Efficient data redistribution for malleable applications

Iker Martín-Álvarez
martini@uji.es
Universitat Jaume I
Castellón, Spain

José I. Aliaga
aliaga@uji.es
Universitat Jaume I
Castellón, Spain

Maribel Castillo
castillo@uji.es
Universitat Jaume I
Castellón, Spain

Sergio Iserte
sergio.iserte@bsc.es
Barcelona Supercomputing Center
Barcelona, Spain

ABSTRACT

Process malleability can be defined as the ability of a distributed MPI parallel job to change the number of processes on–the–fly without stopping its execution, reallocating the compute resources originally assigned to the job, and without storing application data to disk. MPI malleability consists of four stages: resource reallocation, process management, data redistribution and execution resuming. Among them, data redistribution is the most time-consuming and determines the reconfiguration time. In this paper, we compare different implementations of this stage using point-to-point and collective MPI operations, and discuss the impact of overlapping computation-communication. We then combine these strategies with different methods to expand/shrink jobs, using a synthetic application to emulate MPI-based codes and their malleable counterparts, in order to evaluate the effect of different malleability methods in parallel distributed applications. The results show that the use of asynchronous techniques speeds up execution by 1.14 and 1.21, depending on the network used.

CCS CONCEPTS

• **Computing methodologies → Parallel programming languages**; **Concurrent programming languages**; **Modeling and simulation**; • **Networks → Network experimentation**.

KEYWORDS

MPI, Malleability, Data redistribution, Emulations

ACM Reference Format:
Iker Martín-Álvarez, José I. Aliaga, Maribel Castillo, and Sergio Iserte. 2023. Efficient data redistribution for malleable applications. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3624062.3624110

SC-W 2023, November 12–17, 2023, Denver, CO, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0785-8/23/11.
https://doi.org/10.1145/3624062.3624110

1 INTRODUCTION

High performance computing (HPC) facilities require novel programming techniques to take full advantage of the large number of processors interconnected by high-speed networks. A major goal of HPC, and exascale supercomputers in particular, is to use special techniques to maintain high system productivity in terms of completed jobs per unit of time.

Large computing facilities typically include Resource Management Systems (RMS), which are responsible for monitoring available resources and allocating them following users requirements. Therefore, when jobs request resources to the RMS, the petition may vary depending on whether the main goal is to complete the execution as fast as possible, or to improve productivity in the system. Usually, from the application's point of view, the shortest execution times occur when allocating the number of resources that maximize performance, even though not all of them are always used during execution. On the other hand, the highest system productivity can be achieved when all resources are used most of the time. To achieve both goals, RMS must allocate the optimal number of resources to each step of the job, reaching a trade-off between application performance and system productivity.

On–the–fly malleability allows applications to change the initial allocation of compute resources, while the job is running and without storing application data to disk. The benefits of its use can be analyzed from two different points of view. For each individual application, the benefit can come from the increase of its particular performance when the job gets more resources, while for the global system, the benefit can come from the increase in throughput with the reduction of the makespan.

In a parallel job, malleability is triggered at checkpoints, specific points where processes are synchronized throughout the execution. Defining checkpoints at the end of a loop can be the most straightforward option in iterative applications, while for non-iterative applications, a good alternative is to define a checkpoint at the beginning of each phase.

The first task in a malleability checkpoint is to contact the RMS to know if the application has to be reconfigured, since the RMS is in charge of making this decision. If a reconfiguration is proposed, the application has to perform the following tasks:

(1) *Stage 1: Resources reallocation.* RMS allocates new resources and/or relinquishes assigned already resources to/from a job.
(2) *Stage 2: Processes management.* Spawn/terminate processes accordingly to the RMS reconfiguration decision.

(3) *Stage* 3*: Data redistribution.* Communicate data among initial and final processes, so that the execution continues properly using the final processes.
(4) *Stage* 4*: Resuming execution.* Continue the execution at the same point as before the reconfiguration started.

This paper introduces different methodologies to complete Stage 3 of malleability, analyzing how it can be combined with a variety of methods and strategies for Stage 2 [16].

We consider that at each malleability step, the number of processes in a parallel job is changed from NS processes (sources) to NT processes (targets). Moreover, processes management can be performed in two ways: always spawning new NT targets, or creating/terminating a number of processes equal to the positive difference between NS and NT. In [16], these options are defined as *Baseline* and *Merge* methods, respectively.

In turn, the data redistribution considers two types of data: *Constant* (they do not vary during execution) and *Variable* (they are modified at each stage/iteration). Both types can be redistributed using *Point-to-Point* or *Collective* operations. Furthermore, the non-blocking version of the operations can be leveraged for redistributing constant data, overlapping computation and communication. In fact, overlapping techniques can be used to allow sources to continue the execution while Stage 2 and 3 are completed. In this regard, unlike *Synchronous* strategies where source processes halt their computations, *Asynchronous* strategies can increase performance of certain operations.

In this paper, we present different techniques for performing data redistribution. Then, Stage 2 is coupled with Stage 3 by evaluating six of the eight expand/shrink alternatives [16] combined with the presented data redistribution methods. All the combinations have been analyzed using a synthetic application, which allows studying the behaviour of malleable applications considering different scenarios [14, 17]. The main contributions of this paper are the following:

- Two methods for data redistribution, based on Point-to-Point and Collective communications, are presented.
- Strategies to overlap computation and communication are discussed: using *non-blocking* functions and *threading*.
- All these methods and strategies are combined with the alternatives to spawn/shrink processes described in [16].
- These combinations are evaluated using a synthetic application that emulates parallel malleable applications [14, 17].

The rest of this paper is organized as follows. Section 2 discusses related work in the area of malleability and dynamic spawn of processes in MPI applications. Section 3 describes the different methods and strategies to perform data redistribution. Section 4 shows the results obtained when a synthetic application is used, showing the best alternatives in different scenarios. Section 5 summarizes the paper and discusses future work.

2 BACKGROUND

MPI process malleability made its first steps taking advantage of checkpoint/restart (C/R) techniques based on the principle of storing the state of a job in a non-volatile memory device, in order to load it when required. In this regard, on-disk reconfiguration in malleability halts executions in order to resume them with a different shape. Processes are responsible to store and load the appropriate data depending on the number of processes [3, 4, 6, 12]

Traditional C/R solutions show a low performance because of the costly disk access when writing and reading. More modern malleability solutions rely on in-memory data redistribution. Dynamic in-memory mechanisms distribute data among processes without accessing the disk. Data are always stored in volatile memory which accelerates its manipulation.

