Journal Title
XX(X):1–19
©The Author(s) 0000
Reprints and permission:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

Dynamic spawning of MPI processes applied to malleability

Iker Martín-Álvarez[1], José I. Aliaga[1], Maribel Castillo[1], Sergio Iserte[2] and Rafael Mayo[1]

Abstract

Malleability allows computing facilities to adapt their workloads through a resource management systems (RMS) to maximize the throughput of the facility and the efficiency of the executed jobs. This technique is based on reconfiguring a job to a different resource amount during execution and then continuing with it. One of the stages of malleability is the dynamic spawning of processes in execution time, where different decisions in this stage will affect how the next stage of data redistribution is performed, which is the most time-consuming stage. This paper describes different methods and strategies, defining eight different alternatives to spawn processes dynamically and indicates which one should be used depending on whether a strong or weak scaling application is being used. In addition, it is described for both types of applications which strategies benefit most the application performance or the system productivity. The results show that reducing the number of spawning processes by reusing the older ones can reduce reconfiguration time compared to the classical method by up to 2.6 times for expanding and up to 36 times for shrinking. Furthermore, the asynchronous strategy requires analysing the impact of oversubscription on application performance.

Keywords

Process spawning, MPI, Application reconfiguration, Malleability, Threading

1 Introduction

High performance computing (HPC) facilities need the application of novel programming techniques to fully utilise the large number of processors interconnected via high-speed networks. The main goal of HPC, particularly in exascale supercomputers, is to use specialised techniques to maintain a high rate of productivity in the system in terms of jobs completed per unit of time.

Large computing facilities usually include resource management systems (RMS), which monitor available resources and allocate them when users launch new jobs. Therefore, when a job asks the RMS for the resources necessary for its complete execution, its allocation should be different depending on whether the main goal is to end the execution as soon as possible or to improve the productivity in the system. Usually, from the application's point of view, the shortest execution times occur when more resources are assigned, although not all of them are always used during its execution, whereas higher system productivity is obtained when all resources are used most of the time. To combine both goals, RMS must assign the optimal number of resources in each step of the job, achieving a trade-off between the execution time of the applications and the productivity of the system.

In this regard, malleability allows applications to modify the initially allocated computational resources, while the job is running. The benefits of its usage can be analysed from two different points of view. For every single application, the benefit can derive from the increase of its performance when the job obtains more resources, whereas, for the global system, the benefit can derive from the increase of the throughput with the reduction of the makespan.

The application of malleability has shown an approximately 20% reduction of the makespan in Posner and Fohry (2021). Furthermore, Iserte et al. (2020) demonstrated the ways to use it to reduce makespan by approximately 4x when combined with malleability techniques, whereas Iserte and Rojek (2019) showed its impact on energy efficiency improving approximately 2.4x in GPU-capable workloads.

To date, malleability is infrequently used by applications, as described in Hori et al. (2021) and Bernholdt et al. (2018). Although checkpoint/restart (C/R) techniques exhibit similar behaviour and can ease the adoption of malleability in codes, they have exceedingly high reconfiguration times Iserte et al. (2016).

In this study, we considered malleability as the capability of a distributed MPI parallel job to modify the number of processes without stopping its execution, varying the computational resources initially assigned to the job as many times as it is required and without storing application data on disk.

Malleability is applied to specific points of the execution where processes are synchronized in a checkpoint. For iterative applications, defining a single checkpoint will be the most common option, whereas for non-iterative ones, defining a checkpoint at the beginning of each phase will be a good

[1]Universitat Jaume I, *Dept. Ing. y Ciencia de los Computadores*, Castelló, Spain
[2]Barcelona Supercomputing Center, *Dept. of Computer Science*, Spain

Corresponding author:
Iker Martín-Álvarez, Universitat Jaume I, Dept. Ing. y Ciencia de los Computadores, Castelló, Spain.
Email: martini@uji.es

alternative. The first task to execute in the checkpoints is to contact the RMS to determine if the application should be reconfigured because RMS is responsible for making this decision. If the RMS proposes to apply malleability, the following tasks are to be performed:

1. *Resources reallocation*. The RMS allocates new and/or relinquishes assigned resources to/from a job.
2. *Processes management*. Spawn/terminate processes according to the RMS reconfiguration decision.
3. *Data redistribution*. Communicate data among initial and new processes, such that the execution continues properly.

This paper is focused only on stage 2, probing different methods to spawn and terminate MPI processes, and analysing their impact on the performance of malleable applications. In the near future, another study will extend this analysis to the stages 1 and 3, finding the best combination of methods to apply malleability in a job.

The main contributions of the paper can be summarized as follows:

- Two methods are presented for spawning processes:
  - *Baseline*, where any malleability action spawns new processes; and
  - *Merge* which always attempts to reuse the old processes, reducing the number of spawned ones.
- These methods can be applied with different strategies:
  - *Asynchronous*: POSIX threads are responsible for spawning processes in the background (in contrast to the synchronous operation, where processes wait for completing the reconfiguration).
  - *Single*: Only one process is involved in the spawning (in contrast to the original collective spawn).
- All methods were evaluated using a synthetic application that defined two designed benchmarks, simulating weak and strong scaling applications.

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 apply malleability in MPI. Section 4 shows 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

Several strategies of spawning and terminating MPI ranks in reconfigurable jobs have been implemented by different malleability solutions. This section reviews these approaches along with the frameworks in which they are implemented.

On the one hand, C/R strategies for malleability base their processes reconfiguration on launching NC new processes (children). Some examples are the Process Checkpointing and Migration (PCM) API El Maghraoui et al. (2006), Scalable C/R (SCR) Moody et al. (2010) or Stop Restart Software (SRS) Vadhiyar and Dongarra (2002). For this purpose, NP old processes (parents) store the data in a disk and terminate their execution. Then, children are spawned and they load the data from the disk. This approach always spawns all children both to expand and shrink the number of processors.

A specific solution for an MPI Computational Fluid Dynamics (CFD) application is found in Houzeaux et al. (2021), which employs COMP Superscalar (COMPSs) Badia et al. (2015), along with the Tracking Application Live Performance (TALP) library Lopez et al. (2021), to allow performance-aware malleability. COMPSs exploit parallelism directly from sequential code, allowing users to avoid any issue related to concurrency. The CFD-modified application uses C/R techniques, which leverage COMPSs to reconfigure the application. Reconfigurations occur only if the TALP library indicates that it will provide a reduction in execution time while expanding or similar efficiency with fewer resources. When a reconfiguration occurs, the application creates a checkpoint, saving the data to the filesystem, and the application is restarted with the new number of processes and the data is distributed accordingly.

On the other hand, recently, there have been in-memory solutions for malleability, which avoid disk usage during the reconfiguration.

In Iserte et al. (2020), the Dynamic Management of Resources (DMR) framework is introduced, which implements malleability using the OmpSs programming model[1] on the runtime Nanos++[2]. This is a distributed parallel runtime based on MPI, responsible for handling processes and their communication. DMR expands jobs by spawning all the children in a new communicator from scratch, as presented in Iserte et al. (2018). Once data are received by the NC children, the NP parent processes are terminated. As a result, before this termination, there are $NP + NC$ processes during the data redistribution stage. Nevertheless, DMR shrinks jobs by terminating processes if the ids are equal to or greater than NC. In this case, some parents are terminated, rather than spawning any children. This termination occurs when the data is redistributed among the parents to continue the execution of NC processes.

Furthermore, Iserte et al. (2017) presented an asynchronous version of DMR that schedules reconfiguration actions while applications execute their computations. Thus, the runtime already knows if an expansion, shrinkage, or no action has to be performed in the malleability point before asking to the RMS since the action has been programmed in the previous request. Notice that the asynchronous version produces more action abortions than the synchronous because the continuous cluster status changing.

Authors in Lemarinier et al. (2016) present a reconfiguration technique for MPI applications based on the User Level Failure Mitigation (ULFM) MPI Bland et al. (2013), which supports the use of the standard *MPI_Comm_spawn* routine and dynamic removal of processes. In this solution, jobs are expanded by spawning new processes and merging them into the main communicator. Because the collective operations do not return uniformly in the presence of unexpected process termination, ULFM counts with the *MPI_Comm_shrink*

routine that creates a new communicator from the original communicator in which unwanted processes are excluded.

Flex-MPI is a performance-aware reconfiguration library Martín et al. (2015). Processes in Flex-MPI are divided into two types: *initial*, processes spawned when launching the application; and *dynamical*, processes created during execution time. Flex-MPI does not allow initial processes to be terminated during the execution. Therefore, only dynamical processes after an expansion can be shrunk. Furthermore, dynamic processes are created/terminated one at a time. In the case of an expansion, each new process is created in a new communicator connected to the main group of processes, which in turn, is merged into an intra-communicator using the *MPI_Intercomm_merge* routine.

Authors in Comprés et al. (2016) developed an infrastructure for elastic execution of MPI applications relying on Slurm[3] and MPICH[4]. This approach introduces an adaptive mode in which the execution must be explicitly initialized. Then, the reconfiguration operation provides two communicators as output: one communicator that is equivalent to the one provided by spawn routines, and another communicator that provides an early view of the future *MPI_COMM_WORLD* communicator. For shrinking, processes to be removed will not have access to the future *MPI_COMM_WORLD*, setting their communicator to *MPI_COMM_NULL*. Finally, pre-existing and new processes are joint in *MPI_COMM_WORLD*.

In summary, merging communicators for expanding a job is presented as the most adopted solution. However, no consensus exists during shrinking because each framework solves it differently.

Existing solutions rarely use asynchronous MPI methods, although Wittmann et al. (2013) demonstrated that the use of asynchronous primitives in MPI could reduce the execution time by overlapping communications with computation. This technique can also be used to overlap reconfigurations with computation.

The authors in Aliaga et al. (2022) describe a more extensive malleability state-of-the-art including less related works.

3 Methods to dynamically reconfigure MPI applications

This section analyses different methods that allow varying the number of processes during malleable executions.

We assumed that initially NP processes (parents) execute an application and, at any given time, this number is modified, such that NC processes (children) execute the application, where NC can be greater or lower than NP (expand or shrink, respectively).

Without deviating from the standard MPI scope Message Passing Interface Forum (2021), we considered different alternatives to complete the reconfiguration stage of a job reconfiguration. Particularly, this section describes different methods/strategies to set a new layout of MPI ranks, mapping new MPI ranks to nodes or cancelling some of the active processes, while the job is running. About the methods, we consider the following:

- Baseline. A basic method to spawn the required processes.
- Merge. A more complex method in which the number of spawned processes creating new communicators is reducing, reusing, and merging existing communicators.

These methods can be used along with any combination of the following strategies:

- Determining the number of parents involved in the spawn operation: only one, or all collectively.
- Spawning processes synchronously or asynchronously.

Therefore, each method has up to four different ways of spawning processes as they can be used with no strategies, one of them, or both.

These methods are mainly based on the MPI routine *MPI_Comm_spawn*, which is a collective operation on the specified communicator. The definition of this routine, which appears in Message Passing Interface Forum (2021), is as follows:

```
int MPI_Comm_spawn(const char *command,
 char *argv[], int maxprocs, MPI_Info
 info, int root, MPI_Comm comm, MPI_Comm
 *intercomm, int array_of_errcodes[])
```

where:

- `command` and `argv` include the name of the program to be executed on the new processes and the corresponding arguments, respectively.
- `maxprocs` contains the number of processes to be spawned.
- `info` provides additional information to the processes, using user-specified (key, value) pairs.
- `root` determines the MPI rank, which supplies the previous arguments to the routine. For the other processes in the communicator, the value of these parameters is ignored.
- `comm` is the intra-communicator of the calling group, the parents.
- `intercomm` is the inter-communicator defined between the parents and the newly spawned processes.
- `array_of_errcodes` is an integer vector, with a size equal to `maxprocs`. Each component informs whether the launch of the corresponding process has been successful or not.

We note that only the last two are output parameters, whereas the others are input parameters.

