applied sciences

MDPI

Article

A Survey on Malleability Solutions for High-Performance Distributed Computing

Jose I. Aliaga , Maribel Castillo, Sergio Iserte , Iker Martín-Álvarez * and Rafael Mayo

Department de Ingeniería y Ciencia de Computadores, Universitat Jaume I, 12006 Castelló, Spain; aliaga@uji.es (J.I.A.); castillo@uji.es (M.C.); siserte@uji.es (S.I.); mayo@uji.es (R.M.)
* Correspondence: martini@uji.es

Abstract: Maintaining a high rate of productivity, in terms of completed jobs per unit of time, in High-Performance Computing (HPC) facilities is a cornerstone in the next generation of exascale supercomputers. Process malleability is presented as a straightforward mechanism to address that issue. Nowadays, the vast majority of HPC facilities are intended for distributed-memory applications based on the Message Passing (MP) paradigm. For this reason, many efforts are based on the Message Passing Interface (MPI), the de facto standard programming model. Malleability aims to rescale executions on-the-fly, in other words, reconfigure the number and layout of processes in running applications. Process malleability involves resources reallocation within the HPC system, handling processes of the application, and redistributing data among those processes to resume the execution. This manuscript compiles how different frameworks address process malleability, their main features, their integration in resource management systems, and how they may be used in user codes. This paper is a detailed state-of-the-art devised as an entry point for researchers who are interested in process malleability.

Keywords: exascale; job reconfiguration; MPI; data redistribution; resource management; adaptive workloads

check for updates

Citation: Aliaga, J.I.; Castillo, M.; Iserte, S.; Martín-Álvarez, I.; Mayo, R. A Survey on Malleability Solutions for High-Performance Distributed Computing. *Appl. Sci.* **2022**, *12*, 5231. https://doi.org/10.3390/app12105231

Academic Editors: Pedro Valero-Lara and Antonio J. Pena

Received: 18 March 2022
Accepted: 13 May 2022
Published: 22 May 2022

1. Motivation

We are about to cross the exascale frontier in High-Performance Computing (HPC). Year after year, the computational power in large computing facilities is progressively increasing, as reflects the TOP 500 list [1]. In this regard, the persistent effort to pave the road to exascale computation can be understood twofold: on the one hand, hardware improvements in memory, storage, or network communications, as well as processor's massive parallelism, are providing brute force to our systems; on the other hand, novel programming models, runtimes, and libraries capable of leveraging these new technologies are improving.

For this reason, it is not enough to reach exascale [2] performance by running a series of benchmarks. If a system is not reliable and does not provide high productivity, it will impair scientific advances and the investment will not be worth it. At the end of the day, exascale computation has to have a real impact on society; for this reason, a smart system able to maintain high throughput in terms of completed jobs per unit of time is crucial. With this sense of productivity, smartness can be regarded as the rapid system adaptation to the constant changes happening in the workload, and the reliability of a system in order to recover from hardware faults in such a complex environment. These two features would mean maintaining in time a high rate of successfully completed jobs.

Firstly, providing more resources opens the door to run larger applications that properly exploit parallelism; however, not all codes can harness an increase in resources. Within the wide range of applications, we also can find production scientific codes with a scalability limit, or even not scalable at all.

Secondly, the growth in the number of computational resources leads to an increase in failure rates. Fault-tolerant techniques will become essential for long-running executions. Several research efforts have studied fault tolerance in exascale systems [3].

Thus, future exascale systems and their applications must have a dynamic behavior that adapts to both the availability of system resources and the needs of the application itself at any given time. That is, the system has to be able to allocate resources dynamically and applications to be adapted to this operation mode at runtime. This versatility will allow to obtain the best performance for the system without harming the application performance. Such type of applications are named malleable or elastic. Current studies conducted in the USA Exascale project [4] show that including this feature in applications will be very profitable for both users and system.