Malleability frameworks that support in-memory data redistribution have developed different strategies to address this stage. In [8] data redistributions are categorized in three ways: *Automatic*, *Semi-automatic*, and *Manual*. Following, these categories are described together with some malleability tools that utilize those techniques with different implementations:

- *Automatic*, where the user data communication patterns among processes is not explicitly codified. For example, Flex-MPI [13] provides generalist automatic data redistribution using data structure registers, while DMR API [9] leverages data dependencies.
- *Semiautomatic* defines the methods that provide automatic data redistribution for specific scenarios. For instance, AMPI [7] is based on unique virtual memory addresses, while DMR-lib [10] provides predefined redistribution patterns.
- In the *manual* mode, users are fully responsible for defining the communication pattern among processes. Thus, coders directly implement data redistribution with point to point or collective functions. For example, Elastic MPI [1] does not provide any data redistribution assistance.

In this paper, the authors explore manual in-memory data redistribution since it is the most versatile and flexible approach to tackle any communication pattern in any application. Optimizing and reducing the time spent in this stage of the malleability is crucial to constraint reconfiguration overheads. Particularly, this paper explores the effect of asynchronous data redistribution during a reconfiguration operation. To the best of the authors knowledge, malleability data redistribution has not been previously studied in such a detail, for this reason, novel approaches for manual redistribution in malleability are presented and analyzed in this work.

3 DATA REDISTRIBUTION TECHNIQUES

Redistributing data is one of the most important stages in malleability since it is responsible for the major overhead.

The communications use the communicator that connects source and target processes and can be carried out in different ways: depending on the type of communication used (point-to-point or collective); and whether it is possible to overlap communication and computing to reduce the cost of this operation.

In malleability, data redistribution can not be taken as an isolated task, so the actions performed in Stage 2 should be taken into account together with Stage 3. In [16], eight different expansion/shrinkage alternatives are evaluated, always going from NS to NT, grouped into two main methods. *Baseline* method, which always spawns NT new processes, which will continue the execution as "targets", while all sources finalize. And *Merge* method, where some sources continue the execution after the reconfiguration: spawning only $NT - NS$ new processes and persisting NS

sources to expand, or stopping $NS - NT$ sources to shrink, while NT sources persist. The chosen MPI functions in [16] determine that *Baseline* method uses inter-communicators to redistribute data, while the *Merge* method generates intra-communicators.

Note that the processes which finalize their execution, they do so after data redistribution. In Baseline, all sources finalize when the targets have their data, whereas shrinking in Merge require halting some sources when the other sources (targets) have their data. For expansion in Merge, sources and new processes have to be synchronized before continuing with the execution.

Below, we describe the different communication methods which can be used to complete data redistribution and the different strategies analyzed to overlap computation and communication.

3.1 How to redistribute the data

In general, the communication pattern depends on the features of data to be redistributed and the algorithm to execute. The first consideration is that each source should inform the targets of the size of the data to be received, from which the targets can create the internal structures, and then the communication is completed by sending the data. The second comment is related to the content of the data to be redistributed, composed of vector and matrices, and distinguishing two types of matrices: dense and sparse.

When a static block data partition is used for dense data distribution, only the dimension of vectors and matrices is sufficient for sources and targets to calculate the size of the data to send/receive and the destination/origin of each chunk. In addition, most of the time, these values define the communication pattern of several objects. For sparse matrices, targets can not calculate from the matrix dimensions how many non-zeros elements will receive from each source. In this case, each source must calculate and send to each target the number of non-zeros it will receive before the redistribution begins. Note that the communication pattern need not to be complete, since the data communication between some sources and some targets can be empty.

The use of **Point-to-Point** MPI functions (P2P) is initially based on `MPI_Send` and `MPI_Recv`. Although both of them execute the *blocking* mode, deadlock is never reached when the intersection of sources and targets is null, that is, using the `Baseline` method. But using the `Merge` method, some processes can be sources and targets at the same time, and therefore deadlocks are possible. The use of the *buffered* mode of `MPI_Send` can solve in part this problem, but the safest solution is to use *non-blocking* MPI functions, combining `MPI_Isend`/`MPI_Irecv` with `MPI_Wait`/`MPI_Test` functions.

Algorithm 1 describes how these communications will be performed, using *myId* to identify the rank of each process: the sources are in $[frsSrc, lstSrc]$ and the targets are in $[frsTrgt, lstTrgt]$. The code for the sources involves a single loop in which each one sends the size and the values to each target via a pair of `MPI_Isend`, using different tags. But if a process is both source and target, a memcpy is used instead. The targets first execute a `MPI_Irecv` for each source, waiting for the sizes. When a size arrives, the internal structures are created and a new `MPI_Irecv` is executed, waiting for the values. In the targets, *numRcv* is initialized by the number of sources (first loop) and later informs on how many messages with values remain pending to be received (second loop). `MPI_Waitany` is used by the targets to get the source and the tag of the last received message, and determine the next behaviour. Finally, `MPI_Waitall` is used by the sources to ensure that all messages have been sent.

Algorithm 1 Redistribution of data using P2P MPI functions.

```
if ($myId \geq frsSrc$ && $myId \leq lstSrc$) then
    // Send sizes and values
    for ( $i = frstTrgt$; $i \leq lstTrgt$; $i{+}{+}$ ) do
        if ( $i == myId$ ) then
            Local copy using memcpy
        else
            MPI_Isend: $myId \rightarrow i$, $tag = 77$ // Send size
            MPI_Isend: $myId \rightarrow i$, $tag = 88$ // Send values
        end if
    end for
end if
if ( $myId \geq frstTrgt$ && $myId \leq lstTrgt$ ) then
    // Recv sizes
    $numRcv = 0$
    for ( $i = frsSrc$; $i \leq lstSrc$; $i{+}{+}$ ) do
        if ( $i \neq myId$ ) then
            MPI_Irecv: $myId \leftarrow i$ , $tag = 77$
            $numRcv + +$
        end if
    end for
    // Recv values
    while ( numRcv > 0 ) do
        MPI_Waitany: $tag$ , $source$ , $size/values$
        if ( $tag == 77$ ) then
            Create internal structures: $size$ for $source$
            MPI_Irecv: $myId \leftarrow source$ , $tag = 88$
        else // Reception from $source$ is completed
            $numRcv - -$
        end if
    end while
end if
if ($myId \geq frsSrc$ && $myId \leq lstSrc$) then
    // sources verify that the operations have been completed
    MPI_Waitall: $tag = 77$ , $tag = 88$
end if
```

Algorithm 2 Redistribution of data using Collective MPI functions.

```
// Send/Recv sizes
MPI_Alltoall: $sources \rightarrow targets$
if ( $myId \geq frstTrgt$ && $myId \leq lstTrgt$ ) then
    Create internal structures
end if
// Send/Recv values
MPI_Alltoallv: $sources \rightarrow targets$
```

Collective MPI functions (COL) are free of deadlocks, therefore the use of *blocking* functions, as `MPI_Alltoall` or `MPI_Alltoallv`, is a good alternative. The corresponding code is simpler than the previous one, because many of the aspects that programmers have to manage when using P2P functions are solved, such as the local copy or the non-null intersection of source and target groups. Algorithm 2 shows the operations to perform.

