Parallel Spawning Strategies for Dynamic-Aware MPI Applications

Iker Martín-Álvarez[a,*] (PhD student), José I. Aliaga[a] (Coordinator), Maribel Castillo[a] (Coordinator) and Sergio Iserte[b] (Senior Researcher)

[a]Dpto. de Ingeniería y Ciencia de los Computadores, Universitat Jaume I, Av. Vicent Sos Baynat, s/n , Castelló, 12071, Comunitat Valenciana, Spain
[b]Computer Science Department, Barcelona Supercomputing Center, , Barcelona, , Catalunya, Spain

ARTICLE INFO

Keywords:
Dynamic Resources
Malleability
MPI
Process Management

ABSTRACT

Dynamic resource management is an increasingly important capability of High Performance Computing systems, as it enables jobs to adjust their resource allocation at runtime. This capability has been shown to reduce workload makespan, substantially decrease job waiting times and improve overall system utilization. In this context, malleability refers to the ability of applications to adapt to new resource allocations during execution. Although beneficial, malleability incurs significant reconfiguration costs, making the reduction of these costs an important research topic.
Some existing methods for MPI applications respawn the entire application, which is an expensive solution that avoids the reuse of original processes. Other MPI methods reuse them, but fail to fully release unneeded processes when shrinking, since some ranks within the same communicator remain active across nodes, preventing the application from returning those nodes to the system. This work overcomes both limitations by proposing a novel parallel spawning strategy, in which all processes cooperate in spawning before redistribution, thereby reducing execution time. Additionally, it removes shrinkage limitations, allowing better adaptation of parallel systems to workload and reducing their makespan. As a result, it preserves competitive expansion times with at most a 1.25× overhead, while enabling fast shrink operations that reduce their cost by at least 20×. This strategy has been validated on both homogeneous and heterogeneous systems and can also be applied in shared-resource environments.

1. Introduction

Despite advances in High Performance Computing (HPC) systems during the exascale era, the efficient utilisation of resources such as CPUs, GPUs and memory remains a significant challenge [12]. Scientific applications often underuse their allocations due to factors such as overcommitment, bottlenecks or limited scalability.

To address this issue, Resource Management Systems (RMS) must be able to distribute resources dynamically, a capability known as Dynamic Resource Management (DRM). This capability enables RMSs to adjust job allocations at runtime, provided the applications are malleable, that is, capable of resizing and adapting to changes in resource availability during execution. Such flexibility allows applications to achieve more effective optimization of resource utilization [4], throughput [9], energy efficiency [3, 11], and I/O performance [23].

Recent studies by the US Exascale Computing Project [2] suggest that DRM could significantly enhance the performance of both applications and systems, making it a key contribution for improving relevant metrics in future platforms [13]. However, its adoption remains limited compared to other MPI features [6], primarily due to the complexity involved in adapting applications to support malleability and enabling systems to dynamically reallocate resources at runtime.

There are already libraries available to perform reconfigurations in applications; some of the most relevant are DMR[9], FlexMPI[14], MaM[16] or DPP[8]. A particularly challenging scenario for most of these tools is shrinking, i.e., reducing the number of active processes in a job at runtime to free resources or adapt to changing workload demands. This operation is limited by the rigid structure of MPI communicators, especially the standard `MPI_COMM_WORLD` (MCW), which does not allow the partial termination of ranks.

As shown in [20], terminate-based (TB) methods can shrink without spawning new processes, providing a time efficient alternative to traditional spawn-based (SB) approaches. Nevertheless, a major limitation of TB shrinking is the persistence of zombie processes, ranks that remain asleep until the application terminates, as they cannot be fully removed from the global MCW. This creates an issue: the nodes on which these zombies are mapped cannot be returned to the RMS until they terminate, preventing node reuse. A common workaround involves isolating each MCW communicator on a separate node, allowing finer control over process groups. However, this solution depends on node-specific spawning, which suffers from poor scalability due to its inherently sequential nature [22].

To overcome these constraints, we propose a novel technique that retains the benefits of TB shrinking while addressing its key limitations, supporting both expansion and shrinkage at minimal cost. Our approach isolates each MCW communicator on a separate node and enables all available

*Corresponding author
martini@uji.es (I. Martín-Álvarez); aliaga@uji.es (J.I. Aliaga); castillo@uji.es (M. Castillo); sergio.iserte@bsc.es (S. Iserte)
https://www.bsc.es/iserte-agut-sergio (S. Iserte)
ORCID(s): 0000-0002-3337-3298 (I. Martín-Álvarez); 0000-0001-8469-764X (J.I. Aliaga); 0000-0002-2826-3086 (M. Castillo); 0000-0003-3654-7924 (S. Iserte)

processes to perform spawning in parallel rather than sequentially, thereby reducing the cost of expansion. In addition, it avoids spawning new processes during shrinkage and eliminates the persistence of zombie processes while maintaining the high performance observed in TB methods. These improvements make our solution particularly suitable for DRM in large-scale HPC environments, with full support for heterogeneous systems and shared resources. The approach has been included in the MaM library [16], which integrates multiple techniques for resizing and data redistribution, allowing the selection of the optimal solution depending on the context.

The main contributions of this work are:

- The design and implementation of two novel parallel spawning algorithms tailored to the MPI model, enabling scalable malleability for HPC applications.
- The extension of the MaM library [16] to support dynamic reconfiguration in heterogeneous queue systems and shared-resource environments.
- A comprehensive performance evaluation of the proposed algorithms on both homogeneous and heterogeneous clusters, demonstrating their advantages over the previous best method in terms of scalability and reconfiguration overhead.

