actual runtime of FT (the light blue bars) versus the expected (predicted) runtime of FT using the `No-delay` or the `Avg` delay runtimes on *Hydra*. We profiled FT with the mpisee profiler [26] and extracted the computation time. Our predicted runtimes (orange and dark blue bars) are then based on the sum of the computation time and the expected time for executing `MPI_Alltoall` in both scenarios, i.e., `No-delay` and `Avg (excl. FT-Sce.)` (refer to Figure 8). Due to the regular system noise, there is a small variation between the runtimes in the previous figures and Figure 9. It is evident that the Expected FT Runtime, based on the `No-delay` case, does not align with the Actual FT Runtime. As demonstrated with Algorithm 2, the discrepancy in performance stems from its response to process arrival patterns. Consequently, Algorithm 2 consumes more time in the actual application than in the micro-benchmark, resulting in a longer than anticipated FT runtime. If the average runtime of Algorithm 2, taking into account the various process arrival patterns as illustrated in Figure 8, had been used, the Expected FT Runtime would indeed match the Actual FT Runtime.

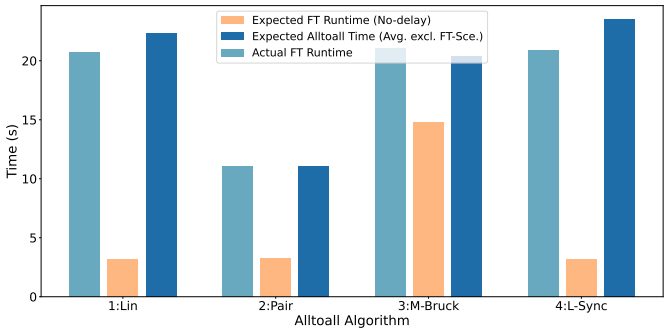


Fig. 9: The actual runtime of FT versus its imprecisely projected runtimes (when processes enter collectives simultaneously, the `No-delay` case, and the average case) on *Hydra* with 32×32 processes.

VI. Related Work

The problem of collective algorithm selection has been explored using various approaches and considering different parameters [1], [2], [3], [4], [5], [6], [27]. Process arrival times have been identified as an influential parameter in this selection process [10], [14], [28]. Research on process

arrival imbalances has explored system and architectural noise, diverse workloads [9], [29], synchronization points, and algorithmic properties [30], [31]. Additionally, current barrier algorithms and their implementations have been recognized as contributors to process arrival time imbalances when reaching collective communications [11], [20].

Faraj et al. [10] were among the first to demonstrate how arrival patterns affect MPI application performance. They showed that process arrival times at a collective operation influence not only the operation itself but also the overall performance of applications, even when MPI workloads are balanced. Other researchers have investigated the effect of different process arrival patterns on different collectives such as Broadcast [32], [28], Alltoall, and Allgather [33], and showed the deficiency of current algorithms in the presence of process arrival imbalance for inter-node communications especially when dealing with large data sizes. Parsons et al. [34] distinguished between the inter- versus intra-node arrival imbalance and tried to improve the performance of MPI collectives in the presence of imbalanced process arrival times by minimizing the synchronization delay of the early arriving processes. Alizadeh et al. [35] proposed an intra-node shared-memory process arrival pattern-aware Allreduce algorithm that imposes less data dependency among processes and evaluated the algorithm with Deep Learning workloads.

Marendić et al. [15] showed having prior knowledge of process arrival imbalances helps to implement a faster algorithm for collective operations, and since it might be expensive to achieve such knowledge, they proposed [36] a robust `MPI_Reduce` algorithm that dynamically re-balances the load between the processes regardless of the prior knowledge of the pattern. Proficz [13], [14] presented an online process arrival pattern detection, to estimate processes' arrival time and their distribution, and introduced optimized Allreduce and Scatter algorithms for imbalanced process arrival patterns. In another work [37], they proposed two process arrival pattern-resilient Allgather algorithms by utilizing an additional background thread for early data exchange from faster processes.

Widener et al. [38] examined the optimization potential of non-blocking collectives in noisy environments to mitigate the impact of process arrival imbalances. They used simulations to investigate the potential speedup of large-scale applications when non-blocking collectives are employed instead of blocking collectives. By using an idealized model of non-blocking collectives, they demonstrated that while non-blocking collectives do not automatically mitigate noise effects, they can be advantageous for certain types of noise, such as those from checkpoint/restart activities.