As it is reported in Comprés et al. (2016), the routine included in MPI Standard has the following issues:

- It is a synchronous operation for all the processes involved in the invocation, parents, and newly spawned processes.
- It produces an inter-communicator based on disjointed processes groups: one for the parents and the other for the newly spawned processes.
- Subsequent creation of processes results in multiple process groups. Communication between them is not straightforward to manage.
- Processes can only be terminated on the entire process group. Therefore, processes in a group are not destroyed until all of them invoke `MPI_Finalize`.

```
1 void baseline_parents(char *cmd, int root,
2     MPI_Comm intracomm, MPI_Comm *newcomm) {
3   char *hostlist;
4   int hostlist_size, spawn_method=BASELINE;
5   MPI_Info info;
6
7   // Calculates where each new process will be mapped,
8   //   returning the list of processes and its size.
9   calculate_physical_distribution( &hostlist,
10    &hostlist_size, spawn_method);
11  MPI_Info_create(&info);
12  MPI_Info_set(info, "hosts", hostlist);
13  MPI_Comm_spawn(cmd, MPI_ARGV_NULL, hostlist_size,
14    info, root, intracomm, newcomm,
15    MPI_ERRCODES_IGNORE);
16  MPI_Info_free(&info);
17
18 // Additional spawn operations for new processes
19 void baseline_children(MPI_Comm *new_comm) {
20   MPI_Comm_get_parent(&new_comm);
21 }
```

Listing 1: Basic skeleton of the Baseline method.

– By default, processes created with spawn operations are run in the same resource allocation.

These features must be considered in the development of the different spawning methods.

3.1 Baseline method

The Baseline reconfiguration is defined as a synchronous method where all NP parents are involved in the spawn of NC children.

In this method, a call to `MPI_Comm_spawn` is executed to spawn NC new processes. The `MPI_Info` argument will propose the mapping of each new MPI rank to the target hosts, although the final decision will be made by the RMS in production systems. Moreover, the returned inter-communicator by the MPI routine allows communication between parents and children. For malleability, once this operation is completed, parents terminate their execution, whereas children will continue with the application.

Figure 1a represents the stages of this method during the execution of an iterative application, which is reconfigured after the first iteration. Therefore, it starts with NP processes running *It0*, then new NC processes are spawned, which continue in *It1*. Notably, before terminating the parents, $NP + NC$ processes are running.

Listing 1 shows a pseudo code on the functioning of this method. Mapping is performed at L7-L10, whereas children are spawned at L13-L15 along with the creation of the inter-communicator `newcomm`. We note that `intracomm` should be the communicator defined by the NP parents.

The theoretical computational time of this method (RT_S) is calculated by using (1),

$$RT_S = T\_Spw(NP, NC), \quad (1)$$

where *T_Spw*(NP,NC) is the consumed time by `MPI_Comm_spawn` to spawn NC processes.

NP and NC are included in the equation because it has been experimentally proved that *T_Spw* depends on both.

3.2 Merge method

The Merge method, which was proposed in Radcliffe et al. (2011), reconfigures a job by creating only the necessary processes or removing those that are no longer required. Therefore, two different options are considered:

– Expanding (when $NP < NC$), which generates $NC - NP$ new processes

– Shrinking (when $NP > NC$), which suspends $NP - NC$ processes.

In both cases, it is considered that the mapping of the surviving parents is not changed.

3.2.1 Expanding a job

In this case, `MPI_Comm_spawn` is configured to create $NC - NP$ new processes. All further communications in the application should be conducted using a new intra-communicator, which includes both the parents and new spawned processes. This intra-communicator can be created from the parents' communicator and the returned inter-communicator, using the MPI routine `MPI_Intercomm_merge`, which is defined as follows:

```
int MPI_Intercomm_merge(MPI_Comm intercomm,
  int high, MPI_Comm *newintracomm)
```

In this definition, the parameter `high` determines the group of processes that are numbered first and last.

Therefore, the steps to perform this method are the following:

1. Set physical mapping for $NC - NP$ processes.
2. Spawn the $NC - NP$ processes.
3. Execute `MPI_Intercomm_merge` to join parents and new processes into a single communicator.

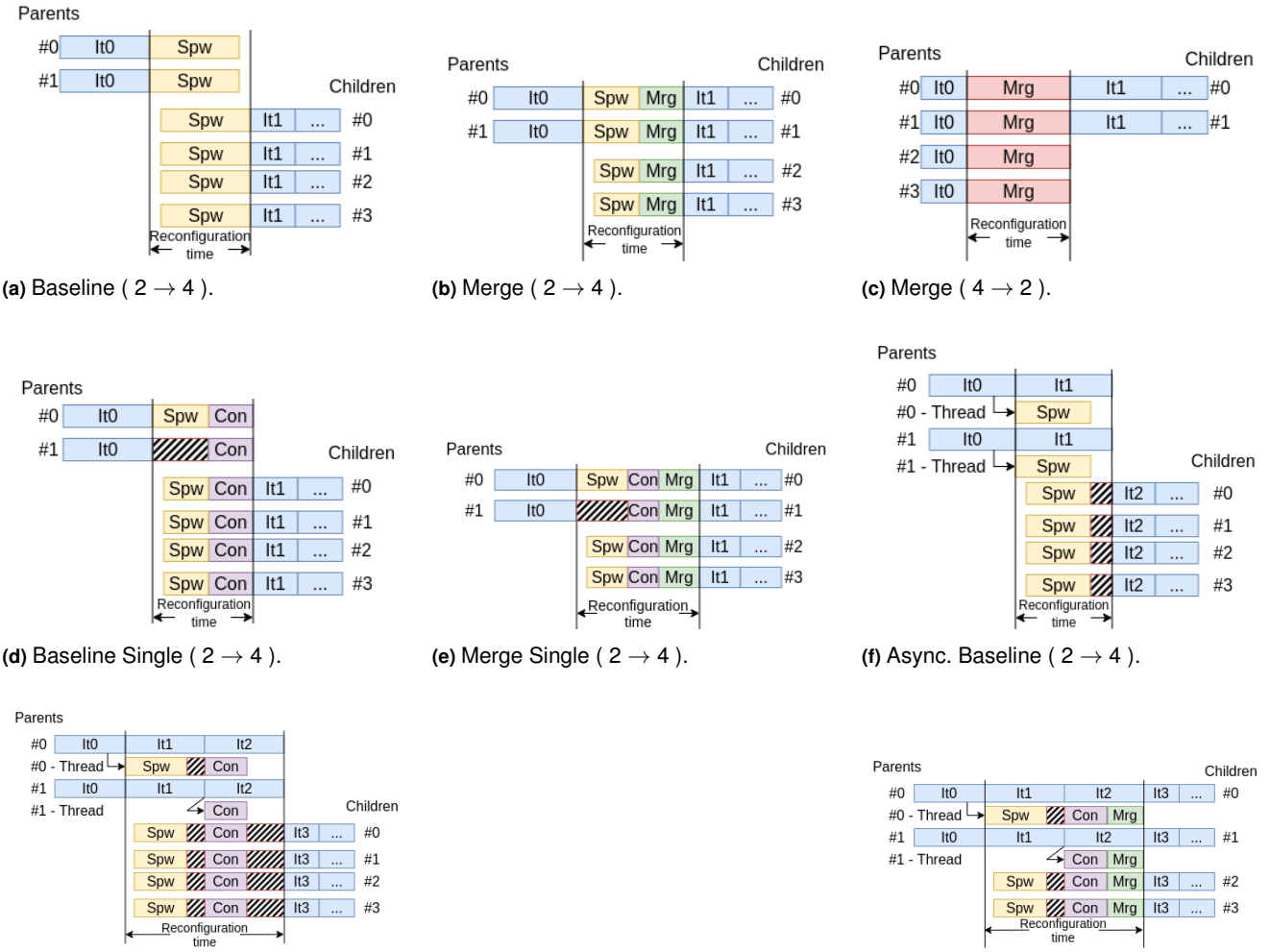
Figure 1b represents the expansion of an iterative application using this method. In addition, the reconfiguration starts at the end of *It0*, involving the spawn (*Spw*) and the communicators merge (*Mrg*) stages. Finally, old and new processes continue running the next iteration (*It1*).

Listing 2 shows a pseudo code for the way of expansion with the Merge method. From the parents' point of view, this pseudo code is close to the Baseline method, because only some additional operations, marked in blue (L4, L6, and L17), are added to join old and new processes into the new intra-communicator; subsequently, the inter-communicator is freed. A similar conclusion can be drawn from the point of view of new processes, because the additional operations, marked in blue operations (L23-L26), only focus on merging both communicators. We note that the second parameter of `MPI_Intercomm_merge` determines that parents are numbered before new processes.

The theoretical computational time of this method (RT_S) is calculated by using (2),

$$RT_S = T\_Spw(NP, NC - NP) + T\_Mrg(NC), \quad (2)$$

where *T_Spw* is the time required by `MPI_Comm_spawn` to spawn ($NC - NP$) processes, and *T_Mrg* is the time required to combine both groups into a single one with NC processes.



(a) Baseline (2 → 4). **(b)** Merge (2 → 4). **(c)** Merge (4 → 2).

(d) Baseline Single (2 → 4). **(e)** Merge Single (2 → 4). **(f)** Async. Baseline (2 → 4).

(g) Async. Baseline Single (2 → 4). **(h)** Async. Merge Single (2 → 4).

Figure 1. Functioning of different reconfiguration methods. In each subplot, the groups of horizontal blocks refer to a process, showing the executed iteration (ItX) and the main operations: spawn (Spw), connect (Con), and merge (Mrg). Moreover, striped blocks refer to the time wasted by process/thread.

```
1 void merge_expand_parents(char *cmd, int root,
2     MPI_Comm intracomm, MPI_Comm *new_comm) {
3   char *hostlist;
4   int hostlist_size, spawn_method=MERGE;
5   MPI_Info info;
6   MPI_Comm intercomm;
7
8   // Calculates where each new process will be mapped,
9   //   returning the list of processes and its size.
10  calculate_physical_distribution( &hostlist,
11    &hostlist_size, spawn_method);
12  MPI_Info_create(&info);
13  MPI_Info_set(info, "hosts", hostlist);
14  MPI_Comm_spawn(cmd, MPI_ARGV_NULL, hostlist_size,
15    info, root, intracomm, newcomm,
16    MPI_ERRCODES_IGNORE);
17  MPI_Intercomm_merge(intercomm, 0, new_comm);
18  MPI_Info_free(&info);
19 }
20
21 // Additional spawn operations to merge new processes
22 void merge_new_procs(MPI_Comm *new_comm) {
23  MPI_Comm intercomm;
24  MPI_Comm_get_parent(&intercomm);
25  MPI_Intercomm_merge(intercomm, 1, new_comm);
26  MPI_Comm_free(&intercomm);
27 }
```

Listing 2: Basic skeleton of the Merge Expand method.

3.2.2 *Shrinking a job*

In this case, only NC parents must be maintained in the application. Therefore, the MPI routine `MPI_Comm_split` is used to create a new intra-communicator with only NC processes. The definition of this routine is shown below,

```
int MPI_Comm_split(MPI_Comm comm, int color,
  int key, MPI_Comm *newcomm),
```

in which the parameter `color` is used to divide the current communicator. For shrinking, only two groups are considered: NC processes to be stayed and $NP - NC$ processes to be suspended.

The discarded processes are no longer needed and they become *zombies*, releasing their resources and suspending their execution until `MPI_Finalize`. It is important to suspend these processes until all other processes in the same communicator complete their work and execute the `MPI_Finalize` routine, avoiding a busy-wait loop that can disturb the execution of active processes. Notice that all processes in the same communicator have to be removed at the same time calling `MPI_Finalize`. For this purpose, the root process must wake up zombies to allow all processes to finalize the MPI environment before `MPI_Finalize` is

invoked by active processes. Therefore, the steps to complete this method are as follows:

1. Execute `MPI_Intercomm_split` to divide the current group of processes into two groups: NC actives and $NP - NC$ *zombies*.

2. Root process obtains the PID and the node name of the *zombies* using `MPI_Gather`.

3. Zombies move to a deep sleep state, whereas the active processes continue running.

4. Before the program ends, the root process awakens the *zombies*, allowing all processes to finalize the MPI environment.

Various alternatives exist to suspend processes, each of them with different overheads. We propose to use the system-call `sigsuspend` to allow zombie processes to move into a deep sleep state where they do not consume resources. Additionally, different factors about the ways the root process wakes up *zombies* must be considered.

First, all active processes must conclude their computation before the *zombies* wake up to avoid execution overheads by a busy-wait. A solution is to synchronize active processes in a `MPI_Barrier` just before the root starts to wake up the *zombies*.