The remainder of this paper is organized as follows: Section 2 presents a comprehensive review of related work on MPI malleability and spawning techniques. Section 3 provides an overview of process spawning mechanisms in MPI and their integration into MaM. Section 4 explains the two new parallel spawning algorithms proposed in this work. Section 5 presents an evaluation of their performance. Finally, Section 6 concludes the paper and outlines directions for future work.

2. Background

According to the classification proposed by Feitelson and Rudolph [5], applications executed in large-scale computing facilities can be grouped into four categories. This classification is based on two main criteria: (1) whether the number of processes can change during execution, i.e., whether reconfiguration is supported, and (2) who determines the job size, either the user or the RMS. Table 1 summarizes this classification.

In this paper, we consider malleability as the ability of a distributed parallel job to change its size in terms of MPI ranks by adjusting the computational resources initially allocated to the job at any point during execution as often as required. This capability is activated at specific checkpoints within the application and requires the execution of a series of stages:

1. **Reconfiguration feasibility**: The RMS determines whether the job should be resized according to a dynamic resource allocation policy. If not, the subsequent steps are not performed.
2. **Process management**. The RMS assigns a new number of cores to the job, which dictates how many MPI ranks should be created or terminated. Processes before resizing are considered *sources*, while those that continue after resizing are considered *targets*.
3. **Data redistribution**: *Sources* transfer their data to *targets*.
4. **Resume execution**. The application resumes execution with the *targets*.

Incorporating all these stages into a parallel application is a complex task that goes far beyond simple integration with the RMS. It requires modifications to the application's structure to support dynamic creation and termination of processes, manage data redistribution efficiently, and reconfigure communication patterns on the fly. These challenges make dynamic applications difficult to implement from scratch, often requiring dedicated tools and frameworks that simplify the transformation of traditional applications into dynamic ones. Without such support, adapting applications to handle resizing becomes both error-prone and time-consuming.

One malleable solution is Flex-MPI [14], a library built on top of MPICH [21] that bases its spawn operations on the `MPI_Comm_spawn` function. Its primary goal is to help applications become dynamic and the main result of this solution is the Merge method, which is described below. When an application expands from x processes to y processes, only the additional $y - x$ processes are spawned and added to the existing ones. Then, each new process is created through a separate call to the spawn function, which allows a fine grain when shrinking the application, as dynamic processes can be terminated at any time. Nevertheless, there is a limitation with the minimum number of ranks: the initial number of ranks cannot be terminated until the end of the application.

A similar solution is MaM [16], a malleability module also built on top of MPICH and within the MPI scope, developed to analyze and compare different malleability techniques. Like Flex-MPI, it bases its spawn operation on the `MPI_Comm_spawn` function, providing a Merge method, but without the limitation of the minimum number of processes. MaM is part of the Proteo framework [19], which helps transform applications into malleable ones, supports the generation of dynamic workloads, and enables performance analysis of different resize operations to select the optimal alternative for a given situation.

Another approach is DPP [8], which explores the new concept of MPI Sessions, extending it to implement on-the-fly dynamic mechanisms. Since the sessions model operates outside the traditional MPI environment, the MCW communicator does not exist and some functions, such as `MPI_Comm_spawn`, cannot be used. However, alternative non-standard MPI functions are provided to allow communication between ranks. To spawn processes, it uses a custom dynamic RMS, which determines whether a job should get or return resources. When a job gets additional resources, it informs a PMIx Reference Runtime Environment (PRRTE) daemon, which then spawns of new processes. Once the

Job Type	Resource Allocation	Who Sets Size	Description
Rigid Job	Static	User	Can only be executed with a fixed number of processes defined by the user. No reconfiguration is allowed.
Moldable Job	Static	RMS	Can start with a variable number of processes. The RMS determines the size when the execution begins.
Evolving Job	Dynamic	User	Includes a user-defined reconfiguration scheme. The RMS must meet the resource requests for the job to continue executing.
Malleable Job	Dynamic	RMS	Can be reconfigured at runtime if the RMS decides to adjust the resource allocation.

Table 1
Classification of parallel jobs according to Feitelson and Rudolph [5].

processes are ready, the job is informed of the new resources and how to connect with them, which is managed by DPP. During shrink, processes do not have the limitations of the traditional environment and can be terminated normally.

Lastly, an alternative solution is Dynamic Management of Resources library (DMRlib) [9], which consists of two components: a parallel distributed runtime based on MPI and an extension of SLURM to enable the execution of malleable jobs. Initially, the library only allowed the simplest approach, the Baseline method, which always spawns the new number of processes y and terminates the previous x processes. However, the library now supports other environments for process management, such as DPP [7] or MaM [10], providing a wider range of possibilities.

For further details on MPI spawn based malleability, see [1], while for a broader review of dynamic resource management [24] is recommended.

3. MaM overview

MaM is the Proteo module responsible for completing malleability. It provides a set of features designed to transform traditional MPI applications into malleable ones, requiring minimal effort from developers. Additionally, it defines a simple and intuitive interface, which lowers the barrier to integrating malleability into existing parallel applications [16].

Specifically, MaM implements multiple methods and strategies to handle the most critical stages of malleability, process management and data redistribution, allowing applications to dynamically adjust their number of MPI ranks during execution.

For the process management stage, i.e., resizing from an initial group of NS processes to a new group of NT processes, MaM offers two main approaches: the Baseline method, which always creates a new group of processes, and the Merge method, which reuses existing processes and only spawns or terminates the difference in ranks. These methods can be combined with two strategies: the Asynchronous strategy, which overlaps process creation with application execution to reduce downtime, and the Single strategy, where only one process performs the spawn operation and informs the rest afterwards. This flexible design allows users to decide the preferred configuration according to their application needs.

