The Journal of Supercomputing (2024) 80:23083–23119
https://doi.org/10.1007/s11227-024-06277-5

Proteo: a framework for the generation and evaluation of malleable MPI applications

Iker Martín-Álvarez[1] · José I. Aliaga[1] · Maribel Castillo[1] · Sergio Iserte[2]

Accepted: 30 May 2024 / Published online: 2 July 2024

Abstract

Applying malleability to HPC systems can increase their productivity without degrading or even improving the performance of running applications. This paper presents Proteo, a configurable framework that allows to design benchmarks to study the effect of malleability on a system, and also incorporates malleability into a real application. Proteo consists of two modules: SAM allows to emulate the computational behavior of iterative scientific MPI applications, and MaM is able to reconfigure a job during execution, adjusting the number of processes, redistributing data, and resuming execution. An in-depth study of all the possibilities shows that Proteo is able to behave like a real malleable or non-malleable application in the range [0.85, 1.15]. Furthermore, the different methods defined in MaM for process management and data redistribution are analyzed, concluding that asynchronous malleability, where reconfiguration and application execution overlap, results in a 1.15× speedup.

Keywords MPI · Emulation · Malleability · Dynamic resources

✉ Iker Martín-Álvarez
martini@uji.es

José I. Aliaga
aliaga@uji.es

Maribel Castillo
castillo@uji.es

Sergio Iserte
sergio.iserte@bsc.es

[1] Dpto. Ingeniería y Ciencia de los Computadores, Universitat Jaume I, Av. Vicent Sos Baynat, s/n, 12071 Castellón de la Plana, Spain

[2] Computer Sciences, Barcelona Supercomputing Center, Placa d'Eusebi Güell, 1-3, 08034 Barcelona, Spain

1 Introduction

Today, we are on the verge of crossing the exascale frontier in high-performance computing (HPC). Each year, the computing capabilities of the large-scale facilities show steady growth, a trend reflected in the prestigious TOP 500 list [1]. This progress is being driven by two main approaches. On the one hand, hardware improvements in memory, storage, or network communications, as well as massive parallelism in processors, provide brute force to the systems. On the other hand, novel programming models, runtimes, and libraries are capable of leveraging these new technologies.

In HPC systems, it is common to find distributed parallel jobs usually developed using the de facto standard Message Passing Interface (MPI) [2], while Resource Management Systems (RMSs) are responsible for controlling the available resources of HPC systems and determining how they are allocated to jobs. However, it is often the case that some resources are not used while there are jobs in the queue, since the resource requirements of the waiting jobs are greater than the available resources. In addition, some jobs in progress do not use all their resources efficiently during execution, for example, because they do not use all of their resources during the entire execution or because they have spare nodes for fault tolerance [3].

In the context of future exascale systems, it would be highly beneficial for applications to have dynamic behavior capable of adapting resources to the needs and/or availability of the system. Consequently, the system should be able to allocate resources dynamically, while applications should adapt to this mode of operation at runtime, all the while ensuring that the system can run at peak performance without compromising application performance.

Malleability allows applications to change their allocated computational resources during runtime. When and how to enable this change is controlled by the RMS, which must balance two different benefits when making its decision. From the point of view of each individual application, it can improve the application performance when expanding the job. From the perspective of the global system, it can increase its throughput in terms of jobs completed per unit of time.

In this paper, we consider malleability as the ability of a distributed parallel job to change its size, in terms of MPI ranks, by changing the computational resources initially allocated to the job at any point in the execution as many times as required. This is triggered at a specific point where processes are synchronized throughout execution, called the Malleability Point (MP). Defining MPs at the beginning of a loop may be the simplest option for iterative applications, while for non-iterative applications, defining a MP at the beginning of each phase is a good alternative.

The first task in a MP is to contact the RMS to find out whether the application needs to be reconfigured, since the RMS is responsible for making that decision. If a reconfiguration is proposed, i.e., the number of processes in a parallel job is changed from NS processes (sources) to NT (targets), the following stages must be performed:

1. *Stage* 1*: Resources reallocation.* Allocate new resources and/or release previously allocated resources to/from a job.
2. *Stage* 2*: Process management.* Spawn/kill processes according to the reconfiguration decision in the previous stage.
3. *Stage* 3*: Data redistribution.* Communicate data between NS and NT processes so that execution continues properly using the target processes.
4. *Stage* 4*: Resuming execution.* Resume execution of the job at the same point as before the reconfiguration began.

This paper presents a highly configurable framework, named Proteo,[1] that allows to set up benchmarks to study the effect of malleability and also to integrate malleability in real applications. This tool has been developed using a modular structure, with two main independent components: the Synthetic Application Module (SAM) and the Malleability Module (MaM), both of which include performance monitoring. One is used to emulate the computational behavior of iterative scientific MPI applications, although it can be extended to non-iterative applications. The other provides the ability to reconfigure a job during its execution, simulating RMS demands, by expanding or shrinking the number of assigned processes. Moreover, each benchmark is generated from a configuration file that details the main features of the computational behavior of the emulated application as well as the description of the different reconfigurations.

Proteo can also be used to facilitate the development of artificial workloads for a system, so that it is possible to analyze the impact of the malleability in their execution. Thus, a workload is composed of a set of benchmarks, each of which emulates a real application with certain computational properties described in the configuration file. This file also simulates how RMS adjusts system resources as their availability changes and determines when the benchmarks must release or acquire new resources. These workload emulations allow users to analyze the impact of malleability from both an application (overhead) and system (productivity) standpoint.

Following the previous comments, the main contributions of this work are presented below:

- Introduce Proteo and describe its main features, structure, and operation, showing how SAM is able to emulate the computational behavior of MPI scientific applications.
- Analyze in detail the emulation of a real application, defined by the parameters stored in a configuration file, and compare the behavior of the real and emulated application.

[1] Its name draws inspiration from Greek mythology, specifically from *Proteus*, the son of Poseidon, known for his ability to transform into various animal forms.

- Describe how Proteo using MaM is able to integrate malleability into an application, explaining the different mechanisms implemented for process management and data redistribution.
- Analyze the performance of a malleable application using the different mechanisms available in Proteo and compare the behavior of the real and emulated application when they become malleable.

In the analysis of the emulation of a real application and the corresponding malleable version, we will use the parallel implementation of the Conjugate Gradient (CG).

The rest of this paper is organized as follows: Section 2 introduces motivation and discusses previous works related to dynamic resources and malleability, mainly for MPI. Related work on malleability and simulation/emulation is discussed in Sect. 3. Section 4 describes the architecture and main features of Proteo, while Sect. 5 shows how Proteo emulates the behavior of a certain MPI parallel application when it is malleable and when it is non-malleable. Finally, Sect. 6 summarizes the paper and discusses future work.

2 Motivation and background

According to Feitelson and Rudolph's classification [4], applications executed in large-scale facilities can be categorized into four groups. These categories are defined by who and when determines the initial size (number of processes) of parallel jobs to be run and also by who and when the new sizes are set, if reconfiguration is supported. Thus, the job classification is the following:

- *Static job*: It maintains the initial resource allocation while running it.
  - *Rigid job*: It can only be run with a fixed number of processes.
  - *Moldable job*: It can be started with a variable number of processes. The size is determined by the RMS just before the job execution is launched.
- *Dynamic job*: It can change the initial allocation of resources during the execution of the job.
  - *Evolving job*: It is provided with a user-defined reconfiguration scheme that specifies how and when the job changes its resources. The RMS must meet the requests, or the job will not be able to continue its execution.
  - *Malleable job*: It can be reconfigured during its execution if the RMS decides to do so.

At the first level, the criterion is whether the resources allocated to the jobs may vary during their execution and, therefore, whether jobs should include some code to manage reconfigurations, as is the case with dynamic jobs. At the second level, the criterion is who determines the resources allocated to the jobs: For rigid and evolving jobs, it is the user, whereas for moldable and malleable jobs, the RMS should make the decision. Therefore, malleable jobs are the most flexible because they can adapt to system workloads and reallocate their resources at any time, to enhance system throughput. The current studies conducted in the USA Exascale project [5] show that including malleable jobs will be very profitable for both, applications and systems.

From an application point of view, it would allow a job to start with a suboptimal resource allocation and later, when available, claim an efficient amount of resources for the job. This would reduce the waiting time and, thus, the turnaround time. A number of papers have demonstrated a reduction in both metrics by employing malleable jobs [6–8]. Another beneficial consideration for jobs is to take advantage of situations where there are free nodes and no jobs in the queue that require those nodes. In this case, these nodes would be requested to reduce the job execution time.

From a system point of view, there are two main benefits. The first is based on a higher utilization of system resources, avoiding situations where there are free nodes while there are jobs in the queue or in execution that could use them. An efficient distribution of nodes makes it possible to assign to each job only the number of nodes that will give it maximum efficiency. In this way, it is possible to increase the throughput of the system [6–8]. On the other hand, it is possible to focus on another efficient distribution related to increasing the energy efficiency of each job [9–11]. In this distribution, the amount of resources given to each job is focused on improving its performance per watt. Therefore, a system can be made green and sustainable through malleability.

An in-depth analysis of how malleability can be applied in a system and its impact on the performance of both the running workload and the system itself is provided in [12]. This paper shows that there are many malleability solutions that aim to reconfigure jobs on-the-fly with different approaches for managing the MPI processes [6, 13–16], the most relevant of which are introduced below. In Elastic-MPI [14], the MPI-2.0 standard is extended along with modifications to SLURM [17] to allow the execution of moldable and malleable jobs. ReSHAPE [13] developed a framework consisting of three components: an application scheduler based on the DQ/GEMS project [18], a monitoring module, and a programming model for reconfigurations. Furthermore, AMPI [19] presents a regular MPI implementation used for the dynamic runtime system CHARM++ [20] and integrated for Maui/Torque [15], allowing the execution of evolving jobs. Another solution is Flex-MPI [16], a library built on top of MPICH [21], to improve the performance of applications by making them malleable. A malleability framework is presented in DMR [6], which consists of two components: a parallel distributed runtime based on MPI and an extension of SLURM to enable the execution of malleable jobs. All these solutions require an RMS to support the scheduling of malleable jobs. Some works explore the new concept of MPI Sessions, extending it to implement on-the-fly malleability [22, 23].

In works where malleability is integrated using MPI, the authors evaluate their frameworks generating synthetic workloads of both: benchmarks such as Conjugate Gradient, N-body, NAS Parallel,[2] etc., [24–26] and scientific applications such as LAMMPS, HPG-aligner, LeanMD,[3] etc. [9, 27–33].

[2] http://www.nas.nasa.gov/Software/NPB

[3] http://charm.cs.illinois.edu/research/leanmd

These codes have been developed ad-hoc for each particular malleability framework and flocked together in workloads to evaluate each malleability solution and measure its impact on HPC systems.

Other studies implement malleability by focusing on fine-grained malleability, where jobs are divided into a task graph with dependencies. These tasks are executed based on the specified dependencies, and each task can use a different amount of resources [34, 35]. Alternatively, in some cases, the number of threads allocated to a task can be dynamically modified during its execution [36, 37].

Finally, there is ongoing research on scheduling policies for RMS aware of malleable jobs using simulators [7, 8, 38, 39] and execution time [40], which aims to evaluate how malleable workloads can improve the makespan of RMS and/or applications.

3 Related work

Proteo is a very powerful tool for analyzing in advance how the dynamic allocation of resources will affect the performance of both the system and the applications themselves during execution. Currently, there are not many parameterizable tools that can emulate the behavior of parallel applications running with dynamic resources while monitoring the entire execution. The most similar solutions that have been found are explained in this section.