Second, *zombie* processes are waking up only to finalize their execution, therefore, it might be better to directly force their cancellation. If all *zombies* are on the same node as the root process, it could send them a system-call `kill`, and then all active processes execute `MPI_Finalize`. However, if a *zombie* is on a different node from the root, the best alternative is to execute `MPI_Abort`, which, in most implementations, terminates all processes.

Figure 1c shows the Merge Shrink method in an iterative execution, where the label *Mrg* refers to the shrink operation.

Listing 3 shows a pseudo code about the use of this method to perform shrinking. The `Gather_zombie_info` routine allows the root process to obtain the PID and node name of all the *zombies*. Moreover, `release_somehow_resources` releases the local resources of a process, whereas `enter_deep_sleep` suspends the process until the root wakes it up.

```
1 #define ZOMBIE 0
2 #define ACTIVE 1
3
4 void merge_shrink(MPI_Comm intracomm,
5     MPI_Comm *newcomm, int rank, int NC) {
6   int situation = ZOMBIE;
7
8   if(rank < NC) situation = ACTIVE;
9
10  MPI_Comm_split(intracomm, situation, rank, newcomm);
11
12  Gather_zombie_info();
13
14  if(situation == ZOMBIE) {
15    release_somehow_resources();
16    enter_deep_sleep();
17  }
18 }
```

Listing 3: Basic skeleton of the Merge Shrink method.

Next, we show the theoretical computational time of this method (RT_S) by using (3),

$$RT_S = T\_split(NP) + T\_gather(NP), \qquad (3)$$

where *T_split* is the time required to divide the communicator into two groups, and *T_gather* is the time required to gather information about the *zombies*. The computation costs are shown in the order in which the operations are executed, and their overhead mainly depends on the number of parents.

3.3 Single strategy

This strategy restricts the spawning operation to be executed on a single process and can be applied to both the Baseline and Merge methods.To achieve this, only the root of the parents invokes `MPI_Comm_spawn` from the `MPI_COMM_SELF` communicator, whereas the other parents wait for the reconfiguration completion. Moreover, an additional step is needed to connect the $NP - 1$ non-root parents to the newly spawned processes because initially only the root is connected to the children via an inter-communicator.

This new step can be summarized as follows:

1. The routine `MPI_Open_port` is used by the root of new processes to open a communication port. Its definition is shown below:

   ```
   int MPI_Open_port(MPI_Info info, char
     *port_name)
   ```

2. Root of new processes sends the port name returned by `MPI_Open_port` to the root of parents through the inter-communicator.

3. All involved processes must be connected to this port: parents invoke `MPI_Comm_connect`, whereas children use `MPI_Comm_accept`. The definition of both routines is as follows:

   ```
   int MPI_Comm_connect(const char
     *port_name, MPI_Info info, int root,
     MPI_Comm intracomm, MPI_Comm
   *newcomm),
   int MPI_Comm_accept(const char
     *port_name, MPI_Info info, int root,
     MPI_Comm intracomm, MPI_Comm
   *newcomm),
   ```

   where `port_name` and `info` in both routines are required only for the root of parents or root of new processes. The output parameter `newcomm`, which will contain the new inter-communicator, is required by all processes. Notably, `intracomm` is the intra-communicator in which parents or children are respectively defined in each function.

4. Finally, the root of parents and all new processes free the inter-communicator returned by `MPI_Comm_spawn`.

The theoretical advantage of this approach is to eliminate the synchronization among parents to perform the spawn operation because it is only performed by the root. Therefore, if the cost of this synchronization is greater than the

mentioned additional operations, this strategy can reduce its reconfiguration time.

Additionally, this strategy can be applied for expanding or shrinking in the Baseline method, whereas only for expanding in the Merge method. Notice that the single strategy has only effect when spawning new processes, which is not done when shrinking ranks.

Figures 1d and 1e show the two expanding methods. In these figures, we can observe that a new operation is required in both cases to connect all parents to the inter-communicator labeled as *Con*.

Listing 4 includes a pseudo code showing the application of the Single strategy on the Baseline method. The coloured lines show the modifications concerning the original method focused mainly on joining all processes in an inter-communicator, particularly L14, L17-L21, and L24-L27 for parents, as well as L37-L43 for new processes. Lines marked in red (L24-L25 and L42-L43) emphasize the communication operations necessary to connect parents and children.

```
1 void baseline_single_parents(char *cmd, int rank,
2     int root, int root_chd, MPI_Comm intracomm,
3     MPI_Comm *newcomm) {
4   char *hostlist, *port_name;
5   int hostlist_size, spawn_method=BASELINE_SINGLE;
6   MPI_Info info;
7   MPI_Comm intercomm;
8
9   // Calculates where each new process will be mapped,
10  //   returning the list of processes and its size.
11  calculate_physical_distribution( &hostlist,
12    &hostlist_size, spawn_method);
13
14  if(rank == root) {
15    MPI_Info_create(&info);
16    MPI_Info_set(info, "hosts", hostlist);
17    MPI_Comm_spawn(cmd, MPI_ARGV_NULL, hostlist_size,
18        info, root, MPI_COMM_SELF, &intercomm,
19        MPI_ERRCODES_IGNORE);
20    MPI_Recv(port_name, MPI_MAX_PORT_NAME, MPI_CHAR,
21        root_chd, tag, intercomm, MPI_STATUS_IGNORE);
22    MPI_Info_free(&info);
23  }
24  MPI_Comm_connect(port_name, MPI_INFO_NULL, root,
25    intracomm, newcomm);
26  if(rank == root)
27    MPI_Comm_free(&intercomm);
28 }
29
30 // Additional spawn operations for the
31 //   single alternative
32 void baseline_single_children(MPI_Comm *newcomm,
33    int rank, int root, int root_prn) {
34  char *port_name;
35  MPI_Comm intercomm;
36  MPI_Comm_get_parent(&intercomm);
37  if(rank == root) {
38    MPI_Open_port(MPI_INFO_NULL, port_name);
39    MPI_Send(port_name, MPI_MAX_PORT_NAME, MPI_CHAR,
40        root_prn, tag, intercomm);
41  }
42  MPI_Comm_accept(port_name, MPI_INFO_NULL, root,
43    MPI_COMM_WORLD, newcomm);
44  MPI_Comm_free(&intercomm);
45 }
```

Listing 4: Basic skeleton of the Baseline Single method.

The theoretical reconfiguration time of this strategy with the Baseline method (RT_S) is calculated by using (4),

$$RT_S = T\_Spw(1, NC) + T\_Con(NP, NC), \quad (4)$$

where *T_Spw* is the time required by `MPI_Comm_spawn` to spawn NC processes with the collaboration of one parent, and *T_Con* is the time required to connect NP parents and NC children into the same inter-communicator. The operations are shown in the order they are executed.

Furthermore, the theoretical computational time when this strategy is applied for expanding using the Merge method (RT_S) is calculated by using (5),

$$RT_S = T\_Spw(1, NC - NP) + T\_Con(NP, NC - NP) + T\_Mrg(NC), \quad (5)$$

where *T_Spw* is the time required by `MPI_Comm_spawn` to spawn $NC - NP$ processes with the collaboration of 1 parent, *T_Con* is the time required to connect NP parents and $NC - NP$ new processes into the same inter-communicator, and *T_Mrg* is the time required to combine both groups (NC processes). The operations are shown in the order they are executed.

We note that the parameters in *T_Con* are different in (4) and (5). The impact of the difference in their performances will be discussed in Section 4.

3.4 Asynchronous strategy

Reconfigurations can also be performed asynchronously using auxiliary *threads*, which are responsible for spawning new processes. To achieve this, first, each parent process creates a thread to perform this task. Subsequently, while these threads perform the reconfiguration, the corresponding main threads can continue the execution of the application. In some cases, overlapping reconfiguration and computation could improve job performance, justifying the study of this strategy.

The combination of processes and threads in MPI requires the following considerations:

- `MPI_Init_thread` should be used to start the MPI environment with thread support. Its definition is shown below:

  ```
  int MPI_Init_thread(int *argc,
    char ***argv, int required,
    int *provided).
  ```

  The environment will start properly for thread management if the input parameter `required` is `MPI_THREAD_MULTIPLE`, and the returned parameter `provided` has the same value.
- POSIX thread routines will be used to manage threads, thus `pthread_create` and `pthread_exit` routines create and finalize a thread, respectively.
- To avoid conflicts between collective operations executed by different threads in the same process, auxiliary threads use the MPI routine `MPI_Comm_dup` to duplicate the main communicator. Then, the main thread in each process will use the original main communicator, whereas the auxiliary threads will use the duplicated one.
- The main and auxiliary threads in each process only communicate to detect if the auxiliaries have finalized the spawning. We use a flag shared by all threads in a process, *shared_state*, indicating if the operation has been completed. To assure a correct updating and reading, its accesses are controlled using *mutex* routines.
- The frequency with which the value of *shared_state* is consulted has an impact on the final performance. This

frequency has to be adjusted to the process creation time in order to prevent the system overload with an intensive pooling, or stagnate the execution until the next request.

This strategy is always started by parent processes creating their auxiliary thread to perform the spawning task, whereas the main thread continues the execution of the application. When the auxiliary thread finishes, it modifies the value of *shared_state*, whereas the main thread will periodically test this shared variable in the checkpoint.

Verifying that all auxiliary threads have completed the operation requires checking the value of *shared_state* in each process. The simplest option is to use a collective operation, allowing each process to send its value, and the final result indicates whether all processes have ended. Considering that *shared_state* is initiated to 0 and changed to 1 when the auxiliary thread completes the local reconfiguration, the proposal is to use the MPI routine `MPI_Allreduce`, computing the minimum of this variable in all processes.

Therefore, only when the value of *shared_state* in all processes is one, the reconfiguration will be considered complete.

Any of the aforementioned methods can be modified to incorporate this Single strategy, such that the different stages of spawning processes (expanding or shrinking) are created by auxiliary threads by adding some special operations. However, the Asynchronous Merge method requires controlling the access to all communicators that are created during an expansion, between the main and auxiliary threads, as well as those associated with old and new processes, until old and new processes communicate through a single communicator.

The combination of Single and Asynchronous strategies also generates special situations; therefore, a new stage is defined, in which only the root of parents creates an auxiliary thread to spawn the new processes. Subsequently, when the root of parents verifies that this operation has finished, it will notify the rest of the parents by using a `MPI_Bcast` operation. Then, these create their auxiliary thread which is responsible for completing the connect operation, whereas the main threads continue with the computation. The main threads will periodically test at the checkpoint if the auxiliaries have finished their work. When it occurs, auxiliary threads of parents will terminate and the children continue running.

Figures 1f, 1g, and 1h show how auxiliary threads are responsible for reconfiguration tasks, whereas the main thread continues running. These figures show the way to incorporate thread management in the Baseline method, the Baseline Single method, and the Merge Single method.

Listing 5 shows the pseudo code executed by the main threads to check whether a reconfiguration operation has been completed. Initially, the main threads test the value of *shared_state*, which is shared by the main and auxiliary threads in each process. When the reconfiguration starts, this variable is initialized to *GENERIC_STAGE*, except in the case of using the Single strategy, in which *SINGLE_STAGE* is used. Other values are *SINGLE_COMPLETED*, which indicates that the Single strategy at the root is complete, or *SPAWN_COMPLETED*, which indicates that the participation of the process in the reconfiguration is complete.