3.2 Overlapping data redistribution and application execution

Data redistribution and application execution in sources can overlap, allowing the final execution time to be faster than the sum of the individual execution times. But, the communicator used for data redistribution and the one used for the application should be different to avoid communication deadlocks. This overlapping is only possible when redistributing constant data, which is maintained throughout the execution of the application, since redistributing variable data requires the sources to halt their execution.

There are different alternatives to ensure that sources continue computing while the communication of constant data is completed: using *non-blocking* MPI operations or creating *auxiliary threads* to manage the communication as a background task.

Non-blocking versions can be used in both P2P and COL MPI functions. Algorithm 1 already employs *Non-blocking* P2P operations, but the use of `MPI_Waitall` forces the sources to wait until communication is finalized. The alternative for sources would be to verify the completion of the communication using `MPI_Testall` instead of `MPI_Waitall`. For collective communications, nothing changes for targets but sources should use either `MPI_Ialltoall` or `MPI_Ialltoallv` in Algorithm 2, whereas the verification of the function completion also requires the use of `MPI_Testall`.

Algorithm 3 shows how the sources overlap data redistribution and application execution using non-blocking communications. Basically, a series of conditions are added at the beginning of the iteration checking if: a redistribution is being performed (*redistStart* == true); a reconfiguration is scheduled (*checkPoint(...)* == true); or the iteration has to continue normally. When *redistStart* equals *true*, the sources call `Test_Redistribution` function to verify if the communications have been completed. When the reconfiguration in sources is complete, the iteration is terminated by executing the `break` statement. However, if the calling source is also a target, it will create internal structures for receiving data, and the iteration will continue when the reconfiguration is complete. When *redistStart* is *false*, *checkPoint* function will contact RMS to know if a reconfiguration has been started and then a new redistribution stage should be performed, setting *redistStart* to *true*.

At the end of the loop, the value of *endLoop* determines whether the loop has ended due to a malleability step (*is false*), or because *loopControl* has decided that the loop should terminate (*is true*).

The main goal of the **creation of auxiliary threads** is to relieve sources of the responsibility of data redistribution. This way, the auxiliary threads handle the communication, while the main ones continue computing. Comparing Algorithms 3 and 4 shows that in the first, a `MPI_Testall` is executed by the sources to check the completion of the communication. In the second, a new thread is created for each source to execute the MPI communication functions. The sources know that the communication has finalized when the corresponding auxiliary thread activates the shared boolean variable *endThread*, in the checkPoint function.

The auxiliary threads often execute Algorithm 1 or Algorithm 2, since they are the best alternative for communicating information. In fact, there is no reason to use a non-blocking strategy with threads because their only objective is to complete the data redistribution as fast as possible. Additionally, this is a good strategy to improve performance when there are enough unused CPUs in the node, but it can overload the system when all CPUs are busy. The reason is that `MPI_Waitall` is implemented as a *polling* loop, wasting CPU-time. The best alternative would be an implementation of `MPI_Waitall`, based on a *blocking* loop, reducing its impact on the CPU usage.

Algorithm 3 Code for sources that overlap data redistribution and application execution using non-blocking operations.

```
endLoop = false; redistStart = false;
while ( not endLoop ) do
    // Begin malleability code
    if ( redistStart ) then
        if ( Test_Redistribution (...) ) then
            redistStart = false;
            if ( not ( is_target ( myId ) ) then
                break;
            end if
        end if
    else if ( checkPoint (...) ) then
        redistStart = true;
        Start data redistribution using non-blocking functions
    end if
    // End malleability code
    Code related to an iteration
    endLoop = loopControl( ...);
end while
Code to conclude the program
```

Algorithm 4 Code for sources that overlap data redistribution and application execution creating auxiliary threads.

```
endLoop = false; redistStart = false; endThread = false;
while ( not endLoop ) do
    // Begin malleability code
    if ( redistStart ) then
        if ( endThread ) then
            redistStart = false; endThread = false;
            if ( not ( is_target ( myId ) ) then
                break;
            end if
        end if
    else if ( checkPoint (...) ) then
        redistStart = true; endThread = false;
        Create a new thread, which performs the communication
    end if
    // End malleability code
    Code related to an iteration
    endLoop = loopControl( ...);
end while
Code to conclude the program
```

4 RESULTS

This section presents the experiments performed to compare the methods described in Section 3 to redistribute data in a reconfiguration, which can be performed using the methods and strategies defined in [16]. The results were obtained using a synthetic iterative application [15, 17], in which the computation time and the communications pattern executed in each iteration can be parameterized.

In this section, first we shortly describe the synthetic application used to emulate the behaviour of the application. Next, we describe the main features of the emulated application, both the algorithm and the managed data. Finally we present the performance analysis.

4.1 Synthetic application

The authors in [15, 17] described a synthetic application, which allows configuring benchmarks to analyze the effect of malleability in applications. This tool allows to emulate and monitor the computational behaviour of scientific MPI iterative applications. Additionally, it also provides the possibility of being reconfigured during its execution, emulating the RMS demands, in which the number of processes of a job is expanded or shrunk. The executions of the tool are parameterized through a configuration file, which includes the main features of the computational behaviour and the communication pattern of the emulated application, as well as the description of the reconfiguration stages.

This tool includes five main modules: Initialization, Application, Malleability, Monitoring and Completion. Each of them is briefly described below.

Initialization module is in charge of starting the execution of the emulations. The main task of this module is to read the parameters from the configuration file and copy them to the new processes after each reconfiguration. This is performed by the first group of processes (those that start execution), which is also responsible for initializing the other modules of the synthetic application. Then, the first group will start the execution of the emulated application in the *Application emulation* module.

Application emulation module simulates the execution of an iterative application with a specific computational behaviour and and the communication pattern at each level of the process hierarchy. The main features of the emulation are defined in some parameters included in the configuration file. The most important features include the number of iterations to be performed by the active processes at each level of the hierarchy, the operation type (communication/computation) executed in each iteration of the emulation, the time spent for completing an iteration, the memory consumed in the emulated application, and the number of bytes transferred in both P2P and COL operations. From these parameters, some others must be computed, such as the number of times the operation type must be executed to achieve the time spent to complete an iteration. Furthermore, this module is responsible for ensuring that a emulation step is computed as many times as the specified number of iterations.