For the data redistribution stage, MaM distinguishes between constant and variable data, as this affects how and when the redistribution can take place. While variable data requires execution to be paused until the transfer is complete, constant data can be redistributed asynchronously, allowing overlap with ongoing computation. The library also supports three main communication methods for this stage: Point-to-Point, Collective, or One Sided primitives.

Additionally, asynchronous redistribution can be implemented through two strategies: Non-blocking communication, which relies on routines like `MPI_Test` to monitor progress, and Threading, where dedicated threads handle data transfer independently of the main computation.

Using MaM to incorporate malleability into an application requires specifying a method with none, one, or more of the strategies for stages 2 and 3. A more detailed description of all alternatives related to process management can be found in [20] and to automatic data redistribution in [15] and [18].

4. Methods for spawning processes in parallel

As previously discussed, the MaM shrinkage approach is constrained by its inability to fully terminate unneeded processes during execution. As a consequence, is not possible to return those resources until the MCW they pertain terminates completely. To overcome these issues, this work proposes new strategies based on parallel spawning, which allows processes to be resized efficiently and overcome said situation.

Parallel process creation is inherently complex and is coordinated across multiple phases: (1) It begins with the spawn of new process groups and opening communication ports. (2) it is followed by a synchronization phase, where all groups wait until every required port is ready. (3) Next, the newly spawned groups are merged into a single communicator through the ports. (4) Finally, a global reordering of ranks is performed to maintain a logical order between nodes.

The first phase has been examined through two alternatives: A simpler option only for homogenous systems called Hypercube strategy, and an improved but more complex one

that works for heterogenous or shared systems, called Iterative Diffusive strategy. The following sections detail each of these steps, highlighting the specific operations involved.

4.1. Hypercube: Parallel Spawning for Homogeneous Systems

This parallel strategy is based on performing a call to the `MPI_Comm_spawn` function for each node that is to be used, assigning all available cores of the node to each newly created process.

The procedure is organized into a series of steps. In each step, all active processes, whether *source* or *target*, concurrently execute a spawn operation, each one acting on a different node. As the number of target processes increases, so does the number of processes participating in the spawning operation. Consequently, with each successive step, a growing number of processes are involved in initiating new spawns, enabling a scalable and distributed expansion across nodes.

In the first step, only the *source* processes participate in the operation. In total, up to $C \cdot I$ processes can be launched, where C is the number of cores per node and I is the number of initial nodes. From the second step onward, both the original and previously created processes participate, exponentially increasing the spawning capacity. However, in the final step, only the first x processes participate, where x corresponds to the remaining number of nodes needed to reach the desired number of nodes.

As an example, consider a system with 20 cores per node. Starting with a single fully occupied node, the first step could launch up to 20 additional nodes. In the second step, with 420 processes available (21 nodes with 20 cores each), it would be possible to occupy 410 additional nodes by creating 20 processes per node.

From a technical standpoint, this strategy is implemented using `MPI_Comm_spawn` in combination with the `MPI_COMM_SELF` communicator and an `MPI_Info` object that specifies the target node for each new group of processes. Furthermore, all processes in each node are assigned the group identifier ranging from 0 to the total number of groups to be created (*group_id*), identifying which group it belongs.

The number of steps s required to reach a total of N nodes, starting from I initial nodes and with C cores per node, can be calculated using Equation 1. N and I are calculated as $\frac{NT}{C}$ and $\frac{NS}{C}$, respectively, where NT is the amount of targets, and NS the amount of sources.

$$s = \frac{\ln(N/I)}{\ln(1+C)} \tag{1}$$

From this, the total number of nodes T_s at the end of step s can be computed as:

$$T_s = (1+C)^s \cdot I \tag{2}$$

And the corresponding number of processes spawned t_s in step s is simply:

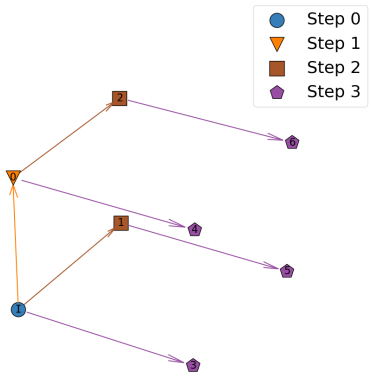


Figure 1: Generation of 7 groups using parallel spawning. Each step is represented with different markers.

$$t_s = T_s \cdot C \tag{3}$$

Figure 1 illustrates this strategy with a simple example in which the number of processes is expanded from $NS = 1$ to $NT = 8$, assuming an architecture with $C = 1$ core per node. In this case, seven additional groups are created over three steps. The process is visualized as a cube, where the edges indicate which group spawns which in each step, and the vertices are labelled with the corresponding `group_id`, or with the label I for the initial group. By the end of the procedure, a total of eight nodes are occupied.

4.2. Iterative Diffusive: Recursive Algorithm for Heterogeneous Systems

This method improves upon the previous scheme by including the ability to create process groups of variable sizes at each step, which the Hypercube is not capable of. This makes it particularly well-suited to heterogeneous or shared computing environments, where nodes may have different capacities or resources may be distributed unevenly.

As in the original scheme, the algorithm proceeds in discrete steps, where all existing processes collaborate to spawn new groups on additional nodes. The assignment of a unique group identifier (*group_id*), ranging from 0 to the total number of groups to be created, is also preserved.

Three vectors are needed to describe the system configuration. The number of entries in these vectors is equal to the number of nodes in the system (N).

- M: Contains the number of available cores assigned to a job on each node.
- A: Stores the number of processes running on each node before starting this algorithm.
- S: Has the number of processes to be created in each node when executing this algorithm.