At the beginning of Listing 5, a local copy of the shared variable is created using mutex routines (L8-L10). If the first stage of the Single strategy is enabled, the local copy of the root parent process is sent to the remaining $NP - 1$ parents using `MPI_Bcast` (L17). When all processes obtain *SINGLE_COMPLETED*, the second stage starts by writing *GENERIC_STAGE* in the shared variable, and the non-root processes create their auxiliary thread (L18-26). This value will be changed to *SPAWN_COMPLETED* in each process when the spawn operation is completed. To verify that all parents have finished, `MPI_Allreduce` is used to compute the minimum of local copies. When *global_state* is equal to *SPAWN_COMPLETED* (L32), spawning is terminated and the new communicator created by auxiliary threads is copied to *newcomm*, whereas the routine returns one (L28-36).

```
1 int shared_state; // Initialized to NO_SPAWN
2 int check_spawn(int rank, int root, MPI_Comm comm,
3     MPI_Comm *newcomm) {
4   int global_state, local_state;
5   int result = 0;
6   pthread_mutex_t m;
7   // The shared value is copied to a local variable
8   pthread_mutex_lock(&m);
9   local_state = shared_state;
10  pthread_mutex_unlock(&m);
11  // Verify if the first step of single
12  //   strategy is running
13  if(local_state == SINGLE_STAGE ||
14        local_state == SINGLE_COMPLETED) {
15    // Test if the first stage of Single
16    //   strategy has ended
17    MPI_Bcast(&local_state, 1, MPI_INT, root, comm);
18    if(local_state == SINGLE_COMPLETED) {
19      pthread_mutex_lock(&m);
20      shared_state = GENERIC_STAGE;
21      pthread_mutex_unlock(&m);
22      if (rank != root) {
23        // Non root parents create their
24        //   auxiliary thread
25        pthread_create(...);
26    }
27  } else {
28    // Verify if the auxiliary threads
29    //   have finished
30    MPI_Allreduce(&local_state, &global_state, 1,
31        MPI_INT, MPI_MIN, comm);
32    if(global_state == SPAWN_COMPLETED) {
33      // Obtains the communicator created by threads
34      get_created_comm(newcomm);
35      result = 1;
36    }
37  }
38  return result;
39 }
```

Listing 5: Basic skeleton to verify if the auxiliary thread ended their work.

Figure 2 shows a flowchart of the Asynchronous and Single strategies' interactions in the Baseline method, when a reconfiguration from two parents to one child is performed. A line divides the Single and Generic stages in the flowchart. Label *P* and *C* denote the parents and children, respectively. In addition, *Main* and *Aux* denote the master threads and the auxiliary threads, respectively.

To obtain the theoretical computational time of this strategy (RT_A), we consider that an iterative application is executed, where T_{iter} is the time required to perform an iteration with NP processes, and RT_S is the theoretical reconfiguration time from NP to NC processes, as defined in previous subsections. Thus, *It*, the theoretical number of iterations during an asynchronous reconfiguration, can be calculated as it is shown in (6),

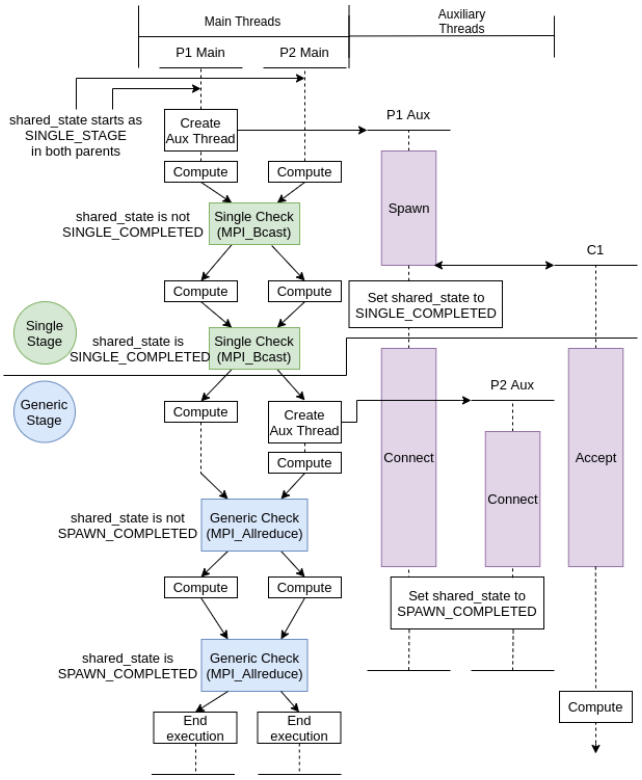


Figure 2. Flowchart example of the asynchronous and single strategies' interactions in the Baseline method. The reconfiguration is performed from two parents (P) to one child (C).

$$\begin{aligned} RT_A &= RT_S(NP, NC) * \alpha, \\ It &= ceil(RT_A/(T_{iter}(NP) * \omega)), \\ CT &= It * T_{iter}(NP) * \omega, \end{aligned} \tag{6}$$

where α and ω are factors $>= 1$, which can increase the cost of the corresponding operations as they have been overlapped. Finally, the theoretical computational time (CT) is obtained as the time consumed in the execution of *It* iterations.

3.5 Integrating the methods in an iterative application

Adopting malleability in C/C++ applications requires modifying their routine `main`. The red lines in Listing 6 show the typical adaptation of an iterative application:

- `MPI_Comm_get_parent` (L4) is used to determine if the current process is a parent or a child.
- If the value of `intercomm` (L5) is `MPI_COMM_NULL`, the process is a parent and some specific data initialization should be performed (L6).
- Otherwise (L8-L11), the process has been spawned using `MPI_Comm_spawn`, and a different initialization is needed to complete the reconfiguration stage. Furthermore, the data redistribution should be performed at the beginning of the code to allow the correct execution of the iterative application.

Special considerations must be applied in L8 of Listing 6, whose implementation could be different in each proposed method. Particularly, when the Merge method is applied for expanding, the execution is based on the code included in Listing 2 (L22-L27). Similarly, the Single strategy is based on the code in L32-45 of Listing 4.

```
1 int main(int argc, char* argv[]) {
2   MPI_Init_thread(...);
3   MPI_Comm intercomm;
4   MPI_Comm_get_parent(&intercomm); // parent or child?
5   if(intercomm == MPI_COMM_NULL){ // Is it a parent?
6     // Application initialization, if it is needed
7   } else { // It is a child
8     // Additional spawn operations, if they are needed
9     // Data redistribution is performed,
10    //   if it is necessary
11    MPI_Comm_free(&intercomm);
12  }
13  MPI_Comm comm = MPI_COMM_WORLD;
14  int break_needed;
15  pthread_t thread_aux;
16  // Code for iterative application
17  while (...) {
18    // Check if a reconfiguration is running or applied
19    checkpoint_for_reconf(&comm, &thread_aux,
20        &break_needed);
21    if (break_needed) break;
22    // Code of each iteration
23  }
24  // Operations to exit the application,
25  //   if they are needed
26  MPI_Finalize( );
27 }
```

Listing 6: Skeleton of an iterative application with malleability.

In addition, the blue lines (L13-L25) in Listing 6 determine the way to apply malleability, although the behaviour of each method is different. The key is the routine *checkpoint_for_reconf*, whose code is summarized in Listing 7. First, it will check whether a previously started asynchronous reconfiguration has been completed invoking `check_spawn` in Listing 5, otherwise, the application asks the RMS if a reconfiguration is required, in which case, a specific method is started.

Some considerations about the adaptations of the output variables are as follows:

- The parameter *comm* could be changed if the reconfiguration is completed during the execution of the routine.
- Asynchronous methods create auxiliary threads, which are stored in the variable *thread_aux*. For the other methods, this variable is not used.
- The parameter *break_needed* is always active for the parents of Baseline methods when the spawning is completed because their execution should be terminated. Although this parameter is inactive for Merge Expand methods because the processes continue the execution.
- For the Merge Shrink methods, processes in which *zombie* state is enabled also require *break_needed* to be set. Therefore, when they wake up they will exit the loop.
- Processes which continue the execution after Merge Shrink methods must wake up the *zombie* processes in L24-L25 of Listing 6.

```
// Indicates if an asynchronous reconfiguration
// has started
int async_reconf_started = 0;
int shared_state = NO_SPAWN;
void checkpoint_for_reconf(MPI_Comm *comm,
    pthread_t *thread_aux, int *break_needed){
  if (async_reconf_started) {
    // Check Asynchronous state
    if (check_spawn( ..., comm) ==
        SPAWN_COMPLETED ) {
      // Adapt parameters: thread_aux and break_needed
      //   async_reconf_started = 0
    } else {
      // Asynchronous reconfiguration still running
      return;
    }
  } else {
    // Ask RMS if a reconfiguration is required

    // Choose method + strategy: mall_meth
    async_reconf_started =
        ((mall_meth is async_meth)? 1: 0);
    if (async_reconf_started)
        // If the asynchronous method,
        //   initialize shared_state
        shared_state = ((mall_meth is Single)?
            SINGLE_STAGE : GENERIC_STAGE);
    // Adapt parameters: comm, thread_aux, break_needed
  }
}
```

Listing 7: Basic code for a reconfiguration point.

4 Experimental results

This section presents the experiments performed to compare the methods described in the previous section for expanding and shrinking the number of processes of a job.

The results were obtained using a synthetic iterative application, in which the computation time and communications executed in each iteration can be parameterized.

4.1 Synthetic application

The authors in Martín-Álvarez et al. (2021) and Martín-Álvarez et al. (2022) described a synthetic application, which allows configuring benchmarks to study the effect of malleability in applications.

This tool simulates and monitors the computational behaviour of scientific MPI iterative applications.

Additionally, it also provides the possibility of being reconfigured during its execution, simulating the RMS demands, in which the number of processes of a job is expanded or shrunk.

This action requires stopping the active application in the job, creating/terminating processes, redistributing the data, and continuing the application with the new process layout.

The executions of the tool are parameterized through a configuration file, in which the main features of the computational behaviour of a simulated application, as well as the description of the reconfiguration stages, are included. Thus, the use of this tool allows to analyse of the impact of malleability on the computational behaviour of MPI applications, comparing the performances of different implementation techniques in distinct scenarios. The conclusions of this analysis will be used to incorporate the best malleability alternatives in the actual implementation of an application. Notably, more than one reconfiguration stage can be included in the configuration file, describing a sequence of events managed by the RMS. Therefore, a hierarchy of processes is obtained, in which each level is composed of its active processes, and a transition between levels is related to a reconfiguration.

This tool includes five main modules, which are briefly described as follows.

Initialization module is in charge of starting the execution of the simulations. 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 simulated application in the *Application simulation* module.

Application simulation module simulates the execution of an iterative application with a specific computational behaviour at one level of the hierarchy of processes. The main features of the simulation 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 simulation, the time spent for completing an iteration, the memory consumed in the simulated application, and the number of bytes transferred in both point-to-point and collective communication 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. Thus, a simulation step will execute the operation the computed number of times or perform the specified communication operation transferring the corresponding number of bytes. Furthermore, this module is responsible for ensuring that a simulation 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 simulation. Two main tasks are involved: creating/terminating processes (*Reconfiguration stage*) and redistributing data from old processes to new processes (*Redistribution stage*). For the first task, all the methods and strategies described in Section 3 are available.

Monitoring module keeps track of the different parts of the simulation timings. These values are stored in intermediate output files after each level of the hierarchy finalizes their execution. *MPI_Wtime* is used to perform these evaluations in each iteration, both in process reconfiguration and in data redistribution.

Completion module has two main tasks. On the one hand, it finalizes processes at the end of each level of the hierarchy of processes. Depending on the method/strategy chosen in *reconfiguration* module, all active processes, some of them or none are finalized. On the other hand, it is also responsible for writing the timings monitored into the intermediate output files for further analysis.