Although these types of applications are now becoming more relevant, their design and performance analysis have been taken into account since quite some time. The beginning of the development of malleable applications took advantage of the solutions designed for fault tolerant systems. This is because the two problems are very similar, as in both cases, applications have to be restarted on a different set of resources (processors); however, malleability counts with a higher degree of complexity than fault tolerance since not only the resources are different but also their amount, also known as size. In the latter case, the overhead of data reallocation is a non-negligible issue to be considered.

On the other hand, in the last years, in the vast majority of HPC facilities, the programming model based on the passing message paradigm has become more and more relevant for designing parallel applications, particularly using the Message Passing Interface (MPI) [4]. The standardization efforts of the MPI forum (https://www.mpi-forum.org (accessed on 7 March 2022)) have allowed applications to be portable across computing systems, and this has led it to be one of the most widely used and the de facto standard of distributed memory programming. In this sense, efforts have also been made to develop tools that allow to easily include malleability in this type of parallel applications.

The main contribution of this paper is to provide a thorough study of the different solutions that have been proposed over the last years in order to design malleable solutions focused mainly on the reallocation of processes at runtime. In this regard, this manuscript is the cornerstone for any researcher willing to contribute to this area of knowledge.

The rest of the paper is structured as follows: Section 2 introduces concepts necessary for understanding malleability and how it has been implemented. Section 3 presents a series of checkpoint/restart mechanisms utilized in process malleability. Section 4 describes deeply MPI-based frameworks that enable malleability in HPC systems. Section 5 introduces the CHARM++ paradigm and how it is leveraged in malleability. Section 6 compiles process malleability solutions based on approaches that cannot fit in the previous sections. Section 7 compares the most remarkable features of the presented malleability solutions. Finally, Section 8 concludes this work.

2. Background

HPC environments are formed by tightly coupled systems that include a cluster of nodes, and accelerators, interconnected via high-speed networks, and parallel files systems with multiple storage tiers.

It is a big challenge for coders to implement applications able to maximize their performance, as well as the efficiency of these complex systems, since parallel applications will have to be prepared to deal with multicore heterogeneous shared resources, distinct types of memory access, parallel I/O, or fault tolerance, among other features.

Applications are submitted to run on HPC systems within the shape of jobs. In this regard, a job is an instance of code, together with the specification of resources needed during its execution, and the input data that have to be processed by the program.

Jobs and resources are controlled by a batch system, which is composed of a job scheduler and a resource manager. Batch systems, also referred to as resource management systems (RMS), are responsible for orchestrating the efficient arbitration of jobs, and maintaining system productivity matching the business rules.

An RMS can include a wide range of options and working modes categorized in different partitions or queues; however, it is usual that RMS follow, in their main queues, very strict policies to allocate resources among running jobs based on two main aspects:

1. **Wait until all requested resources are available**. This causes situations where there are available resources, but are not sufficient to satisfy a job request; therefore, the job has to keep waiting, increasing its waiting time and in consequence its finalization time. In turn, available resources are wasted while waiting to be assigned.
   Notice that this aspect also prevents large jobs from starvation, since not always small jobs, whose resources are available, overtake jobs waiting for remaining resources.
2. **Exclusive resource dedication**. In this case, requested resources are assigned exclusively to a job until its completion. This can lead to have idle resources for some periods during the execution of the jobs, because some applications are decomposed into several phases, and they rarely use all the requested resources in each phase. It is important to take into account this issue to maximize the system productivity.

Clearly, the implementation of these policies can lead to a non-negligible waste of resources since they remain idle but disabled for further usage. In order to deal with this problem, both systems and applications must collaborate to promote a better usage of resources that drives to more productivity.