Malleability module is in charge of modifying the number of active processes during the emulation. Two main tasks are involved: creating/terminating processes (*Processes management stage*) and redistributing data from source to target processes (*Data redistribution stage*). For the first task, eight different expand/shrink alternatives defined in [16] are evaluated, always going from NS sources to NT targets. Two main methods are considered: *Baseline* method (always spawning NT new processes) or *Merge* method

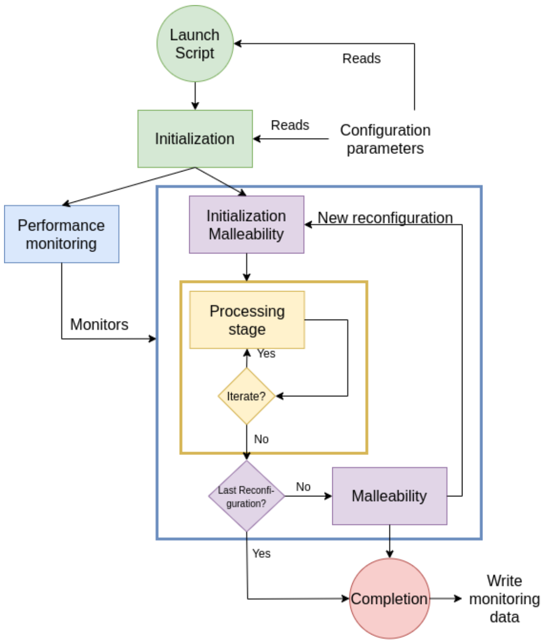

Figure 1: Flowchart of the synthetic application. The colour of each task corresponds to the related module, green for *Initialization*, yellow for *Application emulation*, purple for *Reconfiguration*, blue for *Monitoring*, and red for *Completion*.

(creating/terminating a number of processes equal to the positive difference of NS and NT).

Monitoring module keeps track of the different parts of the emulation timings. These values are stored in intermediate output files when each level of the hierarchy finalizes its execution.

Completion module has two main tasks. On the one hand, it finalizes processes at the end of each level of the process hierarchy. On the other hand, it is also responsible for writing the timings monitored into the intermediate files for further analysis.

Figure 1 shows a workflow diagram of this synthetic application.

4.2 Emulated application description

Our tests with the synthetic application emulate the execution of the Conjugate Gradient (CG), solving a sparse linear system defined by the sparse matrix Queen_4147 [1].

CG is the most common solver for the resolution of positive-definite sparse linear systems ($Ax = b$). This is an iterative method in which the solution is obtained as the projection of an initial vector on a Krylov subspace defined by the coefficient matrix and the residual [19]. In each iteration, a sparse matrix-vector product (SpMV), two Dot products and three Axpy(-like) operations are computed. Given its relevance, a High Performance Conjugate Gradient benchmark[2] has been ultimately defined as a complement of the High Performance LINPACK, which is currently used to rank the TOP500[3] computing systems.

[1]http://sparse.tamu.edu/Janna/Queen_4147
[2]https://hpcg-benchmark.org
[3]https://www.top500.org

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

- *Parallel computation of* S<span style="font-variant:small-caps">p</span>MV. Each process needs a full version of the distributed vector before the local computation has to be performed, and therefore `MPI_Allgatherv` function has to be executed.
- *Parallel computation of* D<span style="font-variant:small-caps">ot</span> *product.* The D<span style="font-variant:small-caps">ot</span> products compute some scalars which are required in all processes, therefore after the local computations using the distributed vectors, `MPI_Allreduce` function should be executed.
- *Parallel computation of* A<span style="font-variant:small-caps">xpy</span>. Since the same distribution is applied to all vectors, this computation is fully parallel, so there is no communication.

As a summary, we conclude that the parallel CG is composed of three communication operations: one `MPI_Allgatherv` and two `MPI_Allreduce`.

Thus, the emulation of the parallel CG needs to define 6 different stages in the synthetic application. Three of them based on intensive matrix computation, two for an `MPI_Allreduce` of one double each, and one `MPI_Allgatherv` of N doubles. The variable N is the number of rows in the matrix, which for the sparse matrix Queen_4147 is $4 * 10^6$ elements, so each process uses approximately 33 MB to store the full version of the distributed vector.

4.3 Hardware and software setup

The experiments were executed on a cluster 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 100GB/s and also with an Ethernet 10GB/s. Two different MPI versions have been used to compile and link the sources. MPICH 4.0.3 [21] was compiled with CH4:OFI netmod (Infiniband) [4] and MPICH 3.4.1 was compiled with CH3:Nemesis netmod (Ethernet).

The study only considers a single reconfiguration stage per experiment, doing it from 2, 10, 20, 40, 80, 120 and 160 processes to any of the same numbers, having 42 different pairs depending on the number of sources and targets. The number of occupied nodes in each execution will be computed as $\lceil N/20 \rceil$, where N will be the maximum between the number of sources (NS) and targets (NT), to minimize the resources allocated by the RMS.

Since only one reconfiguration is considered, two groups of processes are defined, sources and targets, and the malleability stage starts in iteration 500 out of 1000.

The sparse matrix and the vectors are distributed by row blocks among the processes of a group, so that the application allocates 3.947 GB of memory, approximately. This is the number of bytes that will be redistributed during the reconfiguration and 96.6% of it can be redistributed asynchronously.

In the definition of configuration files, both Baseline and Merge methods are included, and for data redistribution all the methods described in Section 3 are used. A total of 12 different configurations are analyzed, because when an asynchronous strategy is activated for the spawn method, the same strategy is also activated for data redistribution.

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

For each configuration and pair of process group, five executions are performed, computing the median of execution times. Then, the Shapiro-Wilk [20], Kruskal-Wallis [11] and Post hoc Connover [2] statistical tests are used to characterize the different configurations related to each pair of process group.

All configurations reject the null hypothesis (H0) of Shapiro-Wilk that data comes from a normal distribution, and therefore medians and non-parametric tests should be used. With the test Kruskal-Wallis, we check the H0 of the 12 configurations that have the same median. For pair of groups rejecting H0, the Post hoc Connover is performed to discover which configurations are different.

For all these analysis, Python 3.9.7 was used along with the modules Numpy 1.20.3 [5], Pandas 1.4.1 [18], SciPy 1.7.3 [23] and Scikit_posthocs 0.6.7 [22].

The next analysis are mainly focused on results of reconfigurations which scale from 160 sources or to 160 targets, since showing all the cases complicates the reading of the plots and the conclusions of the others reconfigurations are similar. In this regard, data from the execution of 42 pairs of process groups are shown and compared to determine the best configurations.

4.4 Evaluation of the reconfiguration techniques in isolation

This section studies the reconfiguration times of the different methods in isolation, without taking into account the executed application. This time is measured from the sources start spawning processes until the data has been fully received in the targets.

First, blocking reconfigurations (Synchronous methods) will be analyzed. Then, asynchronous reconfigurations (Asynchronous methods) are studied. These last ones are able to overlap malleability Stages 2 and 3 with the application execution. Finally, the two approaches are compared.

4.4.1 ***Synchronous methods.***

Figure 2 shows the reconfiguration time, in seconds, for synchronous methods when reconfiguring from 160 sources (top) or to 160 targets (bottom) in an Ethernet-based network.

In both plots, Merge reconfigurations always outperform Baseline strategies. Only when the Baseline method is implemented using P2P for expansion, its performance is barely the same to that of the Merge reconfigurations, with a difference of less than one second. When shrinking, this difference increases as the number of targets grows, since the Merge method does not spawn new processes.