In this context, ElastiSim [39] is a batch-system simulator for testing different scheduling algorithms for rigid and/or malleable workloads. It is based on the framework SimGrid [41], a simulator for distributed computing systems that characterizes resources as compute nodes, network topologies, and file systems. ElastiSim consists of the simulation engine built on top of SimGrid, enhanced with resource management, GPU utilization, and new *I*/*O* semantics that integrate model-relevant interactions into workload management for large distributed computing infrastructures. Thus, the simulator provides an interface for users to implement their own scheduling protocols. It also describes workload modeling, where jobs are broken down into a set of properties, and the application model. The former contains information relevant to the scheduler, while the latter describes the application so that it can be simulated.

The main difference between ElastiSim and Proteo is that the former is a simulator, while the latter is an emulator, so the execution can be performed in the actual architecture where the user application is running. In addition, ElastiSim focuses on how the system responds to a malleable workload with different scheduling algorithms, while Proteo analyzes how applications would respond to different reconfiguration alternatives and can be used to generate malleable workloads.

Similar works can be found in BatSim [42] and Alea [43], simulators built on top of SimGrid for testing different scheduling algorithms for rigid workloads. They have many similarities to ElastiSim, but they do not take malleability into account.

Another simulator is Elastic-Sim [8], which focuses on partially or completely transforming real rigid workloads into malleable ones. It then uses a variety of metrics, such as system utilization or turnaround time, to determine whether adding malleable jobs to HPC systems is beneficial. The simulator takes a rigid workload from a real system as input and simulates its execution using four different queues. Jobs move from one queue to the next until they are completed. Malleable operations are divided into expansion and shrinkage operations, which double and halve the amount of resources used by a job, respectively. In addition, these operations can use two different policies: conservative or aggressive. The former allows a single malleable operation to be applied to a job, while the latter allows the same operation to be applied multiple times to a job in a single reconfiguration.

As with ElastiSim, the main difference between Elastic-Sim and Proteo is that the latter is an emulator. Also, Elastic-Sim focuses on how system metrics are affected by malleable jobs and does not consider all the overheads caused by reconfigurations, whereas Proteo gives more importance to these overheads and how they can be reduced. Furthermore, Proteo allows you to perform reconfigurations on any number of resources.

A technique for generating malleable workloads is presented in [44], which employs LIMITLESS [45], an HPC framework that provides strategies for monitoring clusters. With this monitor, it is possible to create synthetic micro-benchmarks (proxies) based on monitored data from real applications, which are made malleable by using Flex-MPI [16], a performance-aware reconfiguration library.

A limitation of Flex-MPI is that the initial amount of resources cannot be reduced or removed, whereas in Proteo, it is possible to reconfigure from any number to any other number. Another important difference is found in the reconfiguration stages, where Flex-MPI has only one way to complete them, while Proteo has several alternatives to choose from. On the other hand, the monitoring provided by LIMITLESS simplifies the modeling of applications.

In other cases, a sleep-based benchmark [46] has been developed to implement malleability. This type of benchmark is useful for configuring custom iteration times, but it is not able to distinguish between compute/communication stages within an iteration. Stage differentiation and parameterization can provide the freedom needed to emulate the workflow of a particular application. For this purpose, Proteo is highly configurable to reproduce behavior profiles of scientific applications. In this regard, users will be able to create synthetic twins of their application where malleability can be easily adopted and evaluated in workloads.

In summary, Proteo can be used to generate a synthetic malleable workload that, once executed, can be compared to its traditional non-malleable counterpart and to determine the best way to perform reconfigurations in different applications. The adoption of malleability in the HPC system, as well as the malleability fine-tuning configuration of the applications, will be evaluated after a post-mortem analysis of these workload executions.

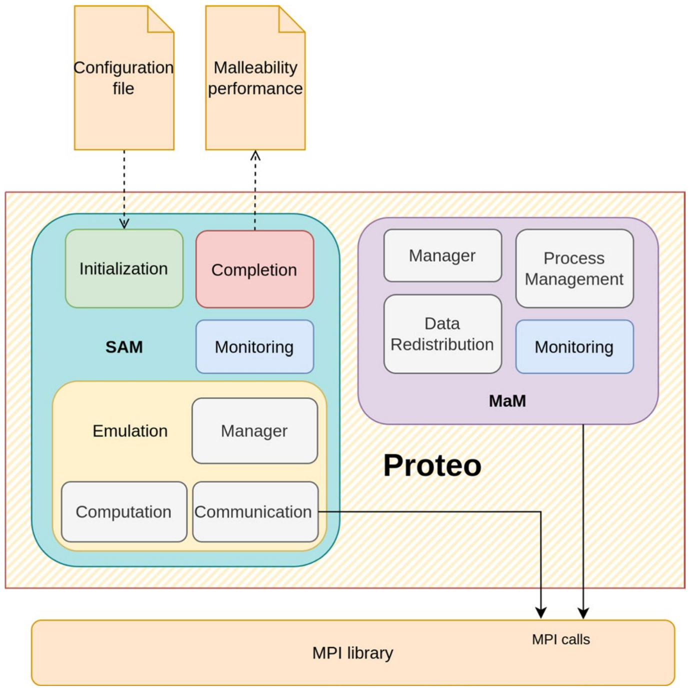


Fig. 1 Architecture of Proteo

4 Description of Proteo operation

Proteo is a framework designed with a modular structure, as shown in Fig. 1. The tool has two main independent modules: the Synthetic Application Module (SAM) and the Malleability Module (MaM), each of which is decomposed into several submodules. A monitoring submodule is included in both of them, to measure the performance of each, which is stored in an output file for later analysis. There is also an input configuration file where the parameters for emulating the computational behavior of a real application and the malleability are stored.

The following subsections introduce the Proteo modules and how they work, before describing the application workflow and how to configure it.

4.1 Synthetic application module

This module is able to emulate the computational behavior of any type of MPI parallel iterative application from the parameters stored in the configuration file. This task is performed in four main parts: *Initialization*, *Emulation*, *Monitoring*, and *Completion*.

Initialization: It is in charge of starting the execution of Proteo, and its main task is performed by the first process group (the ones that start the execution), which reads the parameters from the configuration file. Later, these parameters are transferred through the process groups after each reconfiguration. This module is also responsible for initializing MaM and starting the emulated execution.

In addition, (non-direct) configuration parameters are also calculated by combining other parameters from the configuration file and, in some cases, by combining the number of source (NS) and target (NT) processes. Source processes indicate the number of processes before a reconfiguration, while target processes indicate the number after a reconfiguration. These non-direct configuration parameters should be recalculated after each reconfiguration.

Emulation: It is able to emulate iterative applications where an iteration is composed of a sequence of *processing stages*, in which some computations or communications are performed. Thus, an iteration is considered to be terminated when all its processing stages have been completed. There are two types of computation:

- *Compute-bound applications* (0): where most of the computation are done by the CPU. The emulation executes a π estimator based on Monte Carlo method.
- *Memory-bound applications* (1): where most of the execution time is wasted on memory accesses. The emulation is based in a matrix-matrix product in a column-major order.

In addition, there are other types of communication, as described below:

- *Point-to-Point* (2): All processes are involved in a `MPI_Sendrecv`, choosing a communication pattern that ensures that all processes perform communication across nodes.
- *Collective one-to-all* (3): All processes take part in a `MPI_Bcast`.
- *Collective all-to-all* (4): All processes participate in a `MPI_Allgatherv`.
- *Reduction* (5 and 6): All processes execute a `MPI_Reduction`, or `MPI_Allreduce`, with a `MPI_SUM` operation.
- *Non-Blocking Point-to-Point* (7): All processes are involved in a `MPI_Isend` and `MPI_Irecv`, choosing a communication pattern that ensures that all processes perform communications across nodes. When used, a *Wait communication* stage should also be included.
- *Wait communication* (8): Performs a `MPI_Waitall` function over a previous set of non-blocking communications.

The total number of iterations and the parameters that define the computation and communication operations associated with each processing stage of an iteration are stored in the configuration file.

Monitoring: This submodule supervises the execution and measures the performance of the application, using the `MPI_Wtime` function to obtain the times of each part of the emulation. One of the main goals of Proteo is to measure how the computation and communication operations are executed, allowing a very detailed analysis of the application behavior when malleability is incorporated.

Completion: This submodule is responsible for completing the execution of each process group and saving the measurements taken by the monitoring submodule to an output file for further analysis. It is executed when a process group completes its execution after a reconfiguration or when the last process group ends.

4.2 Malleability module

This module is responsible for reconfiguring applications by changing the number of running processes. The values of NS and NT are set in the configuration file, as well as when the malleability is applied.

In an application, the reconfiguration triggered at a MP is completed in the following four steps:

1. Communicate with the RMS to indicate the availability of the application to be reconfigured, increasing or decreasing the number of processes if necessary.
2. If the RMS has approved the reconfiguration, create new set/group of MPI processes.
3. When the RMS has approved the reconfiguration, redistribute data from sources to targets.
4. Targets resume execution at the point where the sources stopped the application.

Step 1 is simulated by the input parameters in the configuration file, which indicate whether the application will be expanded or shrunk, thus consuming more or less computing resources on the system. Since MaM simulates the amount of resources, it is designed as a framework independent of the RMS used, if any. Instead, the *process management* and *data redistribution* submodules perform Steps 2 and 3. In addition, the *monitoring* submodule works similarly as described in 4.1.

Process management: The creation of processes in MPI is based on the `MPI_Comm_spawn` function, which is a collective operation on a given communicator.

Using this MPI function, two main methods can be used to change the number of processes in execution:

- *Baseline* method, where NT new processes are always spawned to continue execution as targets while all sources terminate.
- *Merge* method, where some sources continue executing after reconfiguration: When expanding, only NT–NS new processes are spawned, while NS sources persist; when shrinking, NT–NS processes are stopped, while NT sources persist.

For *Baseline*, the use of inter-communicators is the most common alternative to communicate between sources and targets, although the use of intra-communicators is also possible. For *Merge*, on the other hand, the use of intra-communicators is the only alternative to complete the communication.

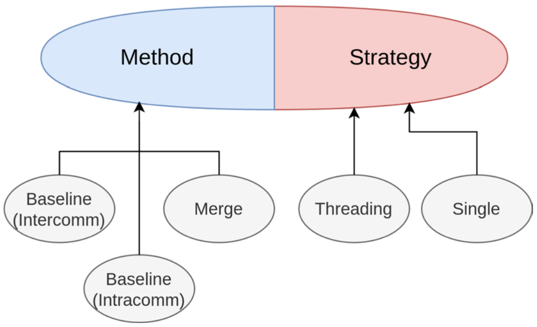


Fig. 2 Methods and strategies for process management

There are also two additional strategies that can be used in combination with the methods described above:

- *Asynchronous spawning of processes*: Auxiliary threads are used to perform the operation. This allows source processes to continue executing the application while the reconfiguration is performed by threads. This strategy requires starting the MPI environment with `MPI_Init_thread` and the argument `MPI_THREAD_MULTIPLE`.
- *Single operation*: When this strategy is enabled, only rank 0 will execute the `MPI_Comm_spawn` function instead of all the processes in the communicator.

Figure 2 shows a diagram of the different methods and strategies that can be selected to perform the application reconfiguration. One method should always be selected, while none, one, or both strategies may be enabled. Although all of these techniques are explained in detail in [47], some combinations are described below.

Figure 3 shows some examples using both methods. It can also show examples with and without asynchronous and single strategies. Examples include expansion, when the application is reconfigured from 2 source to 4 target processes, and shrinkage, from 4 source to 2 target processes. In all cases, the source processes are initially available to run the application, and after the reconfiguration, the target processes eventually continue to run the application while the sources terminate. Figure 3a, b, and c shows a synchronous operation using *Baseline* and *Merge* methods, where the source processes stop their execution before the reconfiguration begins and continue on the target processes after the reconfiguration. Two figures are shown for *Merge* because the procedure for expanding and shrinking is different for this method than for *Baseline*. Figure 3d, e, and f shows the asynchronous versions of the previous combinations, where each source creates an auxiliary thread which is responsible for spawning the new processes, while the source processes continue execution. In these cases, the auxiliary threads will terminate as soon as the target processes are spawned, but these processes may need to wait for the sources to be notified that the reconfiguration is complete; these waits are marked by striped blocks in the figures. Figure 3g and h shows how asynchronous and single strategies can be combined for both methods. Finally, the combination of *Merge* and single strategy is shown in Fig. 3i.