VII. Conclusions

Our work addressed the algorithm selection problem for MPI collective communication operations in the presence of process arrival patterns, typically found in real-world applications. We examined how different arrival patterns affect the performance of collective communication operations, highlighting their impact on each collective algorithm. In a simulation study, we showed a significant difference in behavior between different classes of MPI collectives when being exposed to process arrival patterns. We observed that rooted collectives, such as `MPI_Reduce`, have a greater potential to absorb waiting times caused by process skew compared to non-rooted collectives, especially `MPI_Allreduce`. We conducted a similar analysis using micro-benchmarking in real-world parallel machines and observed an outcome similar to the simulation study's.

We also showed how to enhance the algorithm selection process by considering process arrival patterns. To that end, we measured the runtime of collective algorithms for a variety of process arrival patterns and selected the algorithm that performs best overall across the different scenarios. We showed how to apply this strategy to the FT application from the NAS Parallel benchmarks. We implemented a collective tracing library that reports the processes' arrival times to the collectives. Using the tracer, we could record accurate process arrival times for the FT application and showed that algorithm selection without considering the process imbalance may lead to an inefficient choice. We also demonstrated that the algorithm selected by considering the arrival patterns while benchmarking does indeed improve the performance of FT.

VIII. Acknowledgement

This work was partially supported by the Austrian Science Fund (FWF): project P 33884-N.

We acknowledge the CINECA award under the ISCRA initiative, for the availability of high-performance computing resources and support.

We acknowledge PRACE for awarding us access to Discoverer at SofiaTech, Bulgaria.

References

[1] S. Hunold and S. Steiner, "OMPICollTune: Autotuning MPI collectives by incremental online learning," in *IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 2022, pp. 123–128.

[2] A. Faraj, X. Yuan, and D. Lowenthal, "STAR-MPI: self tuned adaptive routines for MPI collective operations," in *Proceedings of the 20th annual international conference on Supercomputing*, 2006, pp. 199–208.

[3] M. Wilkins, Y. Guo, R. Thakur, P. Dinda, and N. Hardavellas, "ACCLAiM: Advancing the practicality of MPI collective communication autotuning using Machine Learning," in *IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2022, pp. 161–171.

[4] M. Wilkins, Y. Guo, R. Thakur, N. Hardavellas, P. Dinda, and M. Si, "A FACT-based approach: Making machine learning collective autotuning feasible on Exascale systems," in *2021 Workshop on Exascale MPI (ExaMPI)*. IEEE, 2021, pp. 36–45.

[5] E. Nuriyev, J.-A. Rico-Gallego, and A. Lastovetsky, "Model-based selection of optimal MPI Broadcast algorithms for multi-core clusters," *Journal of Parallel and Distributed Computing*, vol. 165, pp. 1–16, 2022.

[6] M. Salimi Beni, S. Hunold, and B. Cosenza, "Algorithm selection of MPI collectives considering system utilization," in *Euro-Par: Parallel Processing Workshops*. Springer, 2023, pp. 302—-307.

[7] M. Salimi Beni and B. Cosenza, "An analysis of long-tailed network latency distribution and background traffic on Dragonfly+," in *International Symposium on Benchmarking, Measuring and Optimization (Bench)*. Springer, 2022, pp. 123–142.

[8] M. Salimi Beni, S. Hunold, and B. Cosenza, "Analysis and prediction of performance variability in large-scale computing systems," *The Journal of Supercomputing*, vol. 80, no. 10, pp. 14 978–15 005, 2024.

[9] S. Li, T. Ben-Nun, S. D. Girolamo, D. Alistarh, and T. Hoefler, "Taming unbalanced training workloads in Deep Learning with partial collective operations," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020, pp. 45–61.

[10] A. Faraj, P. Patarasuk, and X. Yuan, "A study of process arrival patterns for MPI collective operations," in *Proceedings of the 21st annual international conference on Supercomputing*, 2007, pp. 168–179.

[11] J. Schuchart, S. Hunold, and G. Bosilca, "Synchronizing MPI processes in space and time," in *Proceedings of the 30th European MPI Users' Group Meeting*, 2023, pp. 1–11.

[12] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber *et al.*, "The NAS parallel benchmarks—summary and preliminary results," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, 1991, pp. 158–165.

[13] J. Proficz, "Improving All-reduce collective operations for imbalanced process arrival patterns," *The Journal of Supercomputing*, vol. 74, no. 7, pp. 3071–3092, 2018.

[14] ——, "Process arrival pattern aware algorithms for acceleration of scatter and gather operations," *Cluster Computing*, vol. 23, no. 4, pp. 2735–2751, 2020.

[15] P. Marendić, J. Lemeire, T. Haber, D. Vučinić, and P. Schelkens, "An investigation into the performance of reduction algorithms under load imbalance," in *Parallel Processing: 18th International Conference, Euro-Par.* Springer, 2012, pp. 439–450.

[16] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Versatile, scalable, and accurate simulation of distributed applications and platforms," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2899–2917, Jun. 2014.