Figure 3 shows a workflow diagram of the synthetic application. In this figure, its different functionalities are highlighted by colours, using a colour per each different module: green for *Initialization*, yellow for *Application simulation*, purple for *Malleability*, blue for *Monitoring*, and red for *Completion*. The simulation begins with the execution of the *Initialization* module by the first group of processes in the hierarchy. However, only a single process manages to read all the parameters of the configuration file and store

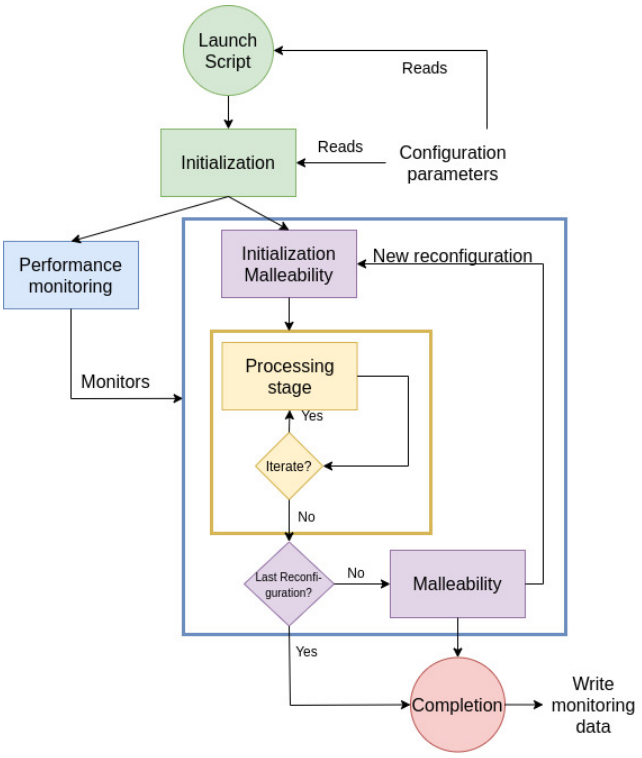


Figure 3. Flowchart of the synthetic application. The colour of each task corresponds to the related module, green for *Initialization*, yellow for *Application simulation*, purple for *Reconfiguration*, blue for *Monitoring*, and red for *Completion*.

them in an internal data structure that is copied to the rest of the active processes. This structure will be also copied through the different levels of the hierarchy of processes when each reconfiguration stage is completed. The *Application simulation* module is then started, computing a simulation step as many times as defined in the configuration file, which also includes the main features of each simulation step. In the case that a new reconfiguration is set (new level in the hierarchy of processes), the execution continues in the *Malleability* module, spawning/shrinking processes and redistributing data. Then, the *Monitoring* module stores performance information on an intermediate file, and the *Application simulation* module continues the simulation. If no more levels exist, the simulation finalizes, and intermediate files generated by *Monitoring* module are merged by *Completion* module, obtaining the performance of the simulation, which is written to an output file.

4.2 Testbed

The experiments were executed on a cluster of six servers with two 10-core Intel Xeon 4210 processors for a total of 120 cores, using MPICH 3.4.3 to compile and link the sources.

The study only considers a single reconfiguration stage per experiment, doing it from 1, 10, 20, 40, 80, and 120 processes to any of the same numbers, and testing the different strategies to expand/shrink processes described in Section 3.

In all cases, the number of occupied nodes is computed as $\lceil N/20 \rceil$, where N will be the number of parents (NP) or children (NC), minimizing the resources allocated by the RMS.

For the experiments, the configuration file contained sequential iteration time (T_{iter}^{seq}), benchmark type, number of iterations before the reconfiguration, and the number of processes after the reconfiguration. It also includes a parameter to choose the method used to modify the number of processes during the reconfiguration, in which all methods explained in Section 3 are considered.

To properly interpret the results, the experiments only consider computation operations, avoiding the impact of communications on the analysis. Therefore, no communication exists among processes and no data redistribution occurs during the reconfiguration. In other words, the processes compute Montecarlo π during $T_{iter}^{bench}(X)$ s, where *bench* defines the executed benchmark and X is the number of processes. Moreover, the chosen operation is a CPU-bound computation, the use of which simplifies the comparison of execution times.

Two types of scenarios were considered for the evaluation of reconfiguring applications. One that simulates a **strong scaling** benchmark, where the system workload is perfectly divided among the processes involved in the computation. Thus, the execution time in each process ($T_{iter}^{strong}(X)$) will be computed dividing T_{iter}^{seq} by the number of processes X, before and after reconfiguring, NP and NC, respectively.

The strong scaling benchmarks were configured to perform 100 iterations, three iterations before the reconfiguration and 97 afterwards. Moreover, T_{iter}^{seq} was set to 4 s.

The other type simulates a **weak scaling** benchmark, where the system workload is modified when the number of processes increases or decreases because the processing workload is fixed in each process ($T_{iter}^{weak}(X) = T_{iter}^{seq}$).

The weak scaling benchmarks perform 30 iterations, three iterations before reconfiguring, and 27 afterwards when T_{iter}^{seq} is equal to 0.2 s.

The number of iterations is smaller for these benchmarks to reduce the effect of performing a large number of iterations after the reconfiguration, allowing better analysis of the impact of the methods defined Section 3 in the final execution time.

Other intermediate experiments were also tested, analyzing the strong scaling of the simulation when its speed-up is distant from the maximum value. However, the results are not shown because they were similar to the strong scaling benchmark. We prefer to include extreme cases to better explain the differences between the benchmarks.

From the previous definitions, the theoretical CT of each test (T_ex) is computed by using (7),

$$T\_ex = 3 * T_{iter}(NP) + RT(NP, NC, M) + (It_{AR} - It) * T_{iter}(NC), \quad (7)$$

where

- $T_{iter}(X)$ is the time required to complete an iteration in X processes. For weak scaling simulation, T_{iter} will be equal to T_{iter}^{seq}, whereas this value will be divided by *NP* or *NC* for strong scaling simulations.
- *RT* is the time to complete a reconfiguration, whose value will be related to *NP* and *NC*, and also to the selected method (*M*). If the chosen method is asynchronous, *RT* is equal to *CT* in (6).

– It_{AR} is the total amount of iterations, which should be performed with *NC* processes without considering an asynchronous strategy. Therefore, this value is 97 for a strong scaling benchmark and 27 for the weak scaling one.
– *It* indicates the number of iterations the initial processes have performed while reconfiguring. Its value will be zero if the method is synchronous and for asynchronous methods a value between one and It_{AR}.

The times shown in the following tables correspond to the median of 10 different executions using the same configuration file, such that variability is reduced.

Moreover, for each table, the lowest time in each row is highlighted in bold, and values within a range of 5% have their cells coloured.

4.3 Experimental reconfiguration evaluation of isolated methods

The first analysis compares the execution time of the different methods, measured from the instant in which parents start the expansion/shrink operation until children continue the execution of the benchmark. The results will be shown by groups to facilitate the analysis, considering expansion and shrinkage separately, as well as synchronous and asynchronous methods.

On the one hand, for the synchronous methods, the reconfiguration time is independent of the benchmark because it pauses its execution while the reconfiguration is being performed. On the other hand, for asynchronous methods, this difference could be important because the benchmark continues running iterations while the reconfiguration is being performed, and the overlapping of both tasks can influence the performance of the benchmark.

4.3.1 Synchronous methods

Table 1 shows the reconfiguration times (RT) in seconds for job expansion depending on the number of initial processes (*NP*) and final processes (*NC*). The Merge method is the clear winner, followed by the Merge Single, with up to a 29% higher time than the winner. This is reasonable since the winning methods spawn fewer processes than the others, and this difference is more relevant as *NP* grows. It is relevant only when $NC - NP$ is equal to $NC - 1$, as the overhead of *T_Mrg(NC)* in (2) is significant, and the Baseline method overcomes the others, e.g., expanding from 1 to 120. Additionally, the Single strategy is not beneficial for Baseline and Merge methods, because the overhead of *T_Con(NP,NC)* is greater than the benefit of using *T_Spw(1,NC)* in (4), instead of just using *T_Spw(NP,NC)* in (1).

Table 2 shows reconfiguration times (*RT*) (in seconds) for process shrinkage using synchronous methods, depending on the number of initial processes and final processes (*NP, NC*), respectively. Again, the Merge method is always the best one, since *T_Split(NP)* in (3) is more than 10 times cheaper than *T_Spw(NP,NC)* in (1). Additionally, the results demonstrate that incorporating the Single strategy into Baseline method makes the shrinkage up to 61% slower.

The conclusion is that the Merge method is nearly always the best option for both expansion and shrinkage when choosing a synchronous method.

Table 1. Expansion median reconfiguration times (RT_S) in seconds for synchronous methods. NP and NC denote the initial and final number of processes, respectively.

NP	NC	RT_S(s)			
		Baseline	Baseline single	Merge	Merge single
1	10	0,316	0,313	**0,284**	0,289
	20	0,861	1,035	**0,716**	0,721
	40	0,861	0,995	**0,799**	0,809
	80	0,989	1,075	**0,932**	1,211
	120	**0,912**	1,029	0,992	1,023
10	20	1,286	1,654	**0,477**	0,486
	40	1,213	1,635	0,766	**0,743**
	80	1,293	1,693	**0,861**	0,888
	120	1,315	1,636	**0,891**	0,915
20	40	1,304	1,991	**0,790**	0,821
	80	1,407	1,932	**0,864**	0,958
	120	1,413	1,870	1,089	**1,071**
40	80	1,429	2,022	0,894	**0,877**
	120	1,526	2,113	**0,923**	0,942
80	120	1,522	2,316	**0,906**	0,996

Table 2. Shrinkage median reconfiguration times (RT_S) in seconds for synchronous methods. NP and NC denote the initial and final number of processes, respectively.

NP	NC	RT_S(s)		
		Baseline	Baseline single	Merge
10	1	0,200	0,205	**0,001**
20	1	0,400	0,427	**0,001**
	10	0,933	1,220	**0,001**
40	1	0,400	0,423	**0,030**
	10	0,883	1,165	**0,025**
	20	1,271	1,740	**0,116**
80	1	0,388	0,418	**0,217**
	10	0,881	1,189	**0,181**
	20	1,262	1,826	**0,149**
	40	1,415	2,039	**0,148**
120	1	0,375	0,424	**0,231**
	10	0,953	1,251	**0,148**
	20	1,229	1,776	**0,178**
	40	1,300	2,102	**0,351**
	80	1,529	2,235	**0,156**

4.3.2 Asynchronous methods

Asynchronous strategies consider that an iterative application is being executed during the reconfiguration. Therefore, tables also include the application point of view, showing the number of times `checkpoint_for_reconf` (L19-L20 of Listing 6) is executed to complete the reconfiguration (*It*), and the corresponding consumed time, in seconds, from the first to the last call to the computational routine (*CT*).

Tables 3 and 4 show the results for the **strong scaling** case, in which, the iteration time is related to *NP*, as follows:

$$T_{iter}^{strong}(NP) = T_{iter}^{seq}/NP. \quad (8)$$

Attempting to avoid extremely small iteration values, T_{iter}^{seq} has been fixed to 4 sec, such that for 120 processes, $T_{iter}^{strong}(120)$ is equal to 0.033 sec.

The first analysis is that *CT* is significantly higher than *RT* ranging from 12% to 1030% higher. It is worth noting that *CT* depends on both *RT* and $T_{iter}^{strong}(NP)$ because the checkpoint is performed at the end of each iteration. If we compare the *RT* in Tables 1-3, and 2-4, we can observe that it is also increased in all asynchronous cases.

Next analysis compares the values in Tables 1-3, and 2-4, computing the values of α in (6). Figure 4 shows the obtained values for expanding (top) and shrinking (bottom). Thus, the expansion produces an α increase lower than 70% in most cases, and usually higher values are related to methods with Single strategy, whereas Baseline and Merge methods are nearly insignificant ranging from 3% to 30%, except in the case (10,20) for Merge method with a 66%. For shrinking, α values are usually higher than in the expansion, particularly for the Baseline Single method, whose values are worse than Baseline. The magnitude of the first five values in Table 2 is extremely small compared to Table 4, generating α values that are not relevant.

To complete this analysis, the ω values in (6) for the different methods are also studied. These values are computed as the quotient CT/It, and should be compared to $T_{iter}^{strong}(NP)$. Figure 5 shows the obtained values for expanding (top image) and shrinking (bottom image). The analysis for expanding allows to conclude that *NP* has a significant impact on the ω values, increasing $T_{iter}^{strong}(NP)$ up to three times. By contrast, the relevance of NC is lower. Moreover, Merge methods obtain smaller ω values than the Baseline methods, and the use of the Single strategy within the method, in general, allows for to reduction of the ω values up to 90%. In the case of shrinkage, both NP and NC have an impact on ω values. For this reason, Merge is the best method and Single strategy usually reduces ω values.

The increase of α and ω values is justified by the appearance of oversubscription problems when asynchronous methods are used, growing the number of threads that are being executed in the nodes. Only for the cases $(1, 10)$ and $(10, 1)$, the problem is mainly avoided in all cases; however, in the rest of the combinations of NP and NC, the Baseline method surpasses until 40 active threads per node, whereas the Merge method outperforms the others. In general, Merge methods yield lower α and ω values because of their lower number of threads compared to other solutions. However, the inclusion of additional synchronization steps, as in the case of the Baseline Single method, may exacerbate the oversubscription problem. In addition, the use of the Single strategy reduces the number of threads in each node and, therefore, ω values are reduced. The higher ω values are more related to the type of operation executed in the iteration. Montecarlo π is a CPU-bound computation, and, therefore, the oversubscription has a high impact on it. The use of memory-bound could reduce ω values.