COL-based implementation of Baseline methods underperforms in relation to the P2P-based counterparts, with a difference greater than 2 seconds. This is because in all executions that use blocking functions, there is oversubscription of $NS + NT$ processes during the reconfiguration.

Figure 3 shows the reconfiguration times, in seconds, for the synchronous methods when shrinking from 160 sources (top) or expanding up to 160 targets (bottom) in the Infiniband setup.

The analysis of these plots shows again that Merge reconfigurations are preferred, but in this case the minimum difference between the worst and the best case is only a second. Now, the

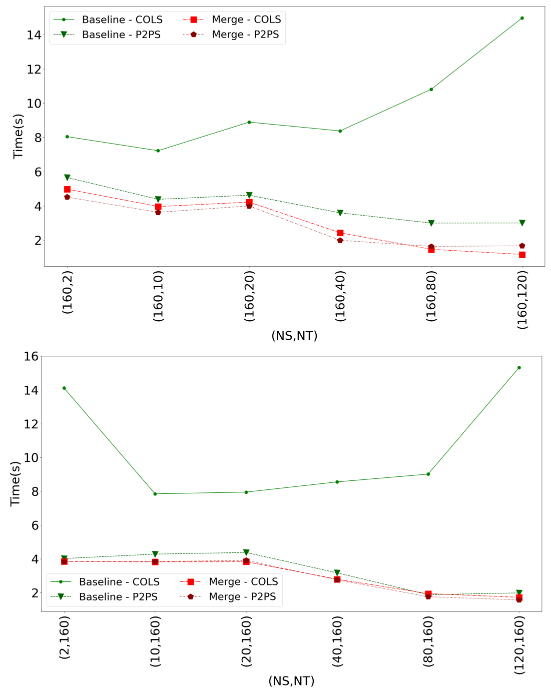

Figure 2: Reconfiguration times for synchronous methods using Ethernet. Shrinkage (top), Expansion (bottom).

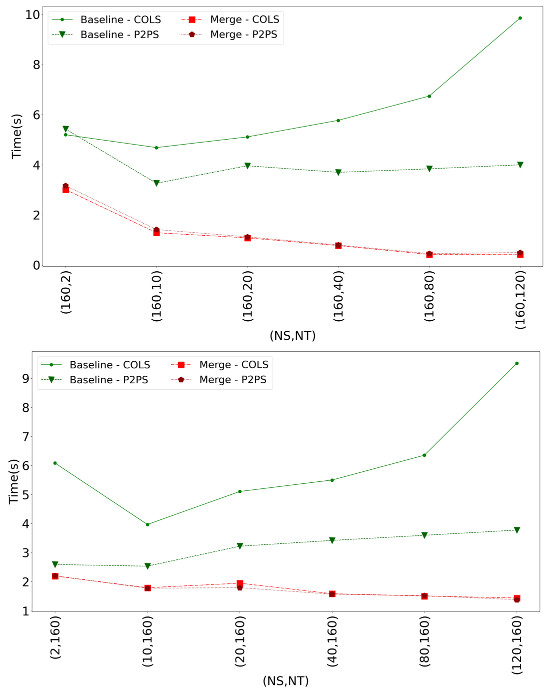

Figure 3: Reconfiguration times for synchronous methods using Infiniband. Shrinkage (top), Expansion (bottom).

oversubscription affects both Baseline implementations, performing worse when using COL operations. Additionally, the behaviour of both Merge reconfigurations is very similar.

Comparing Figures 2 and 3, we can observe that the execution time of Merge reconfigurations is low when the number of processes grows, regardless of the communication network. Moreover, there is a reduction of the reconfiguration time for Infiniband, for all cases, since it is a faster network.

Therefore, Merge reconfigurations are preferred for both networks as they avoid the oversubscription scenarios. But there is no criterion to choose one or the other (P2P or COL).

4.4.2 *Asynchronous methods*.

In this subsection, the asynchronous methods are compared to their synchronous counterpart. For each synchronous method there are two asynchronous, one based on non-blocking MPI calls and another one based on *auxiliary threads*.

To complete this comparison, we define α as the quotient of asynchronous and synchronous time. Thus, if α is greater than one, this indicates that the asynchronous method is more expensive than the synchronous method.

Figure 4 shows α values when reconfiguring from 160 sources (top) or to 160 targets (bottom) using Ethernet. In the legend, configurations ending with the character *S*, refer to synchronous methods, while configurations ending with the character *A*, refer to data redistribution with MPI non-blocking functions, and configurations ending with *T* use auxiliary threads.

When shrinking two major annotations can be made. On the one hand, some values are below 1, needing less time than the synchronous version even though the reconfiguration is being overlapped with application execution. On the other hand, values of α above 1 show an increment on the interval of 1% and 45% with the exception of Baseline configurations with P2P redistributions.

The first observation will be explained later, as also occurs for expansions on Infiniband network. With respect to the increment of the α values, it is a consequence of overlapping the reconfiguration with the application execution. Also we can conclude that Merge configurations provide more stable values than the Baseline ones as there is no oversubscription.

When expanding, Merge configurations produce even more stable α values, while Baseline configurations are more disperse. In the case of baseline P2P configurations, as more sources participate, the oversubscription is higher, producing also larger α values.

For the Merge configurations, when using 10 or more sources, they are hardly affected by overlapping tasks as the α ranges from 1% to 20% with the exception of the Merge P2P with an increase of 50%.

In general, P2P configurations always produce higher α values in relation to their COL counterpart. This denotes P2P communications are more affected by overlapping tasks than the collective operation COL.

In all cases, both expansion and shrinkage, the methods that use auxiliary threads(T) for the reconfiguration produce higher values of α than their counterparts which use non-blocking MPI functions. This occurs as a result of oversubscription of the auxiliary threads.

Figure 5 shows the α values for the asynchronous methods with the same previous reconfigurations, shrinking (top) and expanding (botton) using Infiniband.

When shrinking, the ranges of α differ from −44% to 374%. The highest values come from configurations which use auxiliary

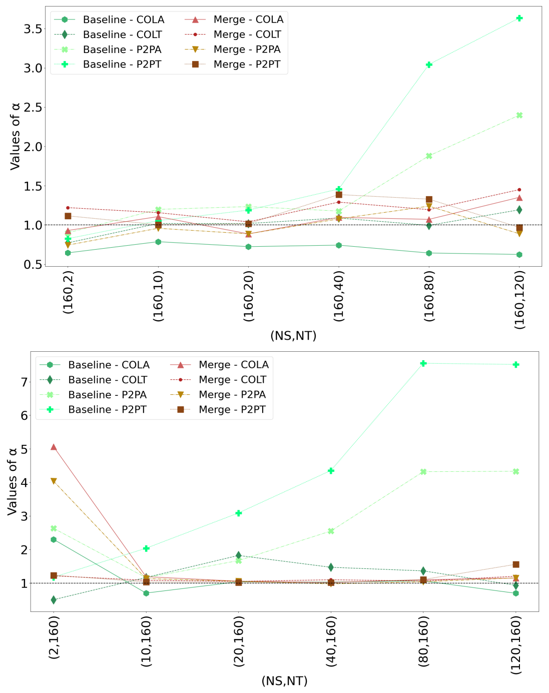

Figure 4: Reconfiguration α values for asynchronous methods using Ethernet. Shrinkage (top) and Expansion (bottom).