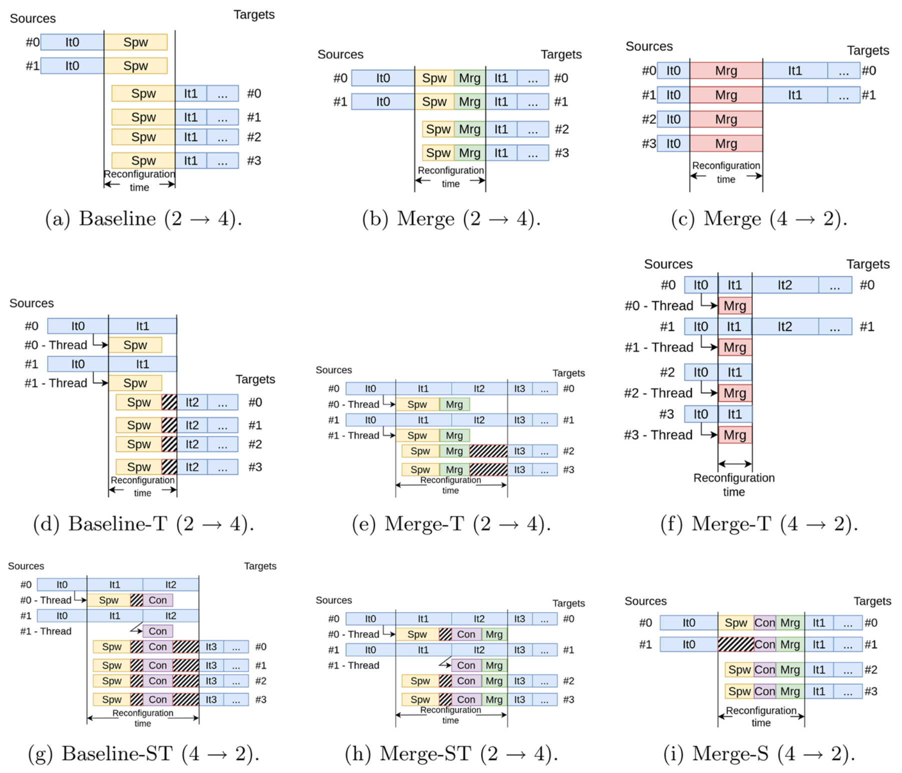


Fig. 3 Reconfiguration methods from *Source* to *Target* processes. *ItX* is iteration number, *Spw*: spawn, *Mrg*: merge processes, and *Con*: connect operation. Striped blocks are waiting time. Methods and strategies: Baseline-/Merge-with *S* (only rank 0 performs spawn), *T* (threading), or nothing

Data redistribution: To move data from source to target processes, a communicator is needed to connect the appropriate groups of processes. Depending on the type of communication, two main methods can be used:

- *Point-to-Point* (P2P): is initially based on simple MPI functions such as `MPI_Send` and `MPI_Recv` where the operation is performed by communicating between two processes at a time. However, a more efficient implementation suggests using `MPI_Isend` and `MPI_Irecv` along with `MPI_Wait`.
- *Collective* (COL): based on MPI functions like `MPI_Alltoall` or `MPI_Alltoallv`, where all active processes are involved at the same time.

Data redistribution is an expensive task that could be reduced by overlapping it with application execution in source processes. The first consideration is that the communicator used for data redistribution and the one used for the application should be different to avoid communication deadlocks. This is usually true since data redistribution uses the inter- or intra-communicator created during process management.

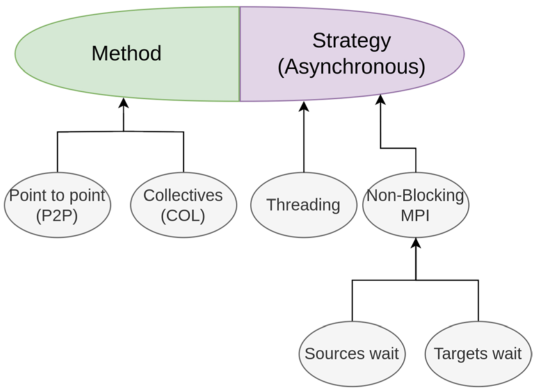


Fig. 4 Methods and strategies for data redistribution

The second consideration is that the overlapping can only be done on data that are maintained throughout the execution of the application (constant data), whereas the other data (variable data) require the sources to stop before the data are redistributed.

There are several strategies to ensure that sources continue to compute while constant data communication is completed:

- *Non-blocking MPI functions*: For P2P, the alternative is to combine `MPI_Isend` and `MPI_Irecv` functions with the `MPI_Test` function, while for COL, the use of the `MPI_Ialltoallv` function is the most common solution, and the verification of the function completion also requires the use of the `MPI_Test` function. In this case, there are two options: (i) wait sources: The redistribution is considered complete for sources when all of their data have been sent or (ii) wait targets: Communication is complete for sources when all data have been successfully received at all targets.
- *Managing threads*: The main goal of creating auxiliary threads is to relieve sources of the responsibility of redistributing data. This way, the auxiliary threads handle the communication, while the main ones continue to compute. The simplest alternative is to launch a new thread in each source that will take care of data redistribution using the selected method. This strategy also requires starting the MPI environment with `MPI_Init_thread` and the argument `MPI_THREAD_MULTIPLE`.

Figure 4 shows a diagram of all the methods and strategies that can be selected to perform data redistribution during application reconfiguration. Again, one method should always be selected, while none, one, or both strategies can be enabled. These techniques are explained in more detail in [48].

4.3 General workflow

Figure 5 shows the workflow diagram of the tool, with the different parts of Proteo being highlighted by colors as shown in Fig. 1: *Initialization* in green, *Emulation* in yellow, *Malleability* in purple, *Monitoring* in blue, and *Completion* in red.

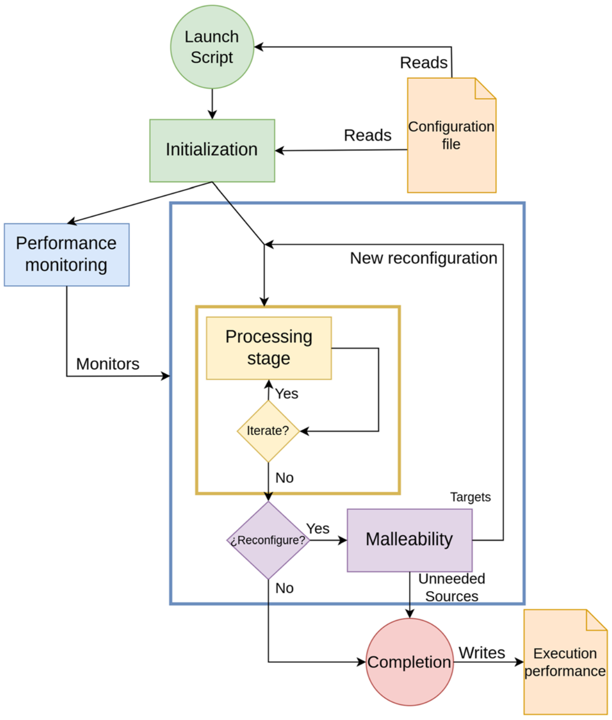


Fig. 5 Flowchart of Proteo

The first process group starts the execution at the *Initialization* submodule. In fact, a single process in this group is responsible for reading all the parameters from the configuration file and storing them in an internal data structure. Later, this structure will be broadcasted to the other processes in its group and also to the other process groups after each reconfiguration.

Then, *Emulation* submodule starts, which is responsible for computing the total iterations programmed for each process group, taking into account the application computational behavior described in the configuration file. At the MP of each iteration, this module checks whether a new reconfiguration is defined in the benchmark. If this is the case, execution continues at the *Malleability* module, where two tasks are accomplished: process management and data redistribution from sources to targets. Then the new group will continue the application execution from the *Initialization* submodule, which will lead it to execute *Emulation* again, and the execution will continue to run the iterations defined for this new group. When the malleability is finished, sources (or part of them) complete their execution in *Completion*, which is responsible to store the application performance recorded by *Monitoring* submodule in an intermediate file.

Listing 1 shows the skeleton of the emulation using Proteo, where the lines associated with each module and part are grouped by color. Lines 2–11 initialize the application and process groups; lines 15–19 represent an iterative computation

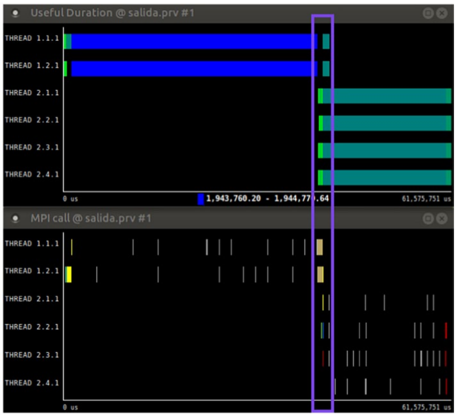


Fig. 6 Emulation of an iterative application with two stages: computation and communication. Reconfiguration from 2 to 4 processes

with its different computation and communication stages (processing_stage); reconfigurations take place from lines 22– 24; and the results are stored in line 27.

```
 1 int main() {
 2   if(init) { // First group initializes the application
 3     if (rank == 0)
 4       read_config();
 5     broadcast_config();       // Send to all ranks in group
 6     calculate_non_direct_parameters();
 7   } else {
 8     broadcast_config();       // Receive from source rank 0
 9     calculate_non_direct_parameters();
10     redistribute_data();      // Receive from sources.
11   }
12
13   do {
14     // Job computation
15     for(int iter=0; iter < Iters; iter++) {
16       for(int i=0; i < n_stages; i++) {
17         processing_stage(stages[i]);
18       }
19     }
20     if (!last_reconf) {   // Reconfigure if not last group
21         // Source could persist if the Merge method is used
22         alive_source = create_target_processes();
23         broadcast_config();  // From rank 0 to all targets
24         redistribute_data(); // Send to targets
25     }
26   } while (!last_reconf && alive_source);
27   store_performance_data();
28 }
```

Listing 1 Basic skeleton of the synthetic application module

Figure 6 shows the emulation of a parallel iterative application using Proteo with a reconfiguration from 2 to 4 processes, running 20 iterations before and after the reconfiguration using Extrae.[4] The upper part of the figure shows the computation time, whereas the lower part includes the MPI calls. In this example, SAM is configured with two stages: a compute-bound computation and a `MPI_Allgather` communication. For MaM, the baseline method without strategies for reconfiguration is used. Therefore, after the reconfiguration, the two sources stop executing, and the four targets start executing. The purple box in the figure identifies the two main reconfiguration steps: the processes management and the data redistribution. In the figure, each row has three numbers on the left (X.Y.Z) that indicate the group identifier, the process identifier within the group, and the thread identifier within the process.

4.4 Configuration

The configuration file contains two types of parameters to generate a malleable benchmark: One defines the computational behavior of the emulated application (SAM), and the other specifies when and how reconfigurations are performed (MaM).

The parameters that determine how the application execution is emulated are as follows:

- `Total_Stages`: The number of processing stages into which each iteration of the emulated application is broken down. Each processing stage will be defined by a computation or communication operation, and its value will always be equal to or greater than 1.
- `Granularity`: Problem size for computation stages. Lower values allow the use of finer-grained tasks when emulating a stage, so that the execution time can be better adjusted.