In summary, we can conclude that the Merge method is the best alternative for both expansion and shrinkage reconfiguration for strong scaling applications.

Results for **weak scaling** benchmarks are included in Tables 5 and 6, where $T_{seq\_iter} = 0.2$ sec. We note that in these experiments, the workload of the benchmark grows to expand and reduces to shrink.

It is worth noting that in these tables, unlike in the strong scaling benchmarks, *RT* and *CT* vales are similar in all combinations of NP and NC. Additionally, the *It* columns include smaller values, all of them lower than 10, whereas Table 4 includes some *It* values close to 40.

The analysis of α and ω values in Figures 6 and 7 allows drawing conclusions similar to the previous ones. Methods based on Single strategy grow α values by 40% on average

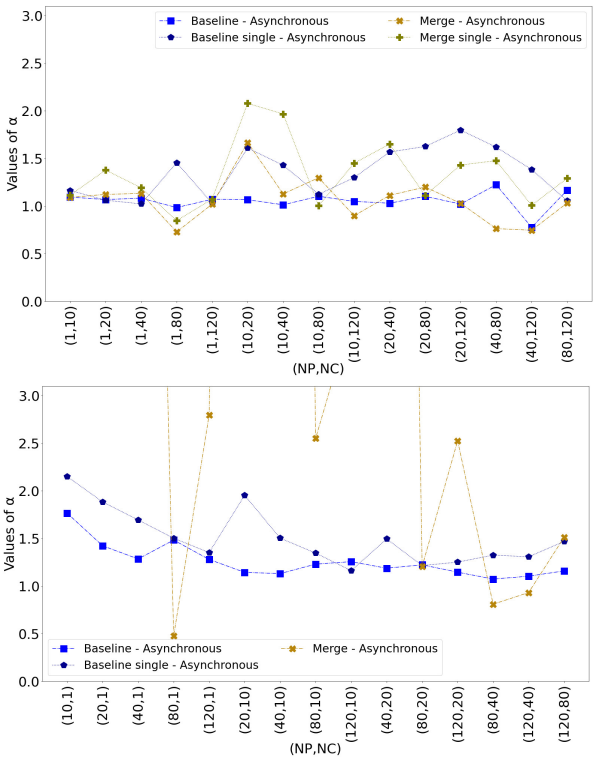


Figure 4. Values of α for a strong scaling benchmark for the asynchronous methods. Expansion at the top and Shrinkage at the bottom.

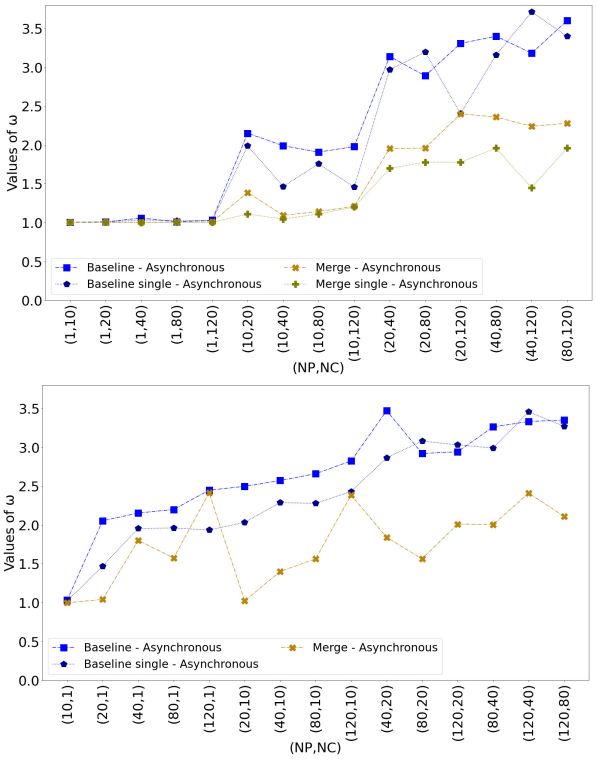


Figure 5. Values of ω for a strong scaling benchmark for the asynchronous methods. Expansion at the top and Shrinkage at the bottom.

while reduce ω values by 50%, compared to non-Single methods. Higher ω values are justified by the CPU-bound

Table 3. Expansion median reconfiguration times (RT_A), compute time of the benchmark (CT), both in seconds, and performs iterations (It) for all asynchronous methods. NP and NC denote the initial and final number of processes, respectively.

NP	NC	Baseline			Baseline single			Merge			Merge single		
		RT_A(s)	CT(s)	It	RT_A	CT(s)	It	RT_A(s)	CT(s)	It	RT_A(s)	CT(s)	It
1	**10**	0,355	4,026	1	0,345	4,025	1	**0,307**	**4,015**	1	0,320	4,025	1
	20	0,900	4,086	1	1,099	4,069	1	**0,853**	4,019	1	0,991	**4,016**	1
	40	0,942	4,264	1	1,009	4,228	1	**0,877**	4,024	1	0,980	**4,014**	1
	80	1,001	4,054	1	1,596	4,275	1	**0,891**	4,060	1	1,113	4,070	1
	120	0,966	**4,246**	1	1,037	4,132	1	**0,898**	4,054	1	1,124	4,086	1
10	**20**	1,317	2,312	2	2,676	4,240	6	**0,799**	**1,114**	2	1,030	1,391	3
	40	1,308	2,122	3	2,292	3,708	6	**0,915**	**1,462**	3	1,367	1,742	4
	80	1,463	2,159	3	2,322	3,678	5	**1,085**	**1,496**	3	1,456	1,799	4
	120	1,356	2,123	3	2,555	3,227	5	**1,130**	**1,570**	3	1,444	1,804	4
20	**40**	1,386	1,946	3	3,049	3,769	7	**0,895**	**1,065**	3	1,259	1,588	5
	80	1,486	1,874	3	3,131	3,857	7	**1,018**	**1,136**	3	1,392	1,839	6
	120	1,429	1,971	3	3,197	3,929	8	**1,069**	**1,536**	3	1,640	2,126	7
40	**80**	1,621	1,861	5	3,233	3,651	12	**0,926**	**1,170**	5	1,247	1,457	8
	120	1,499	1,940	5	2,850	3,169	11	**0,980**	**1,199**	5	1,196	1,416	9
80	**120**	1,727	1,943	7	2,943	3,205	23	**0,934**	**1,059**	7	1,231	1,304	16

Table 4. Shrinkage median reconfiguration times (RT_A), compute time of the benchmark (CT), both in seconds, and iterations (It) for all asynchronous methods. NP and NC denote the initial and final number of processes, respectively.

NP	NC	Baseline			Baseline single			Merge		
		RT_A(s)	CT(s)	It	RT_A(s)	CT(s)	It	RT_A(s)	CT(s)	It
10	**1**	0,347	0,417	1	0,456	0,818	2	**0,021**	**0,401**	1
20	**1**	0,568	0,837	2	0,813	1,229	4	**0,024**	**0,211**	1
	10	1,077	1,567	3	1,980	2,652	5	**0,021**	**0,209**	1
40	**1**	0,502	0,690	3	0,700	0,925	5	**0,146**	**0,233**	1
	10	1,101	1,361	5	1,735	1,985	8	**0,150**	**0,297**	2
	20	1,440	1,758	5	2,452	2,780	10	**0,164**	**0,245**	1
80	**1**	0,544	0,640	5	0,625	0,730	8	**0,103**	**0,168**	2
	10	1,090	1,266	8	1,673	1,881	17	**0,138**	**0,244**	2
	20	1,426	1,620	8	2,280	2,442	17	**0,235**	**0,297**	2
	40	1,418	1,716	8	2,618	2,872	21	**0,180**	**0,286**	2
120	**1**	0,516	0,592	6	0,569	0,654	9	**0,337**	**0,402**	4
	10	1,170	1,269	11	1,466	1,589	19	**0,352**	**0,477**	3
	20	1,368	1,518	11	2,217	2,401	26	**0,315**	**0,386**	3
	40	1,479	1,721	11	2,541	2,707	27	**0,358**	**0,452**	4
	80	1,696	1,868	11	3,195	3,473	38	**0,324**	**0,407**	4

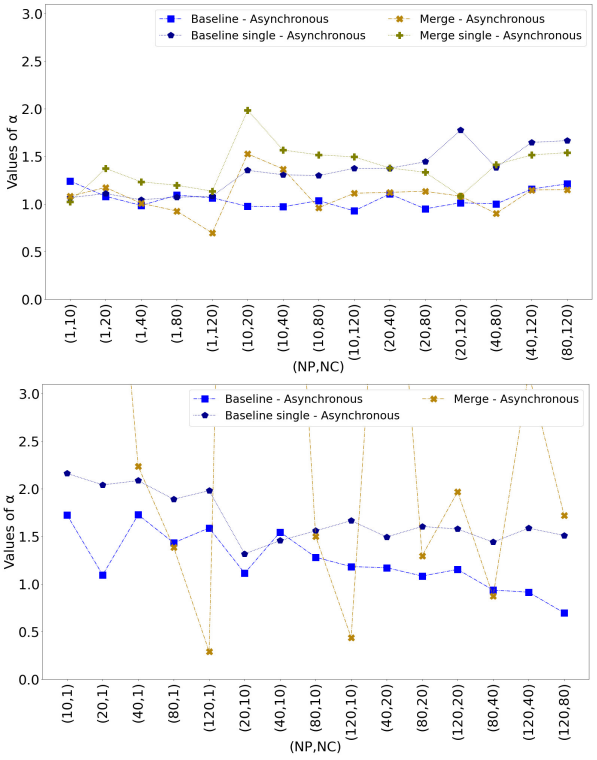


Figure 6. Values of α for a weak scaling benchmark for the asynchronous methods. Expansion at the top and Shrinkage at the bottom.

benchmark executing during the iteration, which is more sensitive to the growth of the number of threads. Moreover, *NP* is directly related to ω values because *NP* determines the number of threads in the nodes, and then the importance of oversubscription problems. Merge methods have a limited impact on α and obtain the best ω values, therefore, these are the best alternative.

4.4 Experimental reconfiguration evaluation for applications and system productivity

Malleability is useful in large computing facilities when the RMS decides to reconfigure jobs. Furthermore, from the standpoint of system productivity, the reconfigure stage should be completed as soon as possible to avoid resources remaining idle while some jobs need more resources (expansion), or to reduce the waiting time for jobs which are ready to be executed using the released resources (shrinking). Alternatively, applications would be more interested in reducing their execution time.

Therefore, this section analyses the performance of the different methods of expanding or shrinking when they are integrated into an application to examine if the overlapping of the computation and the reconfiguration stage using threads is useful. Moreover, the aforementioned two viewpoints have been considered, obtaining the best method for the studied benchmarks and the best method for improving the performances of the entire system.

For the first case, the best method for strong and weak scaling benchmarks are analysed, whereas, for the second case, those methods whose *RT* is closer than 10% of the faster method in each combination of *NP* and *NC* are also considered in the analysis of the system productivity.

Table 5. Expansion median reconfigure times (RT_A), compute time (CT) in seconds, and perform iterations (It) for all asynchronous methods in a task-based benchmark. Labels NP and NC denote the initial and final number of processes, respectively.