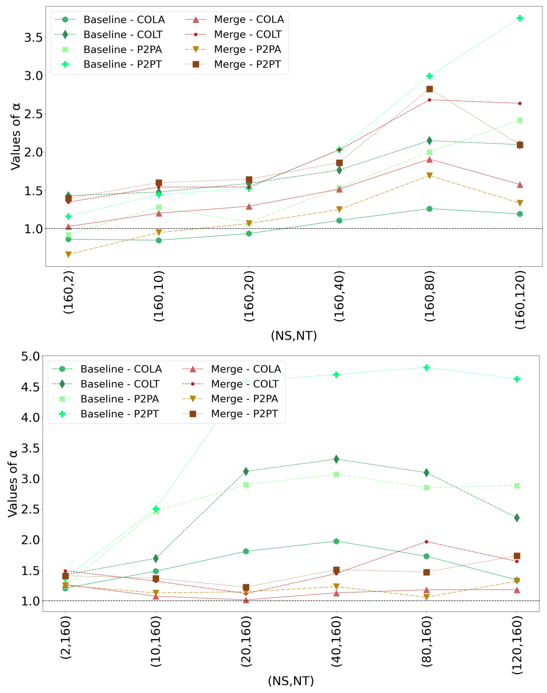

Figure 5: Reconfiguration α values for asynchronous methods using Infiniband. Shrinkage (top), Expansion (bottom).

threads(T), which always provide higher values than their non-blocking(A) counterpart.

With the Infiniband network, all the configurations obtain higher α values than the obtained in Figure 4, growing with the number of targets. This denotes that overlapping affects more to the reconfiguration task than with Ethernet. Additionally, Baseline configurations usually have higher α values than the Merge ones due to oversubscription.

When expanding, the highest α values are produced by the Baseline P2PT and COLT configurations, due to oversubscription. Nevertheless, this is not the case for the Baseline COLA, based on non-blocking MPI functions, in which the behaviour is closer to Merge configurations.

In the case of the Merge configurations, the values become closer to 1 with a maximum of 96% increase. Among the Merge configurations the ones based on non-blocking functions(A) produce lower values of α and their counterpart based on auxiliary threads(T) always have higher values.

Some α values in Figures 4 and 5 are less than 1, but this would be counter-intuitive, as it would imply that the asynchronous reconfiguration would take less time than the synchronous one, even though it overlaps the computation/communication stages of the execution. Moreover, all cases with values with a negative difference greater than 5% are related to non-blocking MPI Baseline-COLA configurations.

This is due to the different algorithms of `MPI_Alltoallv` when using inter- or intra-communicators in MPICH [21]. On the one hand the algorithm for the method COLS is `PairWise Exchange`, which is based on serialized synchronous communications with `MPI_Sendrecv`. On the other hand, the remainder of options discussed in Section 3 are at least partially based on asynchronous communications, which explains the appearance of similar or lower values for α. Therefore, the usage of blocking inter-communicators with `MPI_Alltoallv` is not recommended in MPICH.

4.4.3 *Overall evaluation*.

Figure 6 shows graphically the best method for each pair (*NS* sources, *NT* targets) to perform a redistribution for the Ethernet network (left) and Infiniband network (right). The name of the axes determines that the upper triangular part of the matrix is related to expansion, whereas the lower part is related to shrinkage. Moreover, the number in each cell, along with the colour, identifies the fastest method for each pair according to the tests Kruskal-Wallis and the Post hoc Connover. In case of a tie, the remaining cells will be checked to see which method of this cell appears more often, and this will be selected.

For expansions (upper triangle) with the Ethernet network, the preferred configuration is the Merge spawn with the COLS redistribution. There are four exceptions with the Merge P2PS and Baseline P2PS. The same occurs for Infiniband, where the Merge COLS is the preferred method with three exceptions for those previous methods.

These exceptions appear as a result of similar reconfiguration times between Merge COLS, Merge P2PS and Baseline P2PS as can be seen at Figures 2 and 3.

For shrinking (lower triangle) with the Ethernet network, the preferred configuration is again Merge COLS with exceptions for Merge P2PS and Merge P2PA. When using the Infiniband network all except one cell indicate Merge COLS as the preferred method.

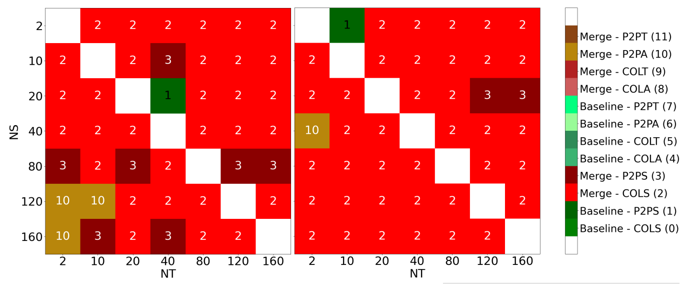

Figure 6: Preferred methods to reconfigure depending on the number of NS and NT. Ethernet (left), Infiniband (right).

Shrinking from 160, 120, 40 sources to 2 or 10 targets, which prefers the asynchronous alternative is a result of overlapping tasks without oversubscripting the processors.

Therefore, the fastest method to reconfigure data is Merge COLS regardless of expanding or shrinking, or the type of network used.

4.5 Evaluation of reconfiguration techniques

The second analysis compares the time required to execute an application that performs a single reconfiguration. The main difference between this analysis and the one presented at Section 4.4 is to show how the behaviour of the application is changed by overlapping it with the reconfiguration operation.

Figure 7 shows the reconfiguration time in seconds for Baseline COLS (right axis) and the speedup time against that configuration (left axis), when reconfiguring from 160 sources (top image) or to 160 targets (bottom image) using the Ethernet network.

Shrinking and expanding based on Baseline P2PS or Merge configurations, provide a speedup which increases with the number of processes. The speedup reaches its top at 1.14x for Merge P2PT. However, the asynchronous baseline configurations perform similarly or worse than the Baseline COLS. This is because they increase the cost of the iterations from a 20% to up to 7000% when expanding to 160 processes.

Therefore, oversubscription in asynchronous Baseline configurations increases the iteration time making it impossible to outperform their blocking counterparts.

Figure 8 shows the reconfiguration time, in seconds, for the Baseline COLS (right axis), and the speedup time against that configuration (left axis), when reconfiguring from 160 sources (top) or to 160 targets (bottom) using the Infiniband network.