[17] "OSU Micro-Benchmarks," http://mvapich.cse.ohio-state.edu/benchmarks/.

[18] S. Hunold and A. Carpen-Amarie, "Reproducible MPI benchmarking is still not as easy as you think," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 12, pp. 3617–3630, 2016.

[19] A. Nigay, L. Mosimann, T. Schneider, and T. Hoefler, "Communication and timing issues with MPI virtualization," in *Proceedings of the 27th European MPI Users' Group Meeting*, 2020, pp. 11–20.

[20] S. Hunold and A. Carpen-Amarie, "Hierarchical clock synchronization in MPI," in *IEEE International Conference on Cluster Computing (CLUSTER)*, 2018, pp. 325–336.

[21] A. Degomme, A. Legrand, G. S. Markomanolis, M. Quinson, M. Stillwell, and F. Suter, "Simulating MPI applications: the SMPI approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 8, pp. 2387–2400, 2017.

[22] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: goals, concept, and design of a next generation MPI implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 19-22, 2004, Proceedings*, ser. Lecture Notes in Computer Science, D. Kranzlmüller, P. Kacsuk, and J. J. Dongarra, Eds., vol. 3241. Springer, 2004, pp. 97–104.

[23] D. K. Panda, H. Subramoni, C.-H. Chu, and M. Bayatpour, "The MVAPICH project: Transforming research into high-performance MPI library for HPC community," *Journal of Computational Science*, vol. 52, p. 101208, 2021.

[24] N. Sultana, M. Rüfenacht, A. Skjellum, P. V. Bangalore, I. Laguna, and K. M. Mohror, "Understanding the use of message passing interface in exascale proxy applications," *Concurr. Comput. Pract. Exp.*, vol. 33, no. 14, 2021.

[25] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. D. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, "Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir," in *Proceedings of the 5th International Workshop on Parallel Tools for High Performance Computing*, 2011, pp. 79–91.

[26] I. Vardas, S. Hunold, J. I. Ajanohoun, and J. L. Träff, "mpisee: MPI profiling for communication and communicator structure," in *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2022, pp. 520–529.

[27] S. Hunold, A. Bhatele, G. Bosilca, and P. Knees, "Predicting MPI collective communication performance using Machine Learning," in *IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2020, pp. 259–269.

[28] A. Ruhela, B. Ramesh, S. Chakraborty, H. Subramoni, J. Hashmi, and D. Panda, "Leveraging network-level parallelism with multiple process-endpoints for MPI broadcast," in *IEEE/ACM Third Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware (IPDRM)*. IEEE, 2019, pp. 34–41.

[29] S. Pumma, D. Buono, F. Checconi, X. Que, and W.-c. Feng, "Alleviating load imbalance in data processing for large-scale Deep Learning," in *20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 2020, pp. 262–271.

[30] I. B. Peng, S. Markidis, and E. Laure, "The cost of synchronizing imbalanced processes in message passing systems," in *IEEE International Conference on Cluster Computing*. IEEE, 2015, pp. 408–417.

[31] Y. Temucin, S. Levy, W. Schonbein, R. Grant, and A. Afsahi, "A dynamic network-native MPI partitioned aggregation over Infiniband verbs," in *IEEE International Conference on Cluster Computing (CLUSTER). IEEE*, 2023.

[32] P. Patarasuk and X. Yuan, "Efficient MPI Bcast across different process arrival patterns," in *IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 2008, pp. 1–11.

[33] Y. Qian and A. Afsahi, "Process arrival pattern aware Alltoall and Allgather on Infiniband clusters," *International Journal of Parallel Programming*, vol. 39, pp. 473–493, 2011.

[34] B. S. Parsons and V. S. Pai, "Exploiting process imbalance to improve MPI collective operations in hierarchical systems," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 57–66.

[35] P. Alizadeh, A. Sojoodi, Y. Hassan Temucin, and A. Afsahi, "Efficient process arrival pattern aware collective communication for Deep Learning," in *Proceedings of the 29th European MPI Users' Group Meeting*, 2022, pp. 68–78.

[36] P. Marendic, J. Lemeire, D. Vucinic, and P. Schelkens, "A novel MPI reduction algorithm resilient to imbalances in process arrival times," *The Journal of Supercomputing*, vol. 72, pp. 1973–2013, 2016.

[37] J. Proficz, "All-gather algorithms resilient to imbalanced process arrival patterns," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 18, no. 4, pp. 1–22, 2021.

[38] P. M. Widener, S. Levy, K. B. Ferreira, and T. Hoefler, "On noise and the performance benefit of nonblocking collectives," *Int. J. High Perform. Comput. Appl.*, vol. 30, no. 1, pp. 121–133, 2016.