Based on these vectors, a stepwise sequence can be constructed to determine the total number of processes and nodes involved at each spawning step. Only nodes with

positive entries in the vector S will allocate new processes; thus, each non-zero entry indicates the creation of a new group on the corresponding node. This vector is computed from the previous ones as follows:

$$S_i = M_i - A_i, \quad \text{for } i = 0, 1, \ldots, N-1$$

Equation 4 defines the total number of processes t_s present in step s, as a function of the cumulative total of previous steps (t_{s-1}) and the number of processes generated in the current step (g_s). Step $s = 0$ corresponds to the initial state, which includes only the *source* processes, as specified by the vector A.

$$t_s = \begin{cases} \sum_{j=0}^{N-1} A_j, & \text{if } s = 0 \\ t_{s-1} + g_s, & \text{if } s > 0 \end{cases} \tag{4}$$

Equation 5 defines the number of processes g_s generated in step s. At each stage, the elements of vector S from index ν_s to index λ_s are accumulated to calculate the number of new processes that need to be generated.

$$g_s = \sum_{j=\nu_s}^{\lambda_s} S_j \qquad \text{where} \quad \lambda_s = \min\left(N-1, \left(\sum_{i=0}^{s-1} t_i\right) - 1\right), \quad \nu_s = \sum_{i=0}^{s-2} t_i \tag{5}$$

Finally, Equation 7 represents the cumulative number of occupied nodes (T_s) up to step s, whereas Equation 6 specifies the number of new nodes (G_s) occupied by newly created groups during that step. In all equations, null elements in vector S are disregarded.

$$G_s = \min(N - T_{s-1}, t_{s-1}) \tag{6}$$

$$T_s = \begin{cases} I, & \text{if } s = 0 \\ T_{s-1} + G_s, & \text{if } s > 0 \end{cases} \tag{7}$$

4.3. Synchronization Between Process Groups

Once all processes have been created, it is necessary to establish connections between them to enable direct communication. With the adopted creation strategy, each newly spawned group initially knows only its parent process (the one that created it) and, if applicable, its own child processes. As shown in Figure 1, processes can only communicate directly with those to which they are connected by an edge in the spawn graph.

To enable communication between separate groups, the functions `MPI_Comm_connect` and `MPI_Comm_accept` are used. These functions establish an inter-communicator between two independent groups. However, to use them correctly, certain preliminary steps must be taken.

First, the process groups must be prepared for connection. The processes designated as roots in the `MPI_Comm_accept` calls must first open a port using `MPI_Open_port` and publish the service name via `MPI_Publish_name`. Since `MPI_Comm_connect` must be called after the corresponding port has been opened by the other group, a global synchronization phase between all groups is required. Otherwise, execution errors may occur, as observed with implementations such as MPICH[1].

The synchronization between groups consists of three steps and is carried out within each group using a dedicated subcommunicator. The objective is to ensure that all processes are aware of the port status before attempting to establish a connection.

1 Subcommunicator creation. Each group creates a subcommunicator using `MPI_Comm_split`, including the root process of the group and all processes that have created child groups. This subcommunicator will be used to coordinate intra-group synchronization.
2 Upside Synchronization. Each process that has spawned child groups waits to receive a token from each of them using `MPI_Irecv` followed by `MPI_Waitall`. Then, all processes in the subcommunicator execute a call to `MPI_Barrier`, ensuring that all tokens have been received. Finally, the root process (if not part of the source group) notifies its parent group using `MPI_Send`.
3 Downside Synchronization The root process of each group (except the source group) waits to receive a token from its creator via `MPI_Recv`. Then, processes in the subcommunicator perform a `MPI_Barrier` call (excluding the source group). Finally, each process that has spawned child groups sends them a token using `MPI_Isend` followed by `MPI_Waitall`. These barriers are essential to ensure that no group proceeds before all of its descendants have completed the Upside step, and to prevent any group from notifying its children before being notified by its parent group.

Listing 1 shows the implementation of the synchronization mechanism. The processes use the `world_c` communicator, which corresponds to MCW for spawned processes, or to a shared communicator among source processes in the case of creators. The `parent_c` communicator is an intercommunicator between a spawned group and its parent. The variable `qty_s` indicates the number of child groups spawned by a process, and the corresponding intercommunicators are stored in the `spawn_c` vector.

Figure 2 illustrates the *upside synchronization* and *downside synchronization* steps, shown on the left and right sides, respectively. In this example, six process groups are created in two spawning steps. The arrows indicate which processes send messages to signal their readiness in the *upside* phase, and to confirm that all groups are ready in the *downside* phase. A group represented in black with stripes indicates

[1] This behaviour has only been tested with MPICH.