As with the Ethernet network, Baseline P2PS and all Merge configurations for shrinking or expanding provide an speedup in most situations, which increases even more with the number of processes. The maximum speedup reached is 1.21x for Merge P2PA. For the asynchronous Baseline configurations, they still have the same issue with the iteration cost increasing from 80% up to 6800%.

Figure 9 shows graphically the best method for each pair (*NS* sources, *NT* targets) to execute the emulated application for the Ethernet network (left) and Infiniband network (right). 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, whereas the lower part is related to shrinkage. The number in each cell is selected by the same criteria than in Figure 6.

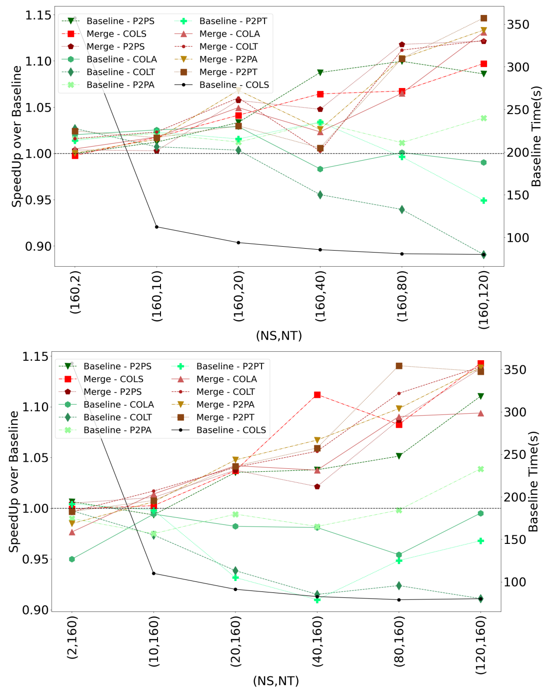

Figure 7: Application execution times using Ethernet. Right axis shows the time for the Baseline COLS and left axis shows the speed up of the other methods against the basic method. Shrinkage (top), Expansion (bottom).

For the Ethernet network, the preferred method is the Merge COLT, which appears in 29 out of 42 cells. This is the result of having values of α near 1 whereas the number of iterations performed during the reconfiguration is between 10 to 80. Therefore, with a similar reconfiguration time to the synchronous methods, but with more iterations performed during resizing, the targets have to perform fewer iterations.

Some exceptions are found when expanding from 2 processes using Baseline P2PS or synchronous Merge configurations. This occurs because the α values grow in the range of 25% to 50% in that row, and the number of overlapped iterations, between 10 and 20, is not enough iterations to compensate for the execution time of synchronous configurations. A similar situation happens with Merge P2PT strategy, since they perform a higher number of iterations than Merge COLT.

When shrinking, it is preferable to perform as many iterations as possible before reconfiguring, due to using more processes decreases the iteration time, so the longer it takes to reconfigure to a lower number of processes, the more iterations will be executed.

For the Infiniband network, the preferred method are the Merge COLA and P2PA, which are in 36 out of 42 cells. The first one was chosen for the color-map as it is considered a simpler implementation.

Both have α values in the range 1% to 30% with some exceptions when expanding from 2 processes. Furthermore the number of overlapped iterations is in the range of 5 to 10 iterations. Then, as the reconfiguration time is similar to the synchronous methods, just a few iterations are enough to overcome them.

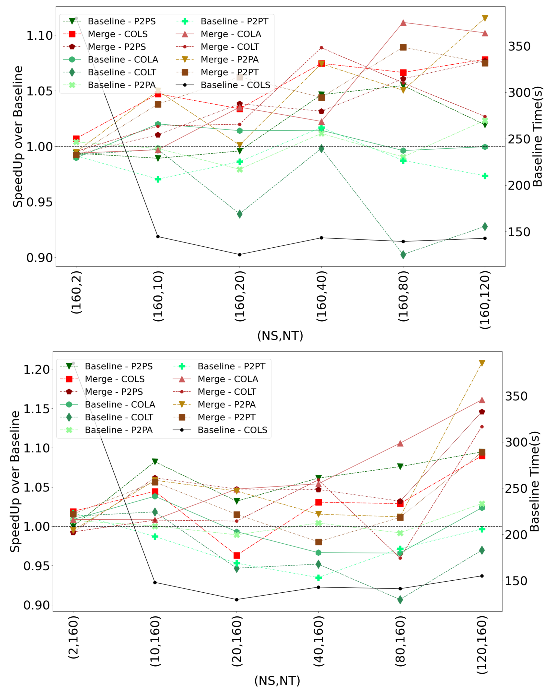

Figure 8: Application execution times using Infiniband. Right axis shows the time for Baseline COLS and left axis shows speed up of the other methods against the basic method. Shrinkage (top), Expansion (bottom).

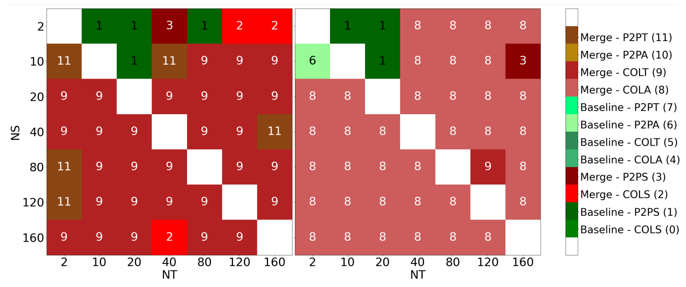

Figure 9: Preferred methods to reconfigure an emulated CG application depending on NS and NT. Ethernet (left), Infiniband (right).

Additionally, 6 exceptions are found where 4 of them are related to 2 or 10 sources. The reason is the same than with the Ethernet network, where the α values are higher and the performed iterations are not enough to overcome the synchronous methods.

When comparing both networks, the Ethernet network favours the use of auxiliary threads, while the Infiniband network prefers non-blocking methods.

5 CONCLUSIONS

The stages of a reconfiguration in an application when applying malleability have been studied. This paper examines different methods for performing reconfiguration in an application when using malleability. Two initial methods for data redistribution (Stage 3) are introduced (P2P and COL), on which two additional asynchronous strategies can be applied (non-blocking MPI functions and auxiliary threads). In addition, each method can be affected by how the previous spawn stage has been performed, using either Baseline or Merge method [16] (Stage 2). All these options generate twelve different options to perform a reconfiguration.

All of them have been evaluated by using a synthetic application configured to simulate a CG application, which is executed in a cluster with eight nodes using either an Ethernet or Infiniband network. The analysis compared all options, taking into account either the reconfiguration time or the application execution time.

For the reconfiguration time, the fastest option is to use the synchronous Merge COL independently of the network used. In more detail, asynchronous methods are not preferred since they increase the reconfiguration time when they overlap with the execution of iterations, while among the synchronous methods the Merge-based ones reduce the spawn time in more than a second.

For the execution time with Ethernet, the Merge COL with the aid of auxiliary threads is preferred in 29 out of 42 cases, while for Infiniband, the non-blocking Merge COL is preferred in 36 out of 42 cases. These methods provide an speedup of 1.14 and 1.21, respectively, over the synchronous Baseline COL method.