For each stage, the parameters described below should be specified:

- `Stage_Type`: Computation(0–1); Communication(2-8).
- `Stage_Time`: Computational or communication time in seconds.
- `Stage_Bytes`: Bytes communicated among processes.

SAM never considers a single stage which combines communication along computation, although there are some implicit computations in the reduction communication stages. Nevertheless, a non-blocking communication could be carried out while performing other stages.

With respect to MaM, the following parameters describe how the reconfiguration will be developed in the malleable benchmark:

- `Total_Reconfigurations`: Number of reconfigurations to perform during the benchmark execution. A value of zero indicates that the benchmark is not malleable.

[4] https://tools.bsc.es/extrae.

- `Total_Data_Redistribution`: Number of bytes to redistribute in each reconfiguration.
- `Asynch_Percentage_Redistribution`: Percentage of `Total_Data_Redistribution` that will be distributed asynchronously (constant data), while the rest will be distributed synchronously (variable data). It is a number between 0 and 100.

Next, `Total_Reconfigurations` + 1 sets of parameters are included characterizing each reconfiguration. For each set, the parameters are the following:

- `Iters`: Total iterations to execute before starting a new reconfiguration or ending the application.
- `Procs`: Number of processes in the group.
- `Dist`: Strategy applied to emulate the node allocation: *spread* or *compact*, i.e., minimize or maximize the number of processes per host.
- `Spawn_Method`: Spawn method to use during reconfiguration: Baseline (0) or Merge (1).
- `Spawn_Strategy`: Spawn strategy to use during reconfiguration: none (1), one of them (threading 2, or single 3), or both (6).
- `Redistribution_Method`: Data redistribution method to use during reconfiguration: Collectives (0) or Point-to-Point (1).
- `Redistribution_Strategy`: Data redistribution strategy, either using none (1); using MPI non-blocking primitives with sources wait (2) or target wait (3); or *threads* (5).
- `FactorS`: This parameter describes how the performance of computation stages is affected by the number of processes. For ideal scalability, its value is $1/Procs$, and greater values characterize loss of performance.

The parameters `Spawn_Method`, `Spawn_Strategy`, `Redistribution_Method`, and `Redistribution_Strategy` are ignored for the first process group, as they are created by *mpirun* or similar commands.

Some other parameters have to be computed each time a new process group is started. If computations have to be done in one stage, i.e., `Stage_Time` is greater than 0, the tool has to calculate:

- T_{op}: Time to execute a single operation of `Stage_Type` (π estimator or matrix-matrix product), when problem size is equal to `Granularity`. The different values of T_{op} are set when Proteo is installed or the first time it is executed.
- `op`: How many times the `Stage_Type` has to be executed in a stage to achieve the scalability defined in `FactorS`, and it is calculated as follows:

$$\text{op} = \text{Stage\_Time} * \text{FactorS}/\text{T}_{\text{op}}$$

This parameter has to be computed after each reconfiguration.

The user manual of Proteo appears in [49], where all parameters are explained in detail, and examples of configuration files are shown.

5 Results

This section evaluates Proteo as an emulation tool and shows how it can be used to integrate malleability into an application. The analysis is divided into several sections, each focusing on the evaluation of different aspects of Proteo.

First, we describe the main features of the Conjugate Gradient application, since this is the real application that we will emulate with SAM, considering both the algorithm and the problem size, before introducing the setup for the experiments. Finally, we present the multiple tests that have been carried out to validate the usefulness of Proteo: first, to emulate an iterative application using SAM; then, to build malleability into a real application using MaM; and finally, to emulate a malleable version of the application using a combination of SAM and MaM. Sources and experimental results can be found in the GitLab repositories for Proteo [50] and the CG [51], and in Zenodo [52].

5.1 Description of the use case

Our experiments with SAM emulate the execution of the Conjugate Gradient application (CG), which solves a sparse linear system defined by a sparse matrix.

CG is the most common solver for the resolution of positive-definite sparse linear systems ($Ax = b$). It is an iterative method in which the solution is obtained as the projection of an initial vector on to a Krylov subspace defined by the coefficient matrix and the residual [53]. In each iteration, one sparse matrix–vector product (SpMV), two Dot products, and three Axpy(-like) operations are computed. Given its relevance, a High-Performance Conjugate Gradient benchmark[5] was ultimately defined as a complement to the High-Performance LINPACK, which is currently used to rank the TOP500[6] computing systems.

Assuming that a row-block distribution is used for sparse matrix and vectors, parallelizing CG requires a proper implementation of its operations:

- *Parallel computation of SpMV*: Each process needs a full version of the distributed vector before it can perform local computation. For this reason, the `MPI_Allgatherv` function is executed.
- *Parallel computation of Dot product*: The Dot products calculate scalars that are needed in all processes. Therefore, after the local calculations with the distributed vectors, the `MPI_Allreduce` function is executed.
- *Parallel computation of Axpy*: As the same distribution is applied to all vectors, this computation can be fully parallelized, requiring no communication.

In summary, it can be concluded that the parallel CG algorithm consists of three communication operations: one `MPI_Allgatherv` and two `MPI_Allreduce`.

[5] https://hpcg-benchmark.org.

[6] https://www.top500.org.

Therefore, to emulate the parallel CG algorithm, six different *processing stages* need to be defined in SAM. Three of these stages involve intensive matrix computation, while the other three involve communication: two `MPI_Allreduce` operations, each for accumulating a double, and one `MPI_Allgatherv` operation for obtaining a duplicate vector of N doubles. Here, N represents the number of rows in the matrix.

5.2 Hardware and software setup

The experiments were conducted using three different clusters of nodes; although depending on the type of experiment and evaluation, only one of them was used. Below is a brief description of each system:

- *System_1*: Consists of eight servers with two 10-core Intel Xeon 4210 processors for a total of 160 cores. The nodes are interconnected with an EDR Infiniband network of 100 Gb/s. The used version of MPI was MPICH 4.0.3 [21], compiled with CH4:OFI netmod (Infiniband)[7]. The analysis considers the simulation for 2, 10, 20, 40, 80, 120, 160 processes.
- *System_2*: Consists of 32 servers with two 10-core Intel Xeon E5-2680 V2 processors for a total of 640 cores in which IntelMPI 2021.6.0 is used. The nodes are interconnected via an EDR Infiniband (100 Gb/s) network, using the Open UCX interface. The analysis considers the simulation for 2, 10, 20, 40, 80, 120, 160 processes.
- *System_3*: Consists of 36 servers with two 16-core Intel Xeon E5-2697A V4 processors for a total of 1.152 cores in which IntelMPI 2021.6.0 is used. The nodes are interconnected via an EDR Infiniband (100 Gb/s) network, using the Open UCX interface. The analysis considers the simulation for 2, 16, 32, 64, 128, 192, 256 processes.

In order to ensure a fair comparison between systems in the experiments, the number of processes considered in each system has been taken into account. Therefore, with the exception of the case of 2 processes, the remaining values are related to the number of cores in each node, evaluating 0.5, 1, 2, 4, 6, and 8 nodes.

The benchmark that emulates the parallel CG was used to provide results, which were analyzed using various systems. Several BSC tools[8] were used to assess the impact of computation and communication on the parallel CG execution, and the resulting parameters were used to describe the application in the SAM configuration file. Extrae was used to gather information about the performance of the application, and Paraver was used to visualize the execution of the application so that unbalanced process executions and bottlenecks could be detected. Both tools were used to determine the computation and communication time of the real application in its

[7] MPICH supports dynamic processes for Netmod OFI but not for UCX [21].

[8] https://tools.bsc.es/extrae.

various processing stages. These data were then transferred to the configuration file so that SAM can emulate the computational behavior of the parallel CG.

For this study, two different matrices, obtained from the SuiteSparse Matrix Collection[9], were also used for the CG evaluation:

- *audikw_1* is a $943,595 \times 943,595$ sparse matrix with 77, 651, 847 nonzeros. Approximately a total of 0.965 GB of memory is allocated when using this matrix.
- *Queen_4147* is a $4,147,110 \times 4,147,110$ sparse matrix with 316, 548, 962 nonzeros. Approximately a total of 3.947 GB of memory is allocated when using this matrix.

Apart from their different sizes, the main difference is the effect of the row-block distribution on the performance of parallel CG, leading to unbalanced distributions for audikw_1 and balanced ones for Queen_4147.

In contrast, for the malleability evaluation, only *System_1* has been used. The reason for this is that the other two systems did not support process spawning during execution[10]. For each experiment, we consider a single reconfiguration from 2, 10, 20, 40, 80, 120, and 160 processes to any of the same numbers. The number of occupied nodes in each execution is calculated by taking the upper bound of $\lceil N/20 \rceil$, where N is the maximum between the number of sources (NS) and targets (NT), in order to minimize the number of nodes allocated by the RMS. For reconfigurations, two groups of processes, sources and targets, are defined, with the malleability stage starting at iteration 500 of 1000. These experiments were carried out using the *Queen_4147* matrix, which allocates approximately 3.947 GB of memory for the matrix and vectors. This is the number of bytes that will be redistributed during reconfiguration. 96.6% of these can be redistributed asynchronously. In the definition of the configuration file, all methods and asynchronous strategies included in MaM are used and evaluated, with the exception of *children wait* (see Sect. 4.2).

For all experiments, a total of five runs are conducted and the mean, standard deviation, and median of the measurements are calculated. Then, the Shapiro–Wilk [54], Kruskal–Wallis [55], and Post-hoc Conover [56] statistical tests are used to characterize the different configurations in relation to each pair of process groups. All configurations reject the Shapiro–Wilk null hypothesis (H0) that the data are from a normal distribution, so medians and nonparametric tests should be used. The Kruskal–Wallis test is used to check the H0 of the 12 configurations having the same median. For pairs of groups that reject H0, the Post-hoc Conover is performed to determine which configurations are different.

[9] https://sparse.tamu.edu.

[10] IntelMPI has a limitation that does not support calls to MPI_Comm_spawn() from within a Slurm allocation with PMIx. https://slurm.schedmd.com/mpi_guide.html.

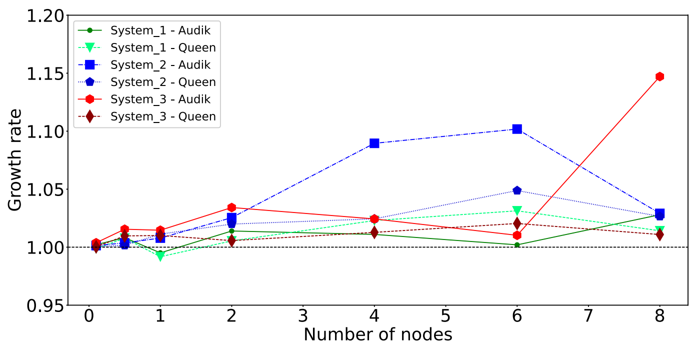


Fig. 7 Growth rate of the computational time of SAM with respect to parallel CG for each sparse matrix and HPC system

5.3 Emulation of a non-malleable application

This subsection evaluates the emulation of the parallel CG provided by SAM in three steps. Firstly, is evaluated the computation part. Secondly, is analyzed the `MPI_Allgatherv` function, which is the heaviest communication operation used in the emulation. Finally, we study how SAM emulates parallel CG by comparing the computation, communication, and total execution time of both.

Figure 7 shows the growth rate of SAM computational time with respect to the original one. It is computed as the quotient of the computational times of the two CG implementations using the same system and sparse matrix. Values greater than 1 indicate an increase in the cost of the emulated CG, while values lower than 1 indicate a decrease. Values close to 1 confirm a similar behavior. Since the computational emulation is implemented as a loop until the total time is reached, in general, the results of SAM are slightly more expensive than the original ones, falling within the interval [1, 1.05]. There are only a few cases of higher growth, all of which are related to the audikw_1 matrix, as this matrix produces unbalanced workloads.