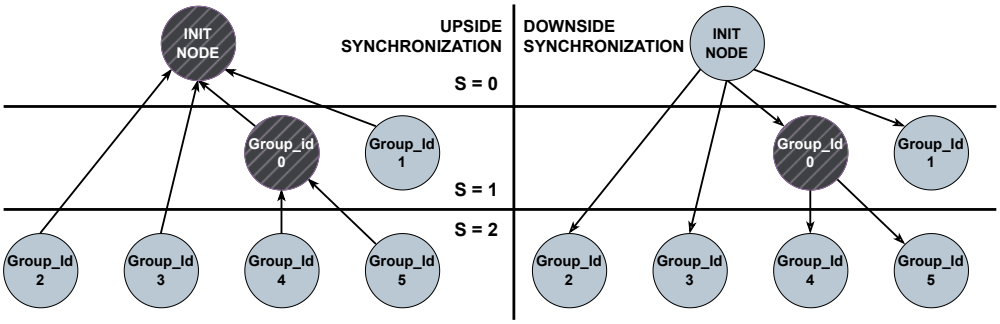


Figure 2: Synchronization of 6 spawned groups and the initial node. The arrows indicate the direction of messages, and stripes the need of that group of using a Barrier.

```
1  void common_synch(MPI_Comm world_c, MPI_Comm parent_c,
2       int qty_s, MPI_Comm *spawn_c) {
3    int myId, i, root = 0;
4    int color = qty_s ? 1 : MPI_UNDEFINED;
5    MPI_Request *reqs = malloc(qty_s * sizeof *reqs);
6    MPI_Comm synch_ranks;
7
8    //Only root ranks or ranks with children synchronize
9    MPI_Comm_rank(world_c, &myId);
10   MPI_Comm_split(world_c, color, myId, &synch_ranks);
11
12   // Wait for all children of this group
13   for(i=0; i<qty_s; i++) {
14     MPI_Irecv (..., source=root, spawn_c[i], &reqs[i]);
15   }
16   if(qty_s){MPI_Waitall(reqs); MPI_Barrier(synch_ranks);}
17
18   if(parent_c != MPI_COMM_NULL && myId == root) {
19     // Send to parent this group is ready
20     MPI_Send (..., dest=root, parent_c);
21     // Wait for the other groups readiness
22     MPI_Recv (..., source=root, parent_c);
23   }
24
25   // Notify this group that all groups are ready
26   if(parent_c != MPI_COMM_NULL && qty_s)
27     {MPI_Barrier(synch_ranks);}
28   // Notify children that all groups are ready
29   for(i=0; i<qty_s; i++)
30     {MPI_Isend (..., dest=root, spawn_c[i], &reqs[i]);}
31   if(qty_s) {MPI_Waitall(reqs);}
32
33   if(synch_ranks != MPI_COMM_NULL)
34     {MPI_Comm_disconnect(&synch_ranks);}
35 }
```

Listing 1: Syncrhonization step before binary connection.

that it must wait to receive all messages before proceeding. At that point, a `MPI_Barrier` is used to synchronize the group, after which the execution can continue.

4.4. Binary connection

Once synchronization between all groups is complete, the connection operation can begin. The adopted strategy consists of making pairwise binary connections in several successive steps. In each step, groups with an identifier less than half the number of active groups perform an

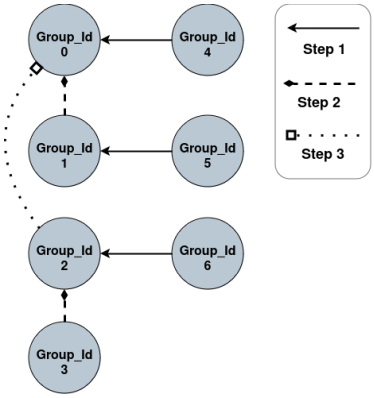


Figure 3: Connection of 7 spawned groups in 3 steps. In each step the amount of groups is halved and connections operations may be unordered.

`MPI_Comm_accept` operation, while those with a higher identifier execute an `MPI_Comm_connect`. In the case of an odd number of groups, the middle group is temporarily excluded from that step and proceeds directly to the next one.

Each group connects to the group whose identifier is its mirror image within the current set of active groups, that is, the group symmetrically positioned relative to the center. Once the connection is completed, both groups are merged into a new communicator and adopt a common group identifier, corresponding to the group that executed the `MPI_Comm_accept` call.

This procedure is repeated, halving the number of active groups at each step, until all processes are connected within a single group. Figure 3 shows an example in which seven spawned groups are connected in three steps, continuing the scenario previously illustrated in Figure 1.

Thanks to the synchronization mechanism described earlier, no strict ordering is required for the execution of Connect operations. Groups performing accept simply wait for incoming connections, regardless of the order in which they arrive.

Listing 2 presents a simplified version of the code used to implement this binary connection strategy between groups. In the code, the groups are distinguished based on whether they should perform a `MPI_Comm_accept` operation (lines

```
1  void binary_connection(int groups, int group_id,
2                         char *my_port, MPI_Comm *new_comm) {
3    int middle, new_groups, new_gid;
4    char *port;
5    MPI_Comm merge_comm, aux_comm, intercomm;
6    merge_comm = aux_comm = MPI_COMM_WORLD;
7    intercomm = MPI_COMM_NULL;
8    while(groups > 1) {
9      middle = groups / 2;
10     new_groups = ceil(groups / 2.0);
11     if(group_id < middle) {
12       MPI_Comm_accept(my_port, MPI_INFO_NULL,
13         MAM_ROOT, merge_comm, &intercomm);
14       MPI_Intercomm_merge(intercomm, 0, &aux_comm);
15       merge_comm = aux_comm;
16
17     } else if(group_id >= new_groups) {
18       new_gid = groups - group_id - 1;
19       get_remote_port(new_gid, &port);
20       MPI_Comm_connect(port, MPI_INFO_NULL,
21         MAM_ROOT, merge_comm, &intercomm);
22       MPI_Intercomm_merge(intercomm, 1, &aux_comm);
23       merge_comm = aux_comm; group_id = new_gid;
24
25     } groups = new_groups;
26   } *new_comm = merge_comm;
27 }
```

Listing 2: Binnary connection step for the parallel spawn.

L11–L15) or a `MPI_Comm_connect` (lines L17–L23). In the latter case, the group obtains the corresponding port name (line L19) using the *group_id* of the target group and the `MPI_Lookup_name` function.

4.5. Rank Reordering and Connection with *sources*

Once all the generated groups have been merged into a single one, the process ranks may not be ordered across nodes. This is because binary connections between groups do not enforce a specific order and are susceptible to race conditions.

To ensure a consistent global order, an `MPI_Comm_split` call is made, where each process specifies its new rank as follows:

$$\text{world\_rank} + \sum_{j=0}^{N-1} A_j + \sum_{j=0}^{\text{group\_id-1}} S_j$$

At the same time, all intermediate communicators created throughout the strategy are disconnected using `MPI_Comm_disconnect`. This frees up resources and ensures that any node can be removed or replaced in future reconfigurations without leaving residual dependencies.