This work is mainly focused on distributed parallel jobs, in other words, jobs that can concurrently execute a task collaboratively with several processes in different processors. These type of applications can be classified into four groups following the Feitelson and Rudolph [5] categories, depending on who and when determines the size (number of processes) of parallel jobs. In this regard, there exist two actors who can request job reconfigurations in two scenarios. The actors responsible for triggering reconfigurations can be either the system or the users themselves. In turn, the scenarios when reconfigurations can be programmed are on job submission, and during the execution itself. Following, the job classification is detailed:

- **Rigid jobs**: Only can run with a fixed number of processes.
- **Moldable jobs**: Can be initiated with a variable number of processes. The size is determined by the resource manager just before launching the execution of the job.
- **Evolving jobs**: Are provided with a user-defined reconfiguration scheme, specifying how and when the job changes its resources. The RMS must satisfy the requests or the job will not be able to continue its execution.
- **Malleable jobs**. Can be reconfigured during their execution if the RMS decides so.

The first two types use static allocation because resources are maintained during the whole execution, whereas the latter two use dynamic allocation since allocated resources may vary on execution time, and therefore, include in their codes the corresponding resize commands. Rigid and evolving job sizes are determined always by the user, while moldable and malleable are decided by the system. In addition, rigid and moldable jobs cannot be resized during their execution. In this way, malleable jobs are the most flexible since they can be adapted to the cluster workload, being capable of reallocating their resources at any time, increasing the system throughput. In the practice, malleable jobs incorporate synchronization points where reconfigurations can be performed. Moreover, these types of jobs are provided with hints, such as the optimal range of processes, provided by the user to narrow down resizing options.

Malleable jobs are the main goal of study of this work. In order to be able to execute these jobs in a cluster, it will be necessary to have three major components:

1. A **parallel runtime** able to rearrange processes and redistribute data.
2. An **RMS** that supports dynamic reallocation of resources.
3. A **mechanism to coordinate** the previous two subsystems.

Additionally, there are some other aspects to be considered when malleable jobs are executed on an HPC system:

- The increase in the resources does not always assure more throughput; therefore, the RMS has to be prepared to analyze the job performance.
- The incorporation of some fairness in shrink/expand could promote the use of malleability among the users. Thus, the scheduling strategy should maximize the system efficiency including as much fairness as can be delivered.
- Usually, the expansion is faster than shrinking, because in the first the nodes are idle whereas in the second they are executing a task and it has to arrive to a checkpoint.
- Some feedback application can be used by the scheduler to make decisions, although the corresponding overhead has to be considered.

Hence, the balance on the overhead and the improvement of the throughput cannot be ignored and has to be analyzed.

3. Fault Tolerance for Malleability

Fault tolerance and malleability are important issues in high-performance computing. Fault tolerance is especially relevant in large-scale systems, where the likelihood of node failures is extremely high. The term malleability captures applications' ability to incorporate, or release, resources upon system request. This issue becomes crucial in the performance of applications and systems, as it implies a significant overhead.

Both approaches share similar features, because they need to stop an application execution and restart it from the last valid checkpoint. Although from a malleability standpoint, there are two main differences:

1. Malleable applications have to be restarted each time they are resized and checkpoint phases are only triggered when such reconfiguration is required.
2. Reconfigurations may consist of allocating more or few resources. In any case, it is very important to address the data redistribution in the new process layout.

Furthermore, several techniques used in fault tolerance are leveraged in malleability. One of the main ones is the checkpoint and restart technique (C/R). This strategy consists of periodically writing an application state to reliable storage, typically a parallel file system. Upon failure, an application can restart from a prior state by reading its checkpoint file.

In the following, some of the most important efforts to adopt malleability in C/R techniques are described.

3.1. SRS Library and RSS

In [6], the authors present a tool for the development and execution of malleable and migratable parallel MPI applications for distributed systems. This infrastructure consists of a user-level semi-transparent checkpointing library called *Stop Restart Software* (SRS) and the *Runtime Support System* (RSS) that provides the necessary mechanisms for reconfigurating applications during its execution.

The solution is composed of a set of functions that users can invoke from their application to specify the checkpoints and restore the state in the case of a resize. The storage of this information and the redistribution of the data are handled internally by the library and it is transparent to the user.

The novelty of this system relies on its ability to administrate data in different file systems without the need for users to manually migrate checkpoint data. This is achieved through the use of a distributed storage infrastructure called *Internet Backplane Protocol* (IBP) [7], which allows applications to remotely access checkpoint data.