NP	NC	Baseline			Baseline single			Merge			Merge single		
		RT_A(s)	CT(s)	It	RT_A(s)	CT(s)	It	RT_A(s)	CT(s)	It	RT_A(s)	CT(s)	It
1	10	0,375	**0,418**	2	0,359	0,419	2	**0,291**	0,420	2	0,313	0,420	2
	20	0,914	1,020	5	1,050	1,217	6	**0,836**	**1,009**	5	0,980	1,013	5
	40	0,871	1,026	5	1,118	1,215	6	**0,837**	**1,018**	5	0,918	1,020	5
	80	0,955	**1,023**	5	1,086	1,210	6	**0,880**	1,026	5	1,047	1,219	6
	120	0,935	**1,028**	5	1,108	1,246	6	**0,917**	1,140	5	1,030	1,271	6
10	20	1,203	1,530	4	2,199	2,813	9	**0,746**	**1,000**	4	0,932	1,088	5
	40	1,219	1,811	5	2,026	2,682	9	**0,997**	**1,255**	5	1,152	1,291	6
	80	1,338	1,789	5	2,273	2,924	9	**0,950**	**1,268**	5	1,343	1,755	7
	120	1,262	1,778	5	2,272	2,984	9	**0,981**	**1,332**	4	1,413	1,648	7
20	40	1,393	1,911	3	3,184	3,932	8	**0,872**	**1,104**	3	1,314	1,618	5
	80	1,567	2,446	4	3,025	3,809	7	**0,998**	**1,235**	3	1,365	1,724	6
	120	1,393	1,841	3	3,107	4,043	7	**0,985**	**1,191**	3	1,549	1,842	7
40	80	1,496	2,002	3	3,407	4,341	9	**0,942**	**1,201**	3	1,317	1,698	5
	120	1,609	2,473	4	3,468	4,220	9	**1,052**	**1,231**	3	1,499	1,807	6
80	120	1,941	2,693	4	3,780	4,514	8	**0,989**	**1,291**	3	1,394	1,763	6

Table 6. Shrinkage median reconfiguration times (RT_A), compute time (CT) in seconds, and perform iterations (It) for all asynchronous methods in a task-based benchmark. Labels NP and NC denote the initial and final number of processes, respectively.

NP	NC	Baseline			Baseline single			Merge		
		RT_A(s)	CT(s)	It	RT_A(s)	CT(s)	It	RT_A(s)	CT(s)	It
10	1	0,349	0,414	2	0,433	0,619	3	**0,020**	**0,201**	1
20	1	0,438	0,778	2	0,798	1,233	3	**0,015**	**0,203**	1
	10	1,086	1,546	3	1,751	2,405	5	**0,016**	**0,204**	1
40	1	0,665	0,865	2	0,869	1,351	4	**0,142**	**0,284**	1
	10	1,227	1,577	3	1,943	2,511	5	**0,174**	**0,296**	1
	20	1,417	2,077	3	2,463	3,446	6	**0,176**	**0,309**	1
80	1	0,575	0,874	2	0,901	1,327	4	**0,317**	**0,606**	2
	10	1,107	1,581	3	1,883	2,481	6	**0,298**	**0,484**	1
	20	1,376	1,998	3	2,884	3,646	7	**0,291**	**0,593**	2
	40	1,669	2,501	4	2,973	3,946	7	**0,257**	**0,491**	1
120	1	0,630	0,884	2	0,976	1,366	4	**0,322**	**0,600**	2
	10	1,153	1,547	3	2,129	2,730	6	**0,330**	**0,606**	2
	20	1,368	1,984	3	2,728	3,462	6	**0,343**	**0,711**	2
	40	1,605	2,119	3	3,068	3,890	7	**0,353**	**0,720**	2
	80	1,700	2,406	3	3,551	4,348	8	**0,281**	**0,672**	2

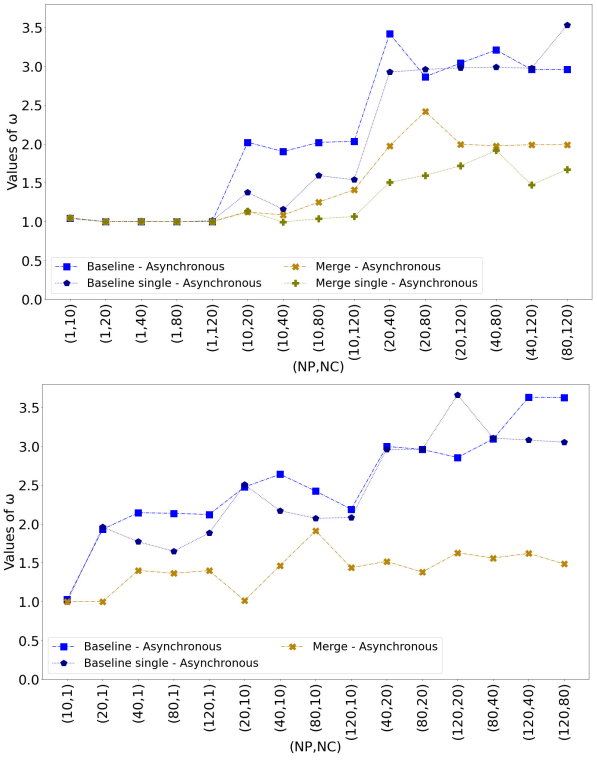


Figure 7. Values of ω for a weak scaling benchmark for asynchronous methods. Expansion at the top and Shrinkage at the bottom.

4.4.1 *Experimental reconfiguration evaluation for strong scaling benchmarks*

Figure 8 shows graphically the best method for each pair (*NP*, *NC*) from both viewpoints, benchmark (top) and system productivity (bottom). The name of the axes, vertical for *NP* and horizontal for *NC*, 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.

For our strong scaling benchmark, increasing the number of processes reduces $T_{iter}^{strong}(X)$, allowing faster finalization of the application. Therefore, Merge methods predominate in the upper triangular part of Figure 8 (top) because these methods were the fastest for reconfiguring. Moreover, nearly all the shown methods are synchronous because asynchronous methods delay the transition in most cases, except pairs $(20, 40)$ and $(80, 120)$, in which the difference to the synchronous counterpart is negligible. Thus, the results in this figure correspond to the values in Table 1.

By contrast, decreasing the number of processes grows $T_{iter}^{strong}(X)$, decelerating the finalization of the benchmark. Thus, Asynchronous Baseline Single methods are more common in the lower triangular part of Figures 8 (top) because they are usually the slowest method (see Tables 2 and 4). Although in some cases ω values are greater than the ratio between $T_{iter}^{strong}(NP)$ and $T_{iter}^{strong}(NC)$, and therefore the aforementioned rule cannot be applied. To confirm this assertion, we compute T_ex for the different cases of the shrinkage from 120 to 80 processes using Equation (7), obtaining that T_ex for Asynchronous Merge is 5.157 s, whereas T_ex for Asynchronous Baseline and Asynchronous

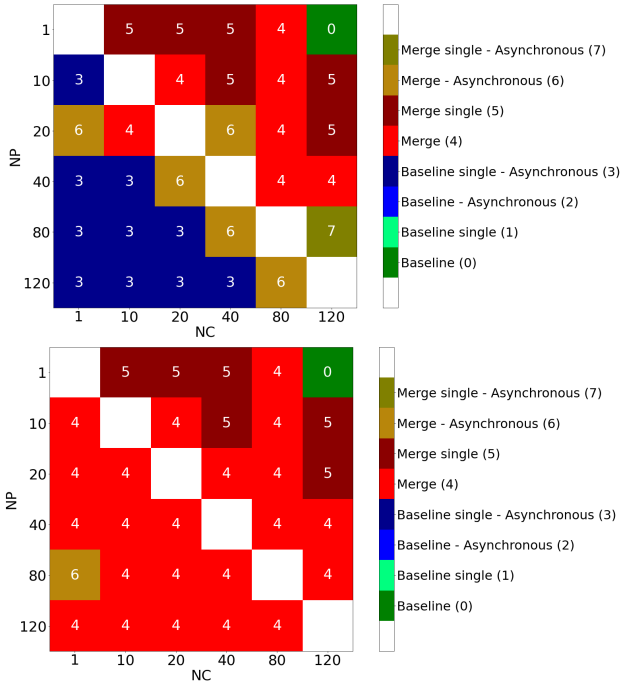


Figure 8. Colour-maps of preferred methods depending on the number of initial processes and final processes in a strong scaling simulation. Application perspective at the top and RMS point of view at the bottom.

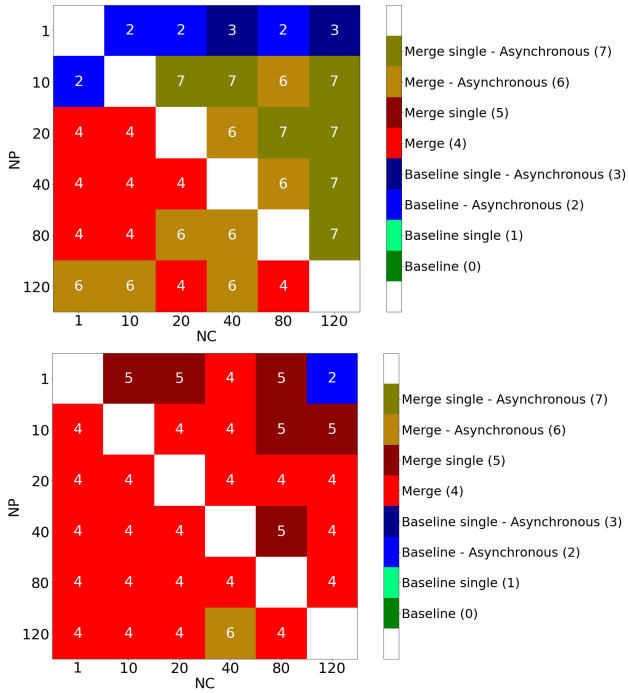


Figure 9. Colour-maps of preferred methods depending on the number of initial processes and final processes in a weak scaling simulation. Application perspective at the top and RMS point of view at bottom.

Baseline Single are 6.268 sec and 6.523 sec, respectively. Therefore, Asynchronous Merge is the best option for this shrinkage.

In terms of the system productivity, the conclusions for expansion in Figure 8 (bottom) are the same as in terms of the application because the best method is the same for each pair (*NP*, *NC*). The only exceptions are the pairs $(20, 40)$ and $(80, 120)$ as Asynchronous Merge and Asynchronous Single Merge are excluded from the analysis because *RT* is greater than 10% threshold of the fastest method (Merge) for these pairs.

By contrast, the conclusion for shrinkage is completely different because the Synchronous Merge method is generally chosen because it is the fastest method under the threshold.

This shows that the overhead of overlapping computation is not beneficial for the system's productivity.

4.4.2 Experimental reconfiguration evaluation for weak scaling benchmarks

Figure 9 shows the same analysis as described in the previous subsection for a weak scaling simulation. We note that in this case, T_{iter}^{weak} is always equal to 0.2 sec regardless of the number of processes, therefore, it is not an aspect to consider in the analysis.

From the point of view of applications, Figure 9 (top) summarize the values in Tables 1-2 and Tables 5-6. For expansion, Asynchronous Merge methods are usually the best option, with or without the Single strategy, depending on the pair. The asynchronous methods are preferred as the iteration time is constant and the Merge method spawns fewer processes than the Baseline counterparts. We note that the Single strategy is usually related to lower ω values, and then, a greater number of iterations is executed during the reconfiguration, reducing the number of iterations after the reconfiguration. Moreover, Asynchronous Baseline methods are the best alternative when the number of parents is equal to one because the cost of spawning processes is the same between the Baseline and Merge methods, although Merge methods must additionally perform the *T_Mrg(NC)* operation.

For shrinking, the best option is to use Merge methods because they are faster than an overlapped iteration, which is affected by an ω. Usually, synchronous methods are more appropriate except for some pairs, in which small α and ω values allow asynchronous methods to be the best alternative.

In terms of system productivity, Figure 9 (bottom), the expansion changes drastically to use the Merge and Merge Single methods in nearly all cases. This occurs because the difference between RT in the asynchronous versions contains at least one overlapped iteration, which is sufficient to exceed the threshold of 10%.

The similarity for shrinkage of both colour-maps in Figure 9 allows us to assert that the conclusions for both viewpoints are extremely similar with a higher tendency to the Merge method. Synchronous methods which outperform their asynchronous counterparts are based on the same reasoning previously discussed for the expansions.

5 Conclusions

There are different methods and strategies to perform the reconfiguration stage in an application when applying malleability.

In this paper, two initial methods are introduced (Baseline and Merge) on which two additional strategies can be applied (Single and Asynchronous), obtaining up to eight different

alternatives to spawn processes dynamically. All of them have been evaluated by using a synthetic application configured to simulate two types of scenarios: one based on strong scaling benchmarks and another based on weak scaling benchmarks. Both were executed on a six-node cluster, analysing their behaviour for expanding and shrinking operations. The final analysis has compared the methods when applied to different aims: application performance and system productivity.