Finally, the *target* group is connected to the *source* group, establishing the final inter-communicator that links them. This operation concludes the process creation and connection phases.

5. Experimental results

5.1. *Hardware* and *Software*

The experiments were conducted on two clusters. The first one, MarenostrumV[2] (MNV), used 32 nodes from the general queue. Each node has two 56-core Intel Xeon 8480 CPUs (3584 cores in total). This system was employed to analyze the behaviour of both spawning techniques described in Section 4.

The second cluster, Nasp[3], consists of two sets of 8 nodes. The first set has nodes with two 10-core Intel Xeon 4210 CPUs (160 cores total), connected through a 100 Gbps InfiniBand EDR and a 10 Gbps Ethernet network. The second set uses nodes with 32-core Intel Xeon 6346 CPUs (256 cores total) connected via 10 Gbps Ethernet. Both sets communicate through a shared 10 Gbps Ethernet link between their switches. This heterogenous cluster was used to evaluate the effectiveness of the improved method in Section 4.2.

All experiments in MNV were run with MPICH 4.2.0 and CH4:OFI (InfiniBand), while in NASP with MPICH 3.4.3 and CH3:Nemesis (Ethernet). Both the Proteo version[4] and the experimental results [17] are publicly available.

Proteo was used to execute the different approaches. Before reconfiguration, 5 iterations of Pi computation with `MPI_Allgather` were performed to ensure MPI initialization. MaM was configured with Baseline (B) and Merge (M), each with both strategies described in 4, or Merge without strategies. For shrinking, only Baseline plus the described techniques was compared against a stand-alone Merge. Each configuration was executed 20 times, and the median result was reported for every triplet of *sources*, *targets*, and method.

5.2. Homogenous System Evaluation

On the homogeneous cluster MNV, both strategies described in Section 4 were evaluated. In total, 5 configurations were tested for expansion and 3 for shrinking. Each configuration involved a single reconfiguration from NS to NT processes, using up to 42 combinations of nodes from the set $1, 2, 4, 8, 16, 24, 32$.

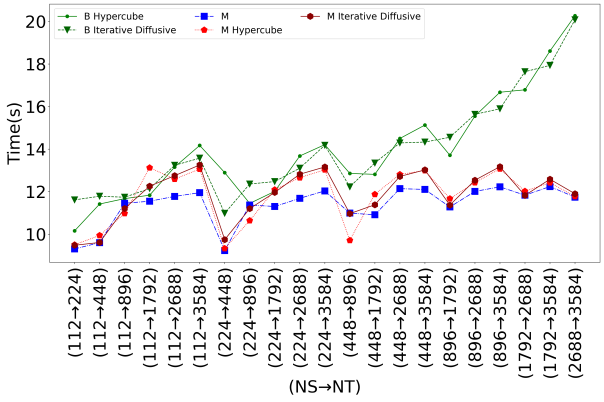
Figure 4 reports the median reconfiguration times depending on the number of sources (NS) and targets (NT). The top Figure 4a shows expansion results, where the Merge method outperforms all others in 17 out of 21 cases (80.9%). Parallel Merge follows closely, requiring up to 1.13× more time due to additional synchronizations among groups. This overhead grows when more than 8 groups are created and the number of groups is not a power of two. In those cases, at least three steps are required for the binary connection, and the presence of unbalanced leaves increases the cost.

Parallel Baseline methods are consistently slower (up to 1.73×), except in one case (1 node to 16 nodes). This

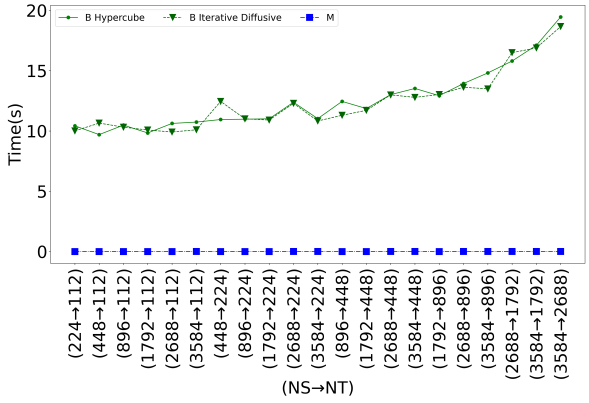
[2] https://www.bsc.es/supportkc/docs/MareNostrum5/overview

[3] https://www.hpca.uji.es/computing-facilities/

[4] https://lorca.act.uji.es/gitlab/martini/malleability_benchmark/-/tree/Paper-ParallelSpawn



(a) Expansion.



(b) Shrinkage.

Figure 4: Resize times in homogenous system.

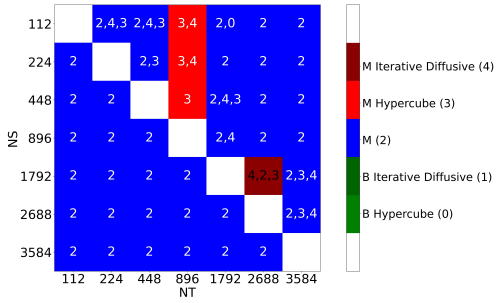


Figure 5: Median reconfiguration times in MNV.

overhead stems from spawning extra processes, which in addition leads to oversubscription.

The bottom Figure 4b shows shrinking results. Shrinking without spawning processes is clearly advantageous, completing with a speedup of 1387× over the other evaluated approaches. These findings are consistent with previous work [20].

Figure 5 summarizes the best method for each (NS, NT) pair based on statistical tests. Axes are defined with NS on the vertical and NT on the horizontal; thus, the upper triangle corresponds to expansion and the lower triangle to shrinking. When multiple methods appear in a cell, they are statistically equivalent, ordered by ascending time.