The SRS library consists of six main functions:

- SRS_Init: Has to be called after invoking MPI_Init in the program. This is a collective operation that initializes the data structures used internally and reads various parameters from a user-supplied configuration file that indicate how the application is to be run. This function also contacts RSS to send and receive information about the resources used.

- SRS_Restart_Value: Provides information about the running status of the application. It returns *false* if it starts its execution, and *true* if it continues from a previous checkpoint.
- SRS_Read: Retrieves the checkpoint data and allows the reconfiguration of an application. Different parameters are used to indicate checkpoint file name, location, and type of data distribution.
- SRS_Register: Is used to mark the data that will be stored during periodic checkpointing or when SRS_Check_Stop is called.
- SRS_Check_Stop: Checks if the application has to be stopped because some external component has requested it. It is invoked from various stages of the application and it contacts RSS to obtain this information. It is a collective operation, so if the answer is to stop, SRS_Check_Stop stores the current data registered by SRS_Register to the IBP depots.
- SRS_Finish: Deletes all the local data structures maintained by the library and contacts RSS requesting to terminate the execution. It is called collectively by all the processes of the parallel application before MPI_Finalize.

RSS is a runtime that enables the reconfiguration of the application. It has communicating daemons running on every node where the parallel application is allocated. RSS exists for the entire execution of the application and handles multiple migrations. Before the actual parallel application is started, RSS is launched by the user and will set up, showing a number port on which it listens for requests; therefore, users have to fill out a configuration file containing this information to enable communication. When the parallel application is started on a set of nodes, the main process reads the configuration file with a call to SRS_Init. RSS maintains the current application configuration, as well as the previous executions and an internal flag, called stop_flag, that indicates if the application has to be stopped. A utility named stop_application is provided and allows to stop the application at any point. When running such a utility, RSS sets stop_flag. The application calls the SRS_Check_Stop function to retrieve the stop_flag's value, and according to this running stops or continues. Thus, to develop a malleable application, this function has to be executed in the users' application.

Listing 1 shows a simple MPI parallel program with additional code that invokes SRS library functions that allow us to stop and continue the application with a different number of processes, i.e., it is a malleable application. First, MPI application initializes data to begin execution. Then, the program enters a loop where data are modified. The additional code includes calls to SRS functions in order to initialize environment, mark which data should be stored when the application stops and, finally, indicate the checkpoint where the test will be performed in each iteration. The interactions between the user, the application, and the SRS library and RSS are illustrated in Figure 1.

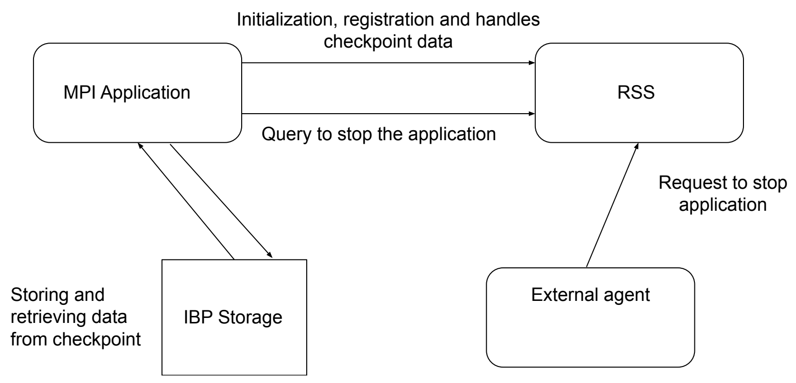
Initialization, registration and handles checkpoint data
MPI Application
RSS
Query to stop the application
Request to stop application
Storing and retrieving data from checkpoint
IBP Storage
External agent

Figure 1. SRS and RSS—interactions between different components in the SRS checkpointing architecture [6]. Based on: http://cds.iisc.ac.in/faculty/vss/publications/vadhiyar-srs-ppletters2003.pdf (accessed on 7