The most communication-intensive operation in the CG is `MPI_Allgatherv`. Therefore, a detailed analysis of its performance is conducted, showing the stability of this MPI function on various HPC systems. An experiment was conducted to measure the cost of performing the same function 1.000 times for different sizes, which were communicating 100 KB, 10 MB, and 1 GB. Initially, the data are evenly distributed among the processes before the function is executed, while a copy of the concatenated data appears in each process after the function is completed.

Table 1 presents the results of the experiments conducted on the three HPC systems. Each column denotes the average times (`AVG`) and standard deviation (`STD`) for data sizes of 100 KB, 10 MB, and 1 GB, with varying numbers of processes. The first conclusion of their analysis is that the communications in System_1 are more expensive than in the other two systems with similar behavior, probably because of the communication interface. For communications of data size 100 KB, System_1 and System_2 present worse stability than System_3 using multinode configurations (4, 6, and 8 nodes), System_1 being even worse for the other configurations

Table 1 Average time and standard deviation of `MPI_Allgatherv` for different data size and number of processes in three HPC systems

NP	100 KB		10 MB		1 GB	
	AVG(s)	STD	AVG(s)	STD	AVG(s)	STD
System_1						
2	2.11E−02	2.96E−03	2.79E+00	3.75E−02	3.79E+01	6.25E−01
10	4.58E−02	8.79E−03	6.95E+00	2.77E−01	7.28E+01	3.53E+00
20	1.38E−01	5.69E−02	8.85E+00	1.05E+00	9.67E+01	6.64E+00
40	4.96E−01	1.85E−01	1.26E+01	7.23E−01	1.25E+02	5.34E+00
80	1.13E+00	3.42E−01	1.56E+01	1.56E+00	1.59E+02	9.28E+00
120	1.99E+00	2.09E−01	1.87E+01	2.21E+00	1.86E+02	1.15E+01
160	2.84E+00	6.15E−01	1.97E+01	1.33E+00	2.16E+02	9.25E+00
System_2						
2	1.04E−02	8.41E−05	1.62E+00	1.61E−02	2.62E+01	1.36E+00
10	3.35E−02	2.55E−04	2.74E+00	3.21E−02	3.23E+01	1.41E−01
20	4.73E−02	4.24E−04	4.26E+00	3.47E−02	5.69E+01	4.67E−02
40	1.34E−01	4.36E−04	5.83E+00	1.57E−01	6.58E+01	1.14E−01
80	2.29E−01	7.36E−02	5.54E+00	6.98E−02	6.57E+01	2.76E−01
120	3.65E−01	4.13E−02	7.71E+00	3.75E−01	6.55E+01	3.74E−01
160	3.73E−01	7.01E−02	7.72E+00	7.35E−01	6.55E+01	2.85E−01
System_3						
2	1.04E−02	1.01E−04	1.88E+00	9.38E−03	2.23E+01	1.63E−01
16	2.47E−02	6.53E−05	2.80E+00	2.12E−02	3.14E+01	5.42E−02
32	3.72E−02	9.39E−05	4.58E+00	3.44E−02	5.78E+01	7.12E−02
64	8.91E−02	4.92E−04	5.09E+00	2.31E−02	6.42E+01	1.46E−01
128	3.15E−01	2.56E−03	6.30E+00	2.97E−01	6.49E+01	9.06E−01
192	1.39E−01	1.14E−03	7.29E+00	1.33E−01	6.49E+01	5.02E−01
256	1.31E−01	9.92E−04	7.82E+00	1.51E−01	6.43E+01	3.62E−01

(2 processes, half, one, and two nodes). Additionally, in System_2 and System_3, the average values for large number of processes (2, 4, 6, and 8 nodes) and big communications (1 GB) seem independent of the number of processes, but this situation never happens in System_1.

The next step in evaluating `MPI_Allgatherv` is to compare AVG values in Table 1 with respect to similar values obtained in SAM. To achieve this, a configuration file is parameterized for each tuple (amount of bytes, HPC system, and number of processes) to emulate 1000 executions of the communication step.

Table 2 shows the average growth rate of SAM with respect to the original execution (column `AVG(%)`), as well as the standard deviation of the quotient between the two times (column STD). The first conclusion is that the execution of the tool in System_3 is the most stable, since all AVG values are close to 100% and STD is small enough. For System_1 and System_2 with small size communications (100 KB), there are many cases in which the simulation is unstable, and this behavior is

Table 2 Comparison between Table 1 and synthetic application

NP	100 KB		10 MB		1 GB	
	AVG (%)	STDQ	AVG (%)	STDQ	AVG (%)	STDQ
System_1						
2	105.60	1.51E−01	97.37	1.86E−02	98.55	1.83E−02
10	115.34	2.09E−01	104.83	6.31E−02	97.01	8.26E−02
20	80.08	4.80E−01	98.73	7.68E−02	97.88	3.11E−02
40	84.10	3.93E−01	98.90	1.10E−01	99.05	2.86E−02
80	94.10	2.68E−01	94.63	8.33E−02	102.10	4.14E−02
120	73.12	1.45E−01	95.37	5.99E−02	98.74	3.26E−02
160	93.98	2.35E−01	98.02	6.94E−02	97.85	3.73E−02
System_2						
2	113.15	2.23E−02	100.60	1.21E−02	98.88	2.46E−02
10	105.19	6.10E−03	99.13	1.61E−02	100.26	5.21E−03
20	101.15	4.08E−03	99.95	8.52E−03	100.05	1.34E−03
40	101.55	6.32E−03	96.98	3.12E−02	117.63	1.16E−01
80	80.81	4.99E−02	106.92	1.15E−01	103.59	6.07E−02
120	74.24	9.55E−02	84.42	5.49E−02	100.41	1.12E−02
160	78.54	1.01E−01	89.66	4.70E−02	106.25	7.64E−02
System_3						
2	111.45	3.71E−02	101.44	1.79E−02	101.07	7.47E−03
16	103.65	3.67E−03	99.65	5.02E−03	99.99	2.25E−03
32	104.15	3.89E−03	99.24	6.63E−03	102.25	1.64E−02
64	103.38	3.48E−02	100.66	2.94E−03	100.46	5.94E−03
128	35.09	3.76E−03	105.61	5.25E−02	100.91	2.85E−02
192	101.37	1.03E−02	101.14	1.67E−02	100.53	3.22E−03
256	101.45	7.90E−03	101.67	8.36E−03	102.18	2.54E−02

AVG (%) shows the growth rate of synthetic application, whereas STDQ is the standard deviation of the quotient

slightly maintained for medium sizes (10 MB), whereas the emulations are almost stable for big sizes (1 GB). Moreover, in general, STD values around 1E−01 identify abnormal AVG values, where `MPI_Allgatherv` emulations produce widely scattered values. There are other outliers, like communication of 100 KB using 128 processors in System_3, whose AVG value in Table 1 is too large, and therefore, the corresponding AVG value in Table 2 is too small, although STD values in both cases show that executions have been stable.

Lastly, the parallel CG execution time and the SAM emulations are evaluated, limiting the maximum number of iterations for each execution to 1000. Figure 8 shows the growth rate of SAM with respect to the parallel CG, considering only their average values. As previously mentioned, the growth rate is calculated by dividing the execution times of the two CG implementations using the same system and sparse matrix. The first conclusion is that, in general, the emulations in

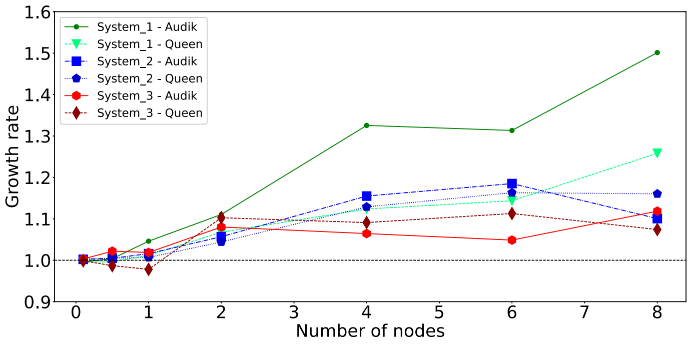


Fig. 8 Growth rate of the execution time of SAM with respect to the parallel CG for each sparse matrix and HPC system

System_3 are more stable than in the other HPC systems, being System_1 where the instability is greater. It makes sense since the emulations of the communications in System_3 are more accurate, and less in System_1, than in the rest. Additionally, the emulations using Queen_4147 are usually more stable than using audikw_1 in all HPC systems, because the latter one generates an unbalanced workload.

5.4 Emulation of a malleable application

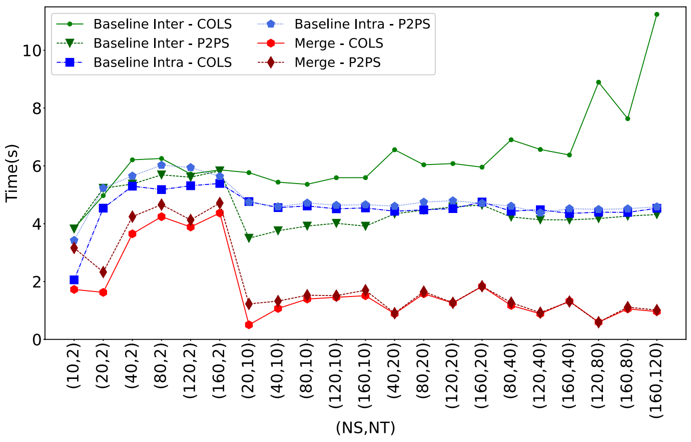
This subsection is divided into three main parts: Firstly, the different implementations of reconfigurations in MaM are analyzed in order to identify malfunctions. Next, the reconfiguration time of a malleable parallel CG using MaM is evaluated. Finally, the total execution time of the application is evaluated and how it is affected by malleability. The last two studies compare the malleable parallel CG with its emulation using Proteo. It is important to note that these experiments were only conducted on System_1, as the other two systems did not support process spawning during execution.

5.4.1 Evaluation of the reconfiguration techniques in isolation

In Sect. 4.2, are described several methods related to process management and other related to data distribution. For the former, we consider Baseline using inter-communicators, Baseline using intra-communicators and Merge, while for redistribution, the use of Point-to-Point or Collectives is presented. We now evaluate all these methods when a reconfiguration occurs, redistributing 3.947 GB of memory.

Figure 9 shows the reconfiguration time, in seconds, for the synchronous methods when processes are either shrunk (top) or expanded (bottom) during the application reconfiguration. This time is measured from the start of process spawning in the sources until the data have been fully received in the targets (data redistribution has ended). In both plots, the reconfigurations based on Merge method always outperform any Baseline combination. The main reason for this is that the total

(a) Shrinkage



(b) Expansion

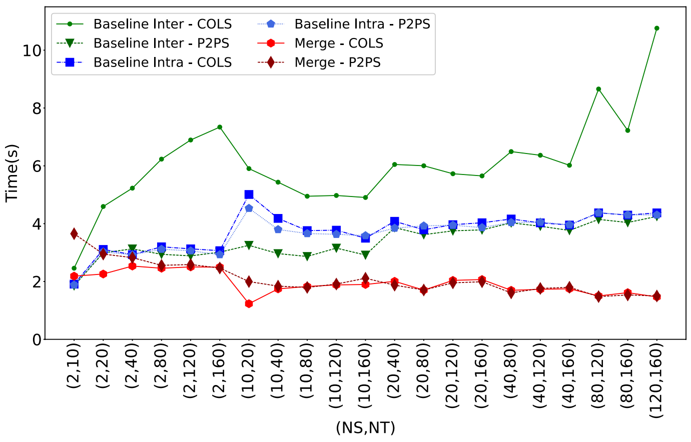


Fig. 9 Reconfiguration times for synchronous methods and combinations

number of processes handled by the Merge method is always lower. This becomes even more noticeable when there is oversubscription, i. e., there are more processes existing than cores available in a node.