The main difference arises in expansions with 8 or fewer groups, requiring at most 3 steps. In such cases, the parallel methods are preferred as they enable shrinking without spawning processes, while maintaining times comparable to Merge, though statistically higher.

5.3. Improved Method Evaluation

On the heterogeneous cluster NASP, only the improved strategy from Section 4.2 was evaluated. Three configurations were tested for expansion and two for shrinking. Each configuration involved a single reconfiguration from NS to NT processes, using up to 72 combinations of nodes from the set $1, 2, 4, 6, 8, 10, 12, 14, 16$. In all cases, the number of nodes of each type was balanced (half of one type and half of the other). When only one node was used, the 20-core node was selected.

Figure 6 shows the median reconfiguration times as a function of NS and NT. The top Figure 6a reports expansion results, where the Merge method consistently outperforms the others. The iterative diffusive variant of Merge achieves comparable results, with at most a 1.25× increase, but its cost grows with the number of generated groups.

Baseline methods are always more expensive, ranging from 1.76× to 4.92× slower, due to oversubscription caused by spawning more processes.

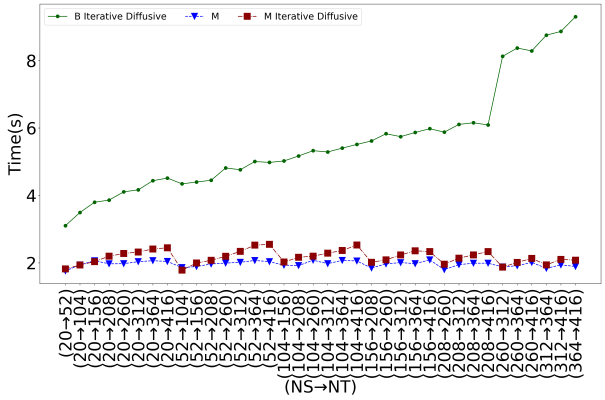
The bottom Figure 6b shows shrinking results. As in the previous evaluation, Merge is the preferred method with a minimum speedup of 20×. To ensure it can be applied, a parallel spawn is required. This approach also benefits future shrink operations, adding only a minor overhead during expansion. However, in difference to the previous evaluation, in 28 out of 32 cases (87.5%), standalone Merge delivered statistically lower times than its iterative diffusive variant.

6. Conclusions and Future Work

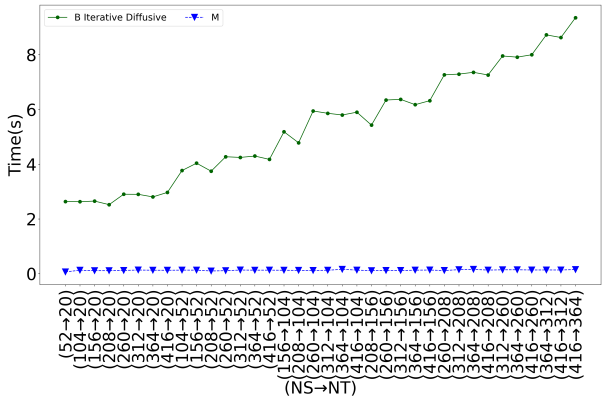
This work has introduced and evaluated two new strategies for spawning processes in the traditional MPI. Both approaches rely on parallel spawning with two main goals: (1) reducing the cost of process creation, and (2) enabling efficient shrink operations with the Merge method, the fastest available option. Achieving the second goal requires for each MCW to be fully contained within a node. Previously, this was not possible: when shrinking, processes that should be terminated remained as zombies because other ranks of the same MCW were still active on different nodes, preventing node reuse. The proposed strategies resolve this issue, enabling fast shrink operations.

Between the two strategies, the Hypercube approach is limited to homogeneous, non-shared systems, while the Iterative Diffusive strategy also applies to heterogeneous or shared environments, at the cost of additional computation and memory overhead.

Experimental results on both homogeneous and heterogeneous systems show that, in the worst case, the parallel



(a) Expansion.



(b) Shrinkage.

Figure 6: Resize times in heterogenous system.

approaches require up to 1.25× the time of the previous best expansion technique, while enabling shrink operations that are at least 20× faster. In summary, this work provides a practical solution for enabling fast shrink operations without compromising expansion performance.

Future work includes study how to further reduce the synchronization (Subsection 4.3) and connection (Subsection 4.4) overheads, with the aim of outperforming even the basic Merge in expansions. Another promising direction is the design of data redistribution strategies that minimize transfers, ensuring that processes after resizing retain as much of their data as possible.

Declaration of competing interest

The authors have no conflicts of interest to declare that are relevant to the content of this article.

Funding sources

The researchers from UJI have been funded by the project PID2023-146569NB-C22, 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.

References

[1] Aliaga, J.I., Castillo, M., Iserte, S., Martín-Álvarez, I., Mayo, R., 2022. A Survey on Malleability Solutions for High-Performance Distributed Computing. Applied Sciences 12. URL: https://www.mdpi.com/2076-3417/12/10/5231, doi:10.3390/app12105231.

[2] Bernholdt, D.E., Boehm, S., Bosilca, G., Gorentla Venkata, M., Grant, R.E., Naughton, T., Pritchard, H.P., Schulz, M., Vallee, G.R., 2020. A Survey of MPI Usage in the US exascale computing project. Concurrency and Computation: Practice and Experience 32, e4851. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4851, doi:https://doi.org/10.1002/cpe.4851, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4851. e4851 cpe.4851.

[3] Cascajo, A., Arbe, A., Garcia-Blas, J., Carretero, J., Singh, D.E., 2023. Malleable Techniques and Resource Scheduling to Improve Energy Efficiency in Parallel Applications, in: High Performance Computing, Springer Nature Switzerland, Cham. pp. 16–27.