Due to fewer processes being created on expansion and no processes being created when shrinking, we have demonstrated that compared to the Baseline method, the Merge method is up to 2.6 times faster when expanding, and up to 36 times faster when shrinking. Conversely, the Baseline method is the best alternative only from the application point of view when shrinking with strong scaling because it is a slow method, and a greater number of faster iterations are executed by our benchmarks.

The Single strategy has increased reconfiguration times of the methods because of the inclusion of additional MPI routines and synchronization steps.

For the Asynchronous strategy, reconfiguration times increase are justified by the appearance of oversubscription issues caused by the growth in the number of threads running on the nodes.

Nevertheless, this overhead is useful for shrinking the strong scaling benchmark, improving the execution time up to 20%, compared with the Baseline method, because more iterations with more resources are completed. Similarly, the weak scaling benchmark leverages overlapping reconfiguration and computation in the asynchronous mode, for this reason the execution time is reduced up to 16% when expanding.

In general, when expanding, the Synchronous Merge method has been demonstrated to be the best alternative in 12 and 14 cases out of 15, for application performance and system productivity points of view, respectively.

Only for the weak scaling benchmark, the Asynchronous Merge method performs better for increasing the application performance because overlapping tasks can reduce the execution time.

In the case of shrinkage, the Synchronous Merge method is usually the best alternative. The exception is the application performance point of view for strong scalability, in which the slowness of the Asynchronous Baseline method allows the execution of a greater number of faster iterations.

Therefore, we conclude that the Merge method is the best alternative in nearly all cases. The Single strategy is not recommended because it does not yield significant benefits. Additionally, the Asynchronous strategy has been proven interesting in nearly half of the cases.

The methods presented in this study are expected to be integrated into a library, which could be leveraged by malleability-enabled RMS. Thus, the best method in different scenarios of reconfiguration will be applied depending on the cluster policies.

Future work will extend the experiments to analyse the behaviour of the methods when the data is distributed from parents to children, both expanding and shrinking. Additionally, the synthetic application will also simulate other features, such as the memory consumed or the number of bytes transferred in both point-to-point and collective operations, such that the simulation will be more realistic.

Additionally, if the ULFM Bland et al. (2013) proposal is added to the MPI standard, the Merge method will be changed to allow the removal of processes according to the proposal, along with the study analysing the way the current implementation will change.

Declaration of conflicting interests

The authors declare that there is no conflict of interest.

Funding

This work has been funded by the following projects: project PID2020-113656RB-C21 supported by MCIN/AEI/10.13039/501100011033 and project UJI-B2019-36 supported by Universitat Jaume I. Researcher S. Iserte was supported by the postdoctoral fellowship APOSTD/2020/026, and researcher I. Martín-Álvarez was supported by the predoctoral fellowship ACIF/2021/260, both from Valencian Region Government and European Social Funds.

Notes

1. https://pm.bsc.es/ompss
2. https://pm.bsc.es/nanox
3. https://slurm.schedmd.com
4. https://www.mpich.org

References

Aliaga JI, Castillo M, Iserte S, Martín-Álvarez I and Mayo R (2022) A survey on malleability solutions for high-performance distributed computing. *Applied Sciences* 12(10). DOI: 10.3390/app12105231. URL https://www.mdpi.com/2076-3417/12/10/5231.

Badia RM, Conejero J, Diaz C, Ejarque J, Lezzi D, Lordan F, Ramon-Cortes C and Sirvent R (2015) Comp superscalar, an interoperable programming framework. *SoftwareX* 3-4: 32–36. DOI:10.1016/J.SOFTX.2015.10.004.

Bernholdt DE, Boehm S, Bosilca G, Venkata MG, Grant RE, Naughton T, Pritchard HP, Schulz M and Vallee GR (2018) A survey of mpi usage in the us exascale computing project DOI: 10.1002/cpe.4851. URL https://doi.org/10.1002/cpe.4851.

Bland W, Bouteiller A, Herault T, Bosilca G and Dongarra J (2013) Post-failure recovery of mpi communication capability: Design and rationale. *The International Journal of High Performance Computing Applications* 27(3): 244–254. DOI: 10.1177/1094342013488238. URL https://doi.org/10.1177/1094342013488238.

Comprés I, Mo-Hellenbrand A, Gerndt M and Bungartz HJ (2016) Infrastructure and api extensions for elastic execution of mpi applications. In: *Proceedings of the 23rd European MPI Users' Group Meeting*, EuroMPI 2016. New York, NY, USA: Association for Computing Machinery. ISBN 9781450342346, p. 82–97. DOI:10.1145/2966884.2966917. URL https://doi.org/10.1145/2966884.2966917.

El Maghraoui K, Szymanski BK and Varela C (2006) An Architecture for Reconfigurable Iterative MPI Applications in Dynamic Environments. In: *International Conference on Parallel Processing and Applied Mathematics*. pp. 258–27.

Hori A, Jeannot E, Bosilca G, Ogura T, Gerofi B, Yin J and Ishikawa Y (2021) An international survey on mpi users. *Parallel Computing* 108: 102853. DOI:10.1016/J.PARCO.2021.102853.

Houzeaux G, Badia RM, Borrell R, Dosimont D, Ejarque J, Garcia-Gasulla M and López V (2021) Dynamic resource allocation for efficient parallel cfd simulations. Technical report, Barcelona Supercomputing Center.

Iserte S, Mayo R, Quintana-Orti E and Pena A (2020) DMRlib: Easy-coding and Efficient Resource Management for Job Malleability. *IEEE Transactions on Computers* DOI:10.1109/TC.2020.3022933.

Iserte S, Mayo R, Quintana-Ortí ES, Beltran V and Peña AJ (2017) Efficient Scalable Computing through Flexible Applications and Adaptive Workloads. In: *46th International Conference on Parallel Processing Workshops (ICPPW)*. Bristol (UK): IEEE. ISBN 978-1-5386-1044-2, pp. 180–189. DOI:10.1109/ICPPW.2017.36. URL http://ieeexplore.ieee.org/document/8026084/.

Iserte S, Mayo R, Quintana-Ortí ES, Beltran V and Peña AJ (2018) DMR API: Improving cluster productivity by turning applications into malleable. *Parallel Computing* 78: 54–66. DOI:10.1016/J.PARCO.2018.07.006. URL https://www.sciencedirect.com/science/article/pii/S0167819118302229?via%3Dihub.

Iserte S, Peña AJ, Mayo R, Quintana-Ortí ES and Beltran V (2016) Dynamic management of resource allocation for ompss jobs. ISBN 978-84-608-6309-0, pp. 55–58.

Iserte S and Rojek K (2019) An study of the effect of process malleability in the energy efficiency on GPU-based clusters. *The Journal of Supercomputing* : 1–20.

Lemarinier P, Hasanov K, Venugopal S and Katrinis K (2016) Architecting Malleable MPI Applications for Priority-driven Adaptive Scheduling. In: *Proceedings of the 23rd European MPI Users' Group Meeting on - EuroMPI 2016*. New York, New York, USA: ACM Press. ISBN 9781450342346, pp. 74–81. DOI:10.1145/2966884.2966907. URL http://dl.acm.org/citation.cfm?id=2966884.2966907.

Lopez V, Ramirez Miranda G and Garcia-Gasulla M (2021) Talp: A lightweight tool to unveil parallel efficiency of large-scale executions. In: *Proceedings of the 2021 on Performance EngineeRing, Modelling, Analysis, and VisualizatiOn STrategy*, PERMAVOST '21. New York, NY, USA: Association for Computing Machinery. ISBN 9781450383875, p. 3–10. DOI:10.1145/3452412.3462753. URL https://doi.org/10.1145/3452412.3462753.

Martín G, Singh DE, Marinescu MC and Carretero J (2015) Enhancing the Performance of Malleable MPI Applications by Using Performance-aware Dynamic Reconfiguration. *Parallel Computing* 46: 60–77.

Martín-Álvarez I, Aliaga J, Castillo MI, Iserte S and Mayo R (2021) A synthetic tool for analysing adaptive workloads. URL https://www.youtube.com/watch?v=kwE2FiU3FM8#t=6h3m20s. Accessed: 2022-03-15.

Martín-Álvarez I, Aliaga JI, Castillo M and Iserte S (2022) Malleable synthetic tool manual. Technical report, Universitat Jaume I.

Message Passing Interface Forum (2021) *MPI: A Message-Passing Interface Standard Version 4.0*. URL https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.

Moody A, Bronevetsky G, Mohror K and de Supinski BR (2010) Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In: *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC10)*. ISBN 978-1-4244-7557-5.

Posner J and Fohry C (2021) Transparent resource elasticity for task-based cluster environments with work stealing. In: *50th International Conference on Parallel Processing Workshop*, ICPP Workshops '21. New York, NY, USA: Association for Computing Machinery. ISBN 9781450384414. DOI:10.1145/3458744.3473361. URL https://doi.org/10.1145/3458744.3473361.

Radcliffe N, Watson L and Sosonkina M (2011) A comparison of alternatives for communicating with spawned processes. In: *Proceedings of the 49th Annual Southeast Regional Conference*, ACM-SE '11. New York, NY, USA: Association for Computing Machinery. ISBN 9781450306867, p. 132–137. DOI:10.1145/2016039.2016078. URL https://doi.org/10.1145/2016039.2016078.

Vadhiyar SS and Dongarra JJ (2002) Srs - a framework for developing malleable and migratable applications for distributed systems. *Parallel Processing Letters* 2: 291–312.

Wittmann M, Hager G, Zeiser T and Wellein G (2013) Asynchronous MPI for the Masses .

Author Biographies

Iker Martín-Álvarez received the BS degree in Computer Engineering from the Universitat Jaume I, Castello de la Plana, Spain, in 2019 and the MS degree in High Performance Computing from the Universidade da Coruña, Spain, in 2021. He is currently carrying out his pre-doctoral studies at the University Jaume I in the High Performance Computing and Architectures (HPC&A) research group, which started in 2021. His main research focuses on the design of a prototype that allows the implementation and malleable execution of MPI scientific applications in data centres. The aim is to analyse different malleable mechanisms to determine how they affect application execution time and system productivity.

José I. Aliaga is a professor of Computer Science and Artificial Intelligence at the University Jaume I (UJI) in Castellón, Spain, where he leads the HPCA group. From 2000 to 2005, he was the head of the department with a staff of over 100 people. He obtained his diploma and PhD degree in Computer Science from Polytechnic University of Valencia (UPV), in 1990 and 1995, respectively. During his career, he has participated in over 35 research projects funded by both national and private organizations (in Spain or within the EU). In 7 of these projects, he figures as the principal investigator. He was also involved in two transfer technology contracts with international partners, leading one of them. His main research interests include the solution of sparse linear algebra problems on current high performance multi-core processors, hardware accelerators and parallel systems.

Maribel Castillo is a member of the High Performance Computing and Architectures group at University Jaume I (UJI) in Castellón, Spain. Shed received her BSC and PhD in Computer Sciences, both from the Universidad Politècnica de Valencia (Spain), in 1992 and 2001, respectively. From 1992 was hired as Assistant Professor and later associate professor at UJI. Her research interest addresses the optimization of scientific applications in general, and in more recent years in bioinformatics applications on general purpose processors as well as hardware accelerators and their parallelization on clusters

and shared memory mutiprocessors. She has also participated in several European and national projects on programming models and energy efficiency. Prof. Castillo has published more than sixty papers in journals and international conferences.

Sergio Iserte holds the degrees of BS in Computer Engineering (2011), MS in Intelligent Systems (2014), and Ph.D. in Computer Science (2018) from Universitat Jaume I (UJI), Spain. Sergio is a senior researcher at Barcelona Supercomputing Center (BSC) in the Computer Science Department, and course instructor of the HPC subject at Universitat Oberta de Catalunya (UOC). He is currently involved in HPC projects related to parallel distributed computing, resource management, workload modeling, deep learning for industrial applications, and in-network accelerators.

Rafael Mayo received the BS degree from the UPV in 1991. He obtained his PhD in Computer Science in 2001 at the same University. Since October 2002, he has been an associate professor in the Department of Computer Science and Engineering in the UJI. His research interests include the optimization of numerical algorithms for general processors as well as for specific hardware, and their parallelization on both message-passing parallel systems (mainly clusters) and shared-memory multiprocessors. He was is involved in several research efforts on HPC energy-aware systems, cloud computing, and HPC system and development tools.