In general, the performance differences between the Merge and Baseline executions can be more than 2 s for shrinking and slightly less for expanding. However, this is not the case for "Baseline Inter-COLS" which is much more expensive than the others. This combination redistributes data synchronously using the `MPI_Alltoallv` function on inter-communicators, which is implemented by MPICH 4.0.3 using the `PairWise Exchange` algorithm [57]. This algorithm generates a serialized synchronous communication based on the `MPI_Sendrecv` function, which communicates data from a source to a target, generating a lot of latency between the processes, since a process cannot start a new communication until the previous one has finished.

In conclusion, the use of the `MPI_Alltoallv` function on inter-communicators with MPICH is not recommended and therefore will not be considered Baseline using inter-communicator to spawn processes in the rest of this subsection.

5.4.2 Evaluation of the reconfiguration in a malleable application

The first analysis compares the behavior of the parallel CG and the emulated CG created using SAM when a reconfiguration is performed. To achieve this objective, is measured the reconfiguration time from the start of the process management until the data are fully received in the targets. The corresponding tasks define the malleability Stages 2 and 3, which are both included in MaM. To determine Proteo's ability to emulate a malleable application, the parallel CG also relies on MaM to be reconfigured. The solution was to include an interface to MaM, which can be used by real applications to convert them to malleable easily.

Figure 10 shows the reconfiguration time for the malleable parallel CG using all the described combinations and different numbers of sources and targets for shrinking (top) and expansion (bottom). Note that suffixes in data redistribution methods define the strategy used: no strategy or synchronous (S), non-blocking MPI functions with wait sources (A), and managing threads (T). To minimize the overhead of malleability, it is important to obtain small values for the reconfiguration time.

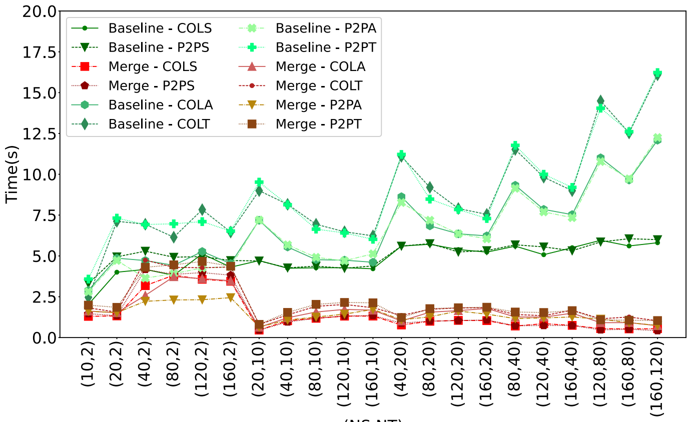
The first conclusion is that Merge methods always perform better than all Baseline methods. In fact, the reconfiguration times for all Merge combinations are very similar and always less than 3 s. This promotes the use of data redistribution strategies, where reconfiguration and application execution overlap, since the overhead of these strategies is close to zero.

Reconfiguration times for Baseline methods are higher than for Merge methods, mainly due to the impact of the oversubscription in the former, and a similar situation occurs with the use of data redistribution strategies on Baseline methods. Thus, no strategy is faster than the use of non-blocking MPI functions, since these functions work slowly when the application is executing. Moreover, using non-blocking MPI functions is faster than managing threads, since the latter increases the oversubscription problem. These comments are easier to observe in the expansion plot than in the shrinkage plot.

Figure 11 compares the reconfiguration time obtained for the malleable versions of the parallel CG and the emulated CG created using SAM. With this objective, the quotient of the reconfiguration times of the two CG implementations using the same combination is computed and shown in the figure.

The analysis of this figure is more complex, but there are some interesting trends to note. In general, most of the values are in the interval [0.75, 1.25], with Baseline combinations are usually greater than 1 and Merge combinations lower than 1. This different trend is also related to the effect of oversubscription.

(a) Shrinkage



(b) Expansion

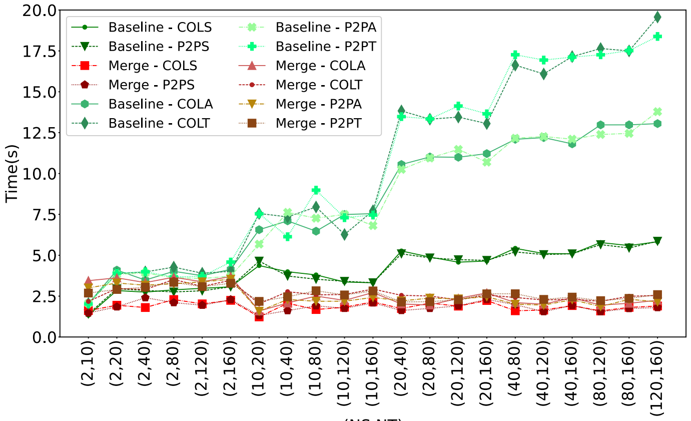
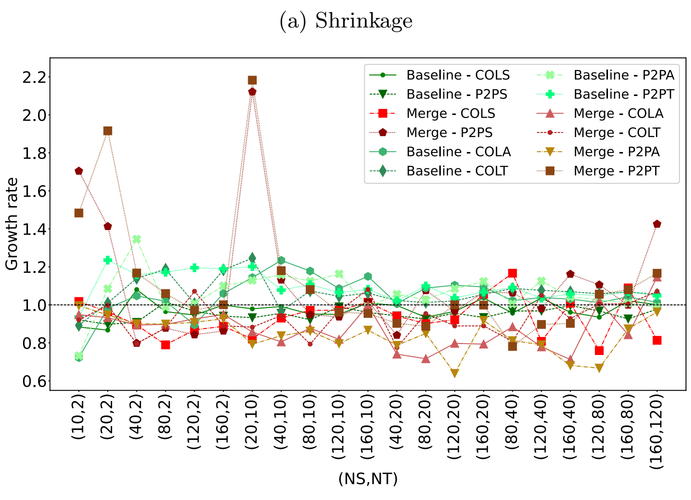


Fig. 10 Reconfiguration times for the malleable parallel CG

Moreover, the use of strategies in data redistribution grows the value of the quotient, showing the complexity of the overlapping of reconfiguration and the execution of SAM. It should be noted, however, that with the exception of the Asynchronous Baselines alternatives, most of the reconfiguration times are less than 5 s (Fig. 10), so the percentile differences are easy to identify.

Further studies will be needed to analyze some of the anomalous results of the P2PS and P2PT Merge combinations for both expansion and contraction.



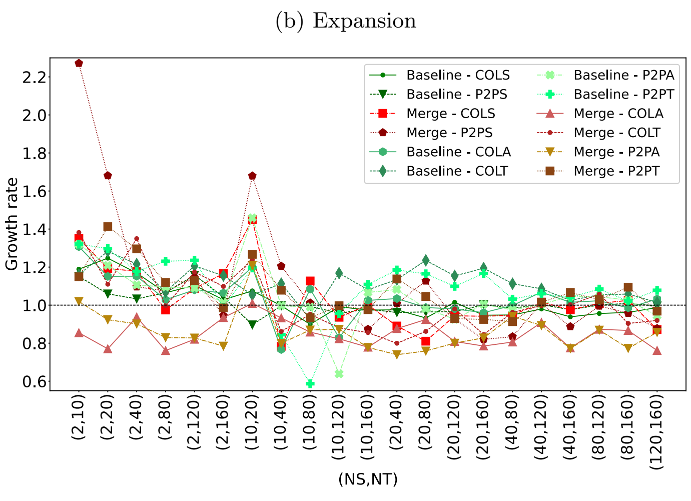


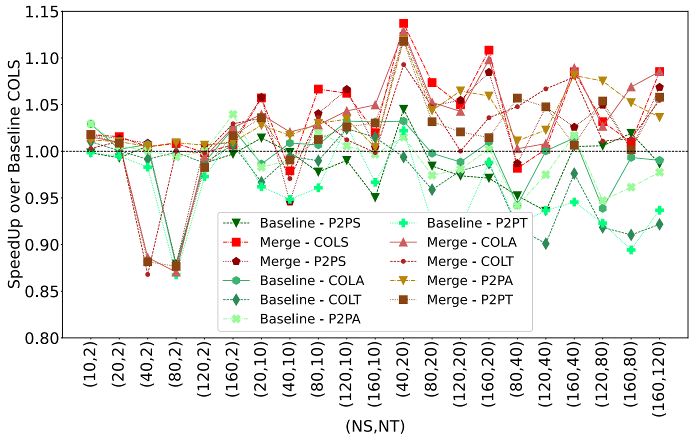
Fig. 11 Reconfiguration times difference of all reconfiguration methods between the malleable versions of the parallel CG and the emulated CG using SAM

5.4.3 Evaluation of a malleable emulated application

The final analysis compares the behavior of the malleable version of the parallel CG and the emulated CG created using SAM, when the complete execution is performed, including the reconfiguration. To ensure a fair comparison, both use MaM to complete the reconfiguration. The main goal is to conclude the impact on the performance of the different reconfiguration combinations defined in the paper.

Figure 12 compares the different reconfiguration combinations on malleable parallel CG, showing the speedup over a reference combination, Baseline-COLS, which uses the Baseline method for spawning processes and collective MPI functions

(a) Shrinkage



(b) Expansion

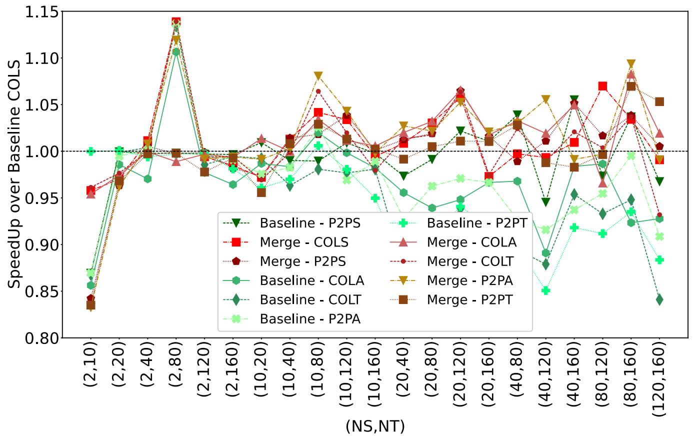
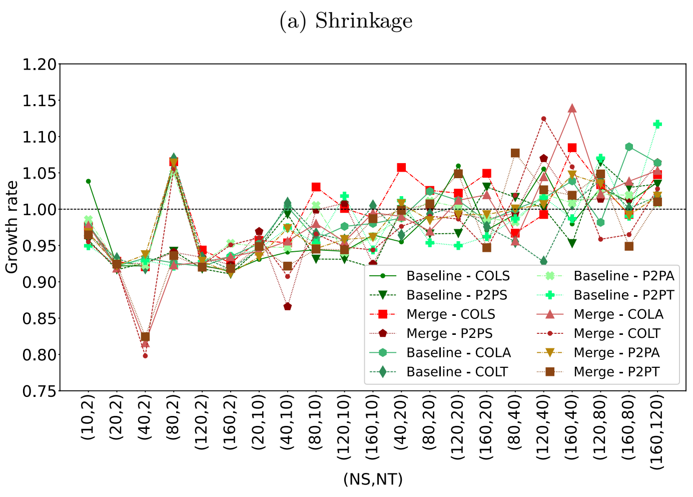


Fig. 12 Speedup with respect to Baseline-COLS of the other combinations, when executing malleable parallel CG with a reconfiguration

for data redistribution. In this way, values greater than 1 mark combinations that are faster than the reference, and values lower than 1 identify combinations that are slower than the reference.

Analyzing the plots, we can see that there are no major differences between the combinations, since all values are in the interval [0.85, 1.15], although many values are closer to 1. These latter values are more common in Merge combinations for the expansion. In addition, there are many extreme values from/to two processes, which may be caused by the congestion of messages sent/received by these two processes.