The main difference among the Ethernet and Infiniband network is the use of auxiliary threads. Infiniband is able to take profit of the characteristics of the network, so that the negative impact of oversubscription can be avoided, while for Ethernet is better to dedicate threads to communication during the reconfiguration.

Future work will extend the experiments to analyse the behaviour of other methods, such as RMA for data redistribution. Furthermore, a strategy to minimize data transfers during the data redistribution stage when using the Merge method will be explored. The basic idea is to ensure that processes, which are source and target, keep as much of their data as possible

Finally, contact with the Slurm resource manager to request/assign resources will also be included. Thus, it will be possible to study how malleability affects the real makespan of a system.

ACKNOWLEDGMENTS

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".

REFERENCES

[1] I. Comprés, A. Mo-Hellenbrand, M. Gerndt, and H-J Bungartz. 2016. Infrastructure and API Extensions for Elastic Execution of MPI Applications. In *Proceedings of the 23rd European MPI Users' Group Meeting* (Edinburgh, United Kingdom) *(EuroMPI 2016)*. Association for Computing Machinery, New York, NY, USA, 82–97. https://doi.org/10.1145/2966884.2966917
[2] W J Conover and R L Iman. 1979. *Multiple-comparisons procedures. Informal report.* Technical Report. Los Alamos National Lab. https://doi.org/10.2172/6057803

[3] K. El Maghraoui, Travis J. Desell, Boleslaw K. Szymanski, and Carlos A. Varela. 2007. Dynamic Malleability in Iterative MPI Applications. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*. IEEE, Rio de Janeiro, Brazil, 591–598. https://doi.org/10.1109/CCGRID.2007.45

[4] A. Gupta, B. Acun, O. Sarood, and L.V Kalé. 2014. Towards Realizing the Potential of Malleable Jobs. In *21st International Conference on High Performance Computing (HiPC)*. IEEE, Goa, India, 1–10. https://doi.org/10.1109/HiPC.2014.7116905

[5] C.R. Harris, K.J. Millman, S.J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N.J. Smith, R. Kern, M. Picus, S. Hoyer, M.H. van Kerkwijk, M. Brett, A. Haldane, J. Fernández del Río, M. Wiebe, P. Peterson, G. Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T.E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[6] G. Houzeaux, R. M. Badia, R. Borrell, D. Dosimont, J. Ejarque, M. Garcia-Gasulla, and V. López. 2022. Dynamic resource allocation for efficient parallel CFD simulations. *Computers & Fluids* 245 (2022), 105577. https://doi.org/10.1016/j.compfluid.2022.105577

[7] Chao Huang, Orion Lawlor, and L. V. alé. 2003. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958*. Springer Berlin Heidelberg, College Station, Texas, 306–322.

[8] Sergio Iserte. 2018. *High-throughput Computation through Efficient Resource Management*. Ph. D. Dissertation. Universitat Jaume I, Castelló de la Plana. https://doi.org/10.6035/14101.2018.176272

[9] S. Iserte, R. Mayo, E.S. Quintana-Ortí, V. Beltran, and A.J. Peña. 2018. DMR API: Improving cluster productivity by turning applications into malleable. *Parallel Comput.* 78 (oct 2018), 54–66. https://doi.org/10.1016/J.PARCO.2018.07.006

[10] S. Iserte, R. Mayo, E.S. Quintana-Orti, and A.J. Pena. 2020. DMRlib: Easy-coding and Efficient Resource Management for Job Malleability. *IEEE Trans. Comput.* 70, 9 (2020), 1443–1457. https://doi.org/10.1109/TC.2020.3022933

[11] W.H. Kruskal and W.A. Wallis. 1952. Use of Ranks in One-Criterion Variance Analysis. *J. Amer. Statist. Assoc.* 47, 260 (1952), 583–621. http://www.jstor.org/stable/2280779

[12] P. Lemarinier, K. Hasanov, S. Venugopal, and K. Katrinis. 2016. Architecting Malleable MPI Applications for Priority-driven Adaptive Scheduling. In *Proceedings of the 23rd European MPI Users' Group Meeting (EuroMPI)*. Association for Computing Machinery, New York, NY, USA, 74–81.

[13] G. Martín, D.E. Singh, M.C. Marinescu, and J. Carretero. 2015. Enhancing the Performance of Malleable MPI Applications by Using Performance-aware Dynamic Reconfiguration. *Parallel Comput.* 46 (jul 2015), 60–77.

[14] I. Martín-Álvarez, J.I. Aliaga, M. Castillo, and S. Iserte. 2022. *Malleable synthetic tool manual*. Technical Report. Universitat Jaume I.

[15] I. Martín-Álvarez, J.I. Aliaga, M. Castillo, S. Iserte, and R. Mayo. 2021. A synthetic tool for analysing adaptive workloads. 19th Annual Workshop on Charm++ and Its Applications. https://www.youtube.com/watch?v=kwE2FiU3FM8#t=6h3m20s Accessed: 2022-03-15.

[16] I. Martín-Álvarez, J.I. Aliaga, M. Castillo, S. Iserte, and R. Mayo. 2023. Dynamic spawning of MPI processes applied to malleability. *The International Journal of High Performance Computing Applications* 0, 0 (2023), 1–25. https://doi.org/10.1177/10943420231176527 arXiv:10.1177/10943420231176527

[17] Iker Martín-Álvarez, Jose I. Aliaga, Maribel Castillo, and Sergio Iserte. 2023. Configurable synthetic application for studying malleability in HPC. *2023 31st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)* (2023), 128–135. https://doi.org/10.1109/PDP59025.2023.00027 Accepted, Waiting for publication.

[18] The pandas development team. 2022. *pandas-dev/pandas: Pandas 1.4.1*. https://doi.org/10.5281/zenodo.6053272

[19] Y. Saad. 2003. *Iterative Methods for Sparse Linear Systems* (second ed.). Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898718003 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9780898718003

[20] S. S. Shapiro and M. B. Wilk. 1965. An Analysis of Variance Test for Normality (Complete Samples). *Biometrika* 52, 3/4 (1965), 591–611. http://www.jstor.org/stable/2333709

[21] MPICH Development team. [n. d.]. MPICH Website. https://www.mpich.org/. https://www.mpich.org/

[22] Maksim A. Terpilowski. 2019. scikit-posthocs: Pairwise multiple comparison tests in Python. *Journal of Open Source Software* 4, 36 (2019), 1169. https://doi.org/10.21105/joss.01169

[23] P. Virtanen, R. Gommers, T.E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S.J. van der Walt, M. Brett, J. Wilson, K.J. Millman, N. Mayorov, A.R.J. Nelson, E. Jones, R. Kern, E. Larson, C.J Carey, İ. Polat, Y. Feng, E.W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C.R. Harris, A. M. Archibald, A.H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

Received ; revised ; accepted