[4] Chadha, M., John, J., Gerndt, M., 2020. Extending SLURM for dynamic resource-aware adaptive batch scheduling. CoRR abs/2009.08289.

[5] Feitelson, D.G., 1996. Packing schemes for gang scheduling, in: Lecture Notes in Computer Science book series (LNCS, volume 1162). Springer, Berlin, Heidelberg, Heidelberg, Germany, pp. 89–110.

[6] Hori, A., Jeannot, E., Bosilca, G., Ogura, T., Gerofi, B., Yin, J., Ishikawa, Y., 2021. An international survey on mpi users. Parallel Computing 108, 102853. doi:https://doi.org/10.1016/j.parco.2021.102853.

[7] Huber, D., Iserte, S., Schreiber, M., Peña, A.J., Schulz, M., 2025. Bridging the gap between genericity and programmability of dynamic resources in hpc, in: ISC High Performance 2025 Research Paper Proceedings (40th International Conference), pp. 1–11.

[8] 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, Association for Computing Machinery, New York, NY, USA. p. 57–67. URL: https://doi.org/10.1145/3555819.3555856, doi:10.1145/3555819.3555856.

[9] Iserte, S., 2018. High-throughput Computation through Efficient Resource Management. Ph.D. thesis. Universitat Jaume I. Castelló de la Plana.

[10] Iserte, S., Martín-Álvarez, I., Rojek, K., Aliaga, J.I., Castillo, M., Folwarska, W., Peña, A.J., 2026. Resource optimization with mpi process malleability for dynamic workloads in hpc clusters. Future Generation Computer Systems 174, 107949. URL: https://www.sciencedirect.com/science/article/pii/S0167739X25002444, doi:https://doi.org/10.1016/j.future.2025.107949.

[11] Iserte, S., Rojek, K., 2019. A Study of the Effect of Process Malleability in the Energy Efficiency on GPU-based Clusters. Journal of Supercomputing , 1–20.

[12] Li, J., Michelogiannakis, G., Cook, B., Cooray, D., Chen, Y., 2023. Analyzing Resource Utilization in an HPC System: A Case Study of NERSC's Perlmutter, in: High Performance Computing, Springer Nature Switzerland, Cham. pp. 297–316.

[13] Lucas, R., Ang, J., Bergman, K., Borkar, S., Carlson, W., Carrington, L., Chiu, G., Colwell, R., Dally, W., Dongarra, J., et al., 2014. Doe advanced scientific computing advisory subcommittee (ascac) report: Top ten exascale research challenges. URL: https://www.osti.gov/biblio/1222713, doi:10.2172/1222713.

[14] Martín, G., Marinescu, M.C., Singh, D.E., 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.

[15] Martín Álvarez, I., Aliaga, J.I., 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, Association for Computing Machinery, New York, NY, USA. p.

416–426. URL: https://doi.org/10.1145/3624062.3624110, doi:10.1145/3624062.3624110.

[16] Martín Álvarez, I., Aliaga, J.I., Castillo, M., Iserte, S., 2025. MaM: A User-Friendly Interface to Incorporate Malleability into MPI Applications, in: Euro-Par 2024: Parallel Processing Workshops, Springer Nature Switzerland, Cham. pp. 346–358. URL: https://doi.org/10.1007/978-3-031-90200-0_28.

[17] Martín Álvarez, I., 2025. Proteo dataset (2025) for an study for parallel spawning in dynamic resources. URL: https://doi.org/10.5281/zenodo.17315810.

[18] Martín-Álvarez, I., Aliaga, J.I., Castillo, M., 2025. Dynamic reconfiguration for malleable applications using RMA. URL: https://arxiv.org/abs/2509.05248, arXiv:2509.05248.

[19] Martín-Álvarez, I., Aliaga, J.I., Castillo, M., Iserte, S., 2024a. Proteo: A framework for the generation and evaluation of malleable MPI applications. The Journal of Supercomputing , 23083–23119URL: https://doi.org/10.1007/s11227-024-06277-5.

[20] Martín-Álvarez, I., Aliaga, J.I., Castillo, M., Iserte, S., Mayo, R., 2024b. Dynamic Spawning of MPI Processes Applied to Malleability. The International Journal of High Performance Computing Applications 38, 69–93. URL: https://doi.org/10.1177/10943420231176527, doi:10.1177/10943420231176527, arXiv:https://doi.org/10.1177/10943420231176527.

[21] MPICH Development team, . MPICH Website. URL: https://www.mpich.org/.

[22] Muñnoz, J.F., Cascajo García, A., Pérez, J.C., 2023. Dynamic management of processes and communicators in malleable mpi applications, in: 2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS), pp. 848–855. doi:10.1109/ICPADS60453.2023.00127.

[23] Sanchez-Gallegos, G., Garcia-Blas, J., Petre, C., Carretero, J., 2023. Malleable and Adaptive Ad-Hoc File System for Data Intensive Workloads in HPC Applications, in: High Performance Computing, Springer Nature Switzerland, Cham. pp. 56–67.

[24] Tarraf, A., Schreiber, M., Cascajo, A., Besnard, J.B., Vef, M.A., Huber, D., Happ, S., Brinkmann, A., Singh, D.E., Hoppe, H.C., Miranda, A., Peña, A.J., Machado, R., Gasulla, M.G., Schulz, M., Carpenter, P., Pickartz, S., Rotaru, T., Iserte, S., Lopez, V., Ejarque, J., Sirwani, H., Wolf, F., . Malleability in Modern HPC Systems: Current Experiences, Challenges, and Future Opportunities. IEEE Transactions on Parallel and Distributed Systems , 1–14.