In general, Merge combinations are faster than the reference, while Baseline combinations are slower, and even slower if threads are managed. The behavior



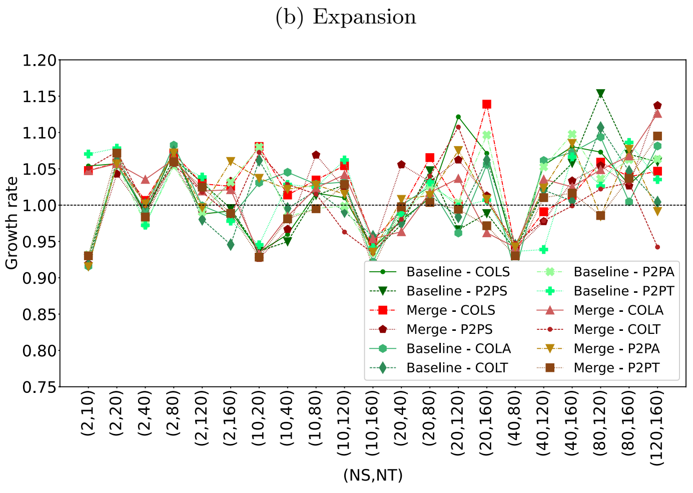


Fig. 13 The ratio of the execution times of all reconfiguration methods between the malleable versions of the parallel CG and the emulated CG created using SAM

of Baseline combinations may be due to the oversubscription, which is closely related to this process management.

Figure 13 compares the execution time obtained for the malleable versions of the parallel CG and the emulated CG created using SAM. With this objective, the quotient of the execution times of the two CG implementations using the same reconfiguration combination is computed and shown in the figure.

Analyzing the figure, we can see that most of the values are in the interval [0.90, 1.15], showing a good correspondence between the two CG implementations. There are only a few Merge combinations that show lower values for

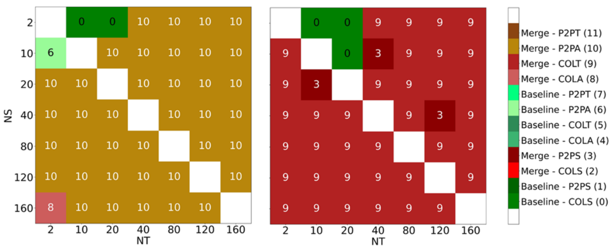


Fig. 14 Preferred methods when reconfiguring to reduce execution time depending on the number of NS and NT, malleable parallel CG (left), and malleable emulated CG using SAM (right)

shrinkage, corresponding to cases from two nodes (40 cores) to one node (2 and 10 cores). Further analysis will be required to justify these low values.

There is a trend toward higher values when more targets are involved in a shrinkage, but there is no clear trend for expansion; only that higher values occur when more sources and targets are involved. In the future, it would be interesting to see whether this increasing trend for shrinkage is maintained when using a larger number of nodes. Instead, there is no clear difference between the Baseline and Merge combinations, perhaps because MaM is used for both CG implementations.

Figure 14 shows graphically the best method for each pair (NS sources, NT targets), when the malleable versions of the parallel CG (left) and emulated CG created using SAM (right) are executed. The name of the axes, vertical for NS and horizontal for NT, determines that the upper triangular part of the matrix is related to expansion, while the lower part refers to shrinkage. Moreover, the number in each cell, along with the color, identifies the fastest method for each pair according to the Kruskal–Wallis and the Post-hoc Conover tests. If there is a tie in one cell, the nearest cells are checked, and the method with the highest number of occurrences is selected.

The first conclusion is that both figures are quite homogeneous, but the color in each one is not the same. For malleable parallel CG, Merge-P2PA appears in 38 out of 42 cells, while for malleable emulated CG, Merge-COLT appears in 36 out of 42 cells. However, this is not a relevant difference, since a broader statistical analysis shows that Merge-COLT is equivalent to Merge-P2PA in 33 out of 42 cells for malleable parallel CG, and Merge-P2PA is equivalent to Merge-COLT in 35 out of 42 cells for malleable emulated CG. Therefore, the best combination of two malleable CG implementations should use Merge for process management, and an asynchronous alternative, COLT or P2PA, for data redistribution.

But there are cells whose colors are not the most common in each plot. Most of these values occur when only one node is involved in the reconfiguration, that is, when NS and NT are not more than 20. For malleable parallel CG, the three corresponding cases prefer Baseline combinations, two using COLS and one using P2PA for data redistribution. Instead, there are four cases for malleable emulated CG, which differ in the process management method: three use Baseline for expansion, and only one uses Merge for shrinkage, but all of them use COLS for data redistribution. The other non-homogeneous cells, one Merge-COLA for malleable parallel

CG and two Merge-P2PS for malleable emulated CG, are statistically equivalent to the most common combinations.

6 Conclusions

This work presents Proteo, a framework for the generation and evaluation of malleable applications, both synthetic and real. This tool has been developed using a modular structure, with the two main independent components being SAM and MaM. SAM is used to emulate the computational behavior of iterative MPI applications, taking into account different computation and communication operations. MaM provides the ability to reconfigure an application during execution, expanding, or shrinking the number of assigned processes and redistributing data between source and target processes. Both modules include a Monitoring submodule that measures the performance of the emulation and malleability steps and writes them to an output file for further analysis. This flexible tool makes it possible to analyze the performance of emulating an iterative application when using SAM alone, of emulating a malleable iterative application when using both SAM and MaM, or of incorporating malleability into a real application when using MaM alone.

The behavior of the different emulations is defined in a configuration file that stores parameters related to SAM and MaM. For the former, the number of application stages, along with the type and duration of each stage, should be determined using tracing tools. We have used Extrae and Paraver in this work, although other tools could be used. The parameters of the configuration file associated with MaM simulate the RMS requirements, setting when reconfiguration occurs, the number of target processes and the theoretical parallel performance of the emulated application.

Two studies have been conducted to analyze the ability of Proteo to emulate both non-malleable and malleable applications. In these studies, parallel CG was the application to be emulated, and different HPC systems and sparse matrices were used.

In the first study, SAM is used to emulate the non-malleable parallel CG on three systems for *audikw_1* and *Queen_4147* sparse matrices. The main conclusion is that Proteo is able to emulate the behavior of applications on different systems, demonstrating its versatility. The percentile differences for emulating balanced workloads are less than 20%, while these differences are around 50% for unbalanced workloads. This makes sense as Proteo currently only considers balanced workloads.

The second study compares the performance of MaM when used in combination with SAM to emulate the execution of a malleable real application, in this case, parallel CG, which also uses MaM for reconfiguration. The study initially analyzed all combinations of process management, and data redistribution alternatives were analyzed. It concluded that the use of collective MPI functions on inter-communicators to redistribute data degrades performance. The study then goes on to examine the reconfiguration and execution times for the malleable parallel CG and its emulation and shows that the differences between the two implementations are less than 15%, except in one case. It also suggests that Merge method is commonly the best method for process management, as there is no oversubscription.

Thus, this work demonstrates that Proteo can reliably emulate malleable applications. These emulations also allow the evaluation of system behavior when malleability is applied, assessing job performance and system productivity prior to its deployment on a production system. Additionally, the use of MaM facilitates the development of malleable versions of real applications.

Future work will focus on completing both modules of Proteo, enabling more accurate emulation of real applications and more efficient execution of reconfigurations. In this way, Proteo can be used to emulate synthetic workloads composed of a mix of malleable and non-malleable applications, allowing the performance of HPC systems to be evaluated when deploying different workloads.

To improve the emulation of real-world applications using SAM, additional mechanisms will be explored, focusing on *I/O* operations and a broader range of MPI communication patterns. Also, new parameters will be defined in configuration files to specify the unbalanced behavior of workloads, which are more common in real HPC systems than balanced workloads. In addition, SAM will be fully tuned to improve the emulation of non-iterative applications, which are typically found in real workloads. Essentially, the emulation of an application will be broken down into phases, each of which consists of several stages that can be repeated one or more times. The evaluation of these improvements in SAM will be tested by modeling a number of more demanding applications, both iterative and non-iterative applications.

In MaM, the focus will primarily be on enhancing existing reconfiguration strategies and defining new ones, with particular emphasis into minimize communication costs during data redistribution. Thus, an alternative implementation of Merge will be analyzed in which source processes that will become targets hold as much of their original data as possible. Additionally, a new redistribution method based on MPI RMA will be implemented so that source processes do not need to collaborate explicitly during data redistribution. Furthermore, a novel approach will be explored for parallel spawning, which has the potential to decrease time spent on spawning as well as provide benefits for additional reconfigurations during execution.

Another area of future work is the use of Proteo within DMR [58]. This combination will allow Proteo to avoid the need to emulate the amount of resources used, as DMR is capable of exchanging information with Slurm and modifying the resources used by a job with different policies. In addition, this collaboration will facilitate the utilization of diverse reconfiguration techniques implemented in Proteo, thereby extending the capabilities of DMR.

Author contributions Author I. Martín-Álvarez undertook the development, execution, and data analysis of the experiments, guided by the supervision of Maribel Castillo and José I. Aliaga. All authors collaborated in drafting and reviewing the final manuscript.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. This work has been funded by project PID2020-113656RB-C21 supported by MCIN/

AEI/10.13039/501100011033. Researcher I. Martín-Álvarez was supported by the predoctoral fellowship ACIF/2021/260 from Valencian Region Government and European Social Funds. The European DEEP-SEA project has received funding from the European Commission's EuroHPC-JU program under grant agreement No. 955606; the JU receives support from the European Union's Horizon 2020 research and innovation programme and by the Spanish Ministry under Grant PCI2021-121958 funded by MCIN / AEI / 10.13039 / 501100011033 and by the "European Union NextGenerationEU/PRTR."

Availability of data and materials All data used to create the tables and figures presented in the Results section are available at [52].

Code availability The codes are available at [50, 51] in a specific branch for this publication.

Declarations

Conflict of interest The authors have no conflict of interest to declare that is relevant to the content of this article.

Ethical approval Not applicable.

Consent to participate Not applicable.

Consent for publication All authors read and approved the final manuscript.

References

1. Dongarra J, Luszczek P (2011) In: Padua D (ed) TOP500. Springer, Boston, pp 2055–2057. https://doi.org/10.1007/978-0-387-09766-4_157
2. Message Passing Interface Forum: MPI: a message-passing interface standard version 4.1. (2023). https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf
3. Hori A, Yoshinaga K, Herault T, Bouteiller A, Bosilca G, Ishikawa Y (2020) Overhead of using spare nodes. Int J High Perform Comput Appl 34(2):208–226. https://doi.org/10.1177/1094342020901885
4. Feitelson DG (1996) Packing schemes for gang scheduling. Lecture notes in computer science book series (LNCS), vol 1162. Springer, Heidelberg, pp 89–110
5. Bernholdt DE, Boehm S, Bosilca G, Gorentla Venkata M, Grant RE, Naughton T, Pritchard HP, Schulz M, Vallee GR (2020) A survey of MPI usage in the US exascale computing project. Concurr Comput Pract Exp 32(3):4851. https://doi.org/10.1002/cpe.4851
6. Iserte S, Mayo R, Quintana-Orti ES, Pena AJ (2020) DMRlib: easy-coding and efficient resource management for job malleability. IEEE Trans Comput. https://doi.org/10.1109/TC.2020.3022933
7. Posner J, Fohry C (2021) Transparent resource elasticity for task-based cluster environments with work stealing. In: 50th International Conference on Parallel Processing Workshop. ICPP Workshops '21. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3458744.3473361
8. Eberius D, Rahman MW-U-, Ozog D (2023) Evaluating the potential of elastic jobs in HPC systems. In: Proceedings of the SC '23 Workshops of The International Conference on High Performance

Computing, Network, Storage, and Analysis. SC-W '23. Association for Computing Machinery, New York, NY, USA, pp 1324–1333. https://doi.org/10.1145/3624062.3624199
9. Iserte S, Rojek K (2019) An study of the effect of process malleability in the energy efficiency on GPU-based clusters. J Supercomput. https://doi.org/10.1007/s11227-019-03034-x
10. Rodríguez-Gonzalo M, Singh DE, Blas JG, Carretero J (2016) Improving the energy efficiency of MPI applications by means of malleability. In: Proceedings—24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016. Institute of Electrical and Electronics Engineers Inc., Heraklion, Greece, pp 627–634. https://doi.org/10.1109/PDP.2016.98
11. Alberto C, Alvaro A, Javier G-B, Jesus C, Singh DE (2023) Malleable techniques and resource scheduling to improve energy efficiency in parallel applications. High performance computing, vol 13999. Springer, Hamburg, pp 16–27
12. Aliaga JI, Castillo M, Iserte S, Martín-Álvarez I, Mayo R (2022) A survey on malleability solutions for high-performance distributed computing. Appl Sci 12(10):5231. https://doi.org/10.3390/app12105231
13. Sudarsan R, Ribbens CJ (2007) ReSHAPE: a framework for dynamic resizing and scheduling of homogeneous applications in a parallel environment. In: International Conference on Parallel Processing
14. Comprés I, Mo-Hellenbrand A, Gerndt M, Bungartz HJ (2016) Infrastructure and API extensions for elastic execution of MPI applications. In: ACM International Conference Proceeding Series, vol 25-28-Sep. ACM Press, New York, New York, USA, pp 82–97
15. Prabhakaran S, Neumann M, Rinke S, Wolf F, Gupta A, Kale LV (2015) A batch system with efficient adaptive scheduling for malleable and evolving applications. In: 2015 IEEE International Parallel and Distributed Processing Symposium, pp 429–438. https://doi.org/10.1109/IPDPS.2015.34
16. Martín G, Marinescu M-C, Singh DE, Carretero J (2013) FLEX-MPI: an MPI extension for supporting dynamic load balancing on heterogeneous non-dedicated systems. In: Euro-Par Parallel Processing, pp 138–149
17. Yoo AB, Jette MA, Grondona M (2003) Slurm: simple linux utility for resource management. In: Feitelson D, Rudolph L, Schwiegelshohn U (eds) Job scheduling strategies for parallel processing. Springer, Berlin, Heidelberg, pp 44–60
18. Tadepalli S (2003) Gems: a fault tolerant grid job management system. Master's thesis, Virginia Polytechnic Institute
19. Huang C, Zheng G, Kumar S, Kalé LV (2006) Performance evaluation of adaptive MPI. In: Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006
20. Kale LV, Krishnan S (1993) CHARM++: a portable concurrent object oriented system based on C++. ACM SIGPLAN Not 28(10):91–108. https://doi.org/10.1145/167962.165874
21. MPICH development team: MPICH Website. https://www.mpich.org/
22. Fecht J, Schreiber M, Schulz M, Pritchard H, Holmes DJ (2022) An emulation layer for dynamic resources with MPI sessions. In: HPCMALL 2022—Malleability Techniques Applications in High-Performance Computing, Hambourg, Germany. https://hal.science/hal-03856702
23. Huber D, Streubel M, Comprés I, Schulz M, Schreiber M, Pritchard H (2022) Towards dynamic resource management with MPI sessions and PMIx. In: Proceedings of the 29th European MPI Users' Group Meeting. EuroMPI/USA '22. Association for Computing Machinery, New York, NY, USA, pp 57–67. https://doi.org/10.1145/3555819.3555856
24. Iserte S, Mayo R, Quintana-Ortí ES, Beltran V, Peña AJ (2017) Efficient scalable computing through flexible applications and adaptive workloads. In: 10th International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2), Bristol
25. Sudarsan R, Ribbens CJ (2009) Scheduling resizable parallel applications. In: International Symposium on Parallel & Distributed Processing. IEEE, Rome, Italy, pp 1–10. https://doi.org/10.1109/IPDPS.2009.5161077
26. Martín G, Singh DE, Marinescu M-C, Carretero J (2015) Enhancing the performance of malleable MPI applications by using performance-aware dynamic reconfiguration. Parallel Comput 46:60–77
27. Sudarsan R, Ribbens CJ, Farkas D (2009) Dynamic resizing of parallel scientific simulations: a case study using LAMMPS. In: International Conference on Computational Science (ICCS)
28. Wong AT, Oliker L, Kramer WT, Kaltz TL, Bailey DH (2000) ESP: a system utilization benchmark. In: Supercomputing, ACM/IEEE 2000 Conference. IEEE, pp 1–12
29. Houzeaux G, Badia RM, Borrell R, Dosimont D, Ejarque J, Garcia-Gasulla M, López V (2021) Dynamic resource allocation for efficient parallel CFD simulations. Technical report, Barcelona Supercomputing Center (December)

30. Iserte S, Martínez H, Barrachina S, Castillo M, Mayo R, Peña AJ (2018) Dynamic reconfiguration of noniterative scientific applications. Int J High Perform Comput Appl. https://doi.org/10.1177/1094342018802347
31. Mo-Hellenbrand A, Comprés I, Meister O, Bungartz H-J, Gerndt M, Bader M (2017) A large-scale malleable tsunami simulation realized on an elastic MPI infrastructure. In: CF'17. Association for Computing Machinery, New York, NY, USA, pp 271–274. https://doi.org/10.1145/3075564.3075585
32. Carretero J, Exposito D, Cascajo A, Montella R (2023) Malleability techniques for HPC systems, pp 77–88. https://doi.org/10.1007/978-3-031-30445-3_7
33. Genaro S-G, Garcia-Blas J, Cosmin P, Jesus C (2023) Malleable and adaptive ad-hoc file system for data intensive workloads in HPC applications. Springer, Hamburg, pp 56–67
34. Duran A, Ayguadé E, Badia RM, Labarta J, Martinell L, Martorell X, Planas J (2011) Ompss: a proposal for programming heterogeneous multi-core architectures. Parallel Process Lett 21:173–193. https://doi.org/10.1142/S0129626411000151
35. Badia RM, Conejero J, Diaz C, Ejarque J, Lezzi D, Lordan F, Ramon-Cortes C, Sirvent R (2015) COMP superscalar, an interoperable programming framework. SoftwareX 3–4:32–36. https://doi.org/10.1016/j.softx.2015.10.004
36. Castelló A, Catalán S, Igual FD, Quintana-Ortí ES, Rodríguez-Sánchez R (2023) QR factorization using malleable BLAS on multicore processors. In: High performance computing. ISC High Performance 2022 International Workshops: Hamburg, Germany, May 29–June 2, 2022, Revised Selected Papers. Springer, Berlin, Heidelberg, pp. 176–189. https://doi.org/10.1007/978-3-031-23220-6_12
37. D'Amico M, Garcia-Gasulla M, López V, Jokanovic A, Sirvent R, Corbalan J (2018) DROM: enabling efficient and effortless malleability for resource managers. In: Proceedings of the 47th International Conference on Parallel Processing Companion. ICPP '18. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3229710.3229752
38. Lina DH, Ghafoor S, Hines T (2023) Scheduling of elastic message passing applications on HPC systems. In: Klusáček D, Julita C, Rodrigo GP (eds) Job scheduling strategies for parallel processing. Springer, Cham, pp 172–191
39. Özden T, Beringer T, Mazaheri A, Fard HM, Wolf F (2022) ElastiSim: a batch-system simulator for malleable workloads. In: International Conference on Parallel Processing (ICPP '22). https://doi.org/10.1145/3545008.3545046
40. D'Amico M, Jokanovic A, Corbalan J (2019) Holistic slowdown driven scheduling and resource management for malleable jobs. In: Proceedings of the 48th International Conference on Parallel Processing. ICPP '19. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3337821.3337909
41. Casanova H, Quinson M, Legrand A (2008) SimGrid: a generic framework for large-scale distributed experiments. IEEE Computer Society, Washington, DC, USA, pp 126–131. https://doi.org/10.1109/UKSIM.2008.28. https://doi.ieeecomputersociety.org/10.1109/UKSIM.2008.28
42. Dutot P-F, Mercier M, Poquet M, Richard O (2017) Batsim: a realistic language-independent resources and jobs management systems simulator. In: Desai N, Cirne W (eds) Job scheduling strategies for parallel processing. Springer, Cham, pp 178–197
43. Klusáček D, Soysal M, Suter F (2020) Alea—complex job scheduling simulator. In: Wyrzykowski R, Deelman E, Dongarra J, Karczewski K (eds) Parallel processing and applied mathematics. Springer, Cham, pp 217–229
44. Cascajo A, Singh DE, Carretero J (2022) Detecting interference between applications and improving the scheduling using malleable application proxies. Springer, Heidelberg, Germany
45. Cascajo A, Singh DE, Carretero J (2021) LIMITLESS—LIght-weight MonItoring tool for LargE scale systems. In: 2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp 220–227. https://doi.org/10.1109/PDP52278.2021.00042
46. Sudarsan R, Ribbens CJ (2016) Combining performance and priority for scheduling resizable parallel applications. J Parallel Distrib Comput 87:55–66
47. Martín-Álvarez I, Aliaga JI, Castillo M, Iserte S, Mayo R (2024) Dynamic spawning of MPI processes applied to malleability. Int J High Perform Comput Appl. https://doi.org/10.1177/10943420231176527
48. Martín Álvarez I, Aliaga JI, Castillo M, Iserte S (2023) Efficient data redistribution for malleable applications. In: Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis. SC-W '23. Association for Computing Machinery, New York, NY, USA, pp 416–426. https://doi.org/10.1145/3624062.3624110

49. Martín-Álvarez I, Aliaga JI, Castillo M, Iserte S (December 2022) Malleable synthetic tool manual. Technical report, Universitat Jaume I
50. Martín-Álvarez I (2023) Proteo Code—Branch Journal of Supercomputing. https://lorca.act.uji.es/gitlab/martini/malleability_benchmark/-/tree/JournalSupercomputing23/24
51. Martín-Álvarez I (2023) Malleable Conjugate Gradient Code—Branch Journal of Supercomputing. https://lorca.act.uji.es/gitlab/martini/malleable_cg/-/tree/JournalSupercomputing23/24
52. Martín-Álvarez Iker (2023) Proteo Dataset (2023) for Article Proteo: a framework for the generation and evaluation of malleable MPI applications. Zenodo. https://doi.org/10.5281/zenodo.10229558
53. Saad Y (2003) Iterative methods for sparse linear systems, 2nd edn. Society for Industrial and Applied Mathematics, Philadelphia. https://doi.org/10.1137/1.9780898718003
54. Shapiro SS, Wilk MB (1965) An analysis of variance test for normality (complete samples). Biometrika 52(3/4):591–611
55. Kruskal WH, Wallis WA (1952) Use of ranks in one-criterion variance analysis. J Am Stat Assoc 47(260):583–621
56. Conover WJ, Iman RL (February 1979) Multiple-comparisons procedures. Informal Report. Technical report, Los Alamos National Lab. https://doi.org/10.2172/6057803. https://www.osti.gov/biblio/6057803
57. Martín-Álvarez I, Aliaga JI, Castillo M, Iserte S (2023) Análisis de Métodos de Redistribución de Datos para Aplicaciones MPI Maleables. In: Avances en Arquitectura y Tecnología de Computadores, Jornadas Sartecto 23, pp 453–462. Zenodo, Facultad de Informática - UCM, Madrid, Spain. https://doi.org/10.5281/zenodo.8099552
58. Iserte S, Mayo R, Quintana-Ortí ES, Beltran V, Peña AJ (2018) DMR API: improving cluster productivity by turning applications into malleable. Parallel Comput 78:54–66. https://doi.org/10.1016/J.PARCO.2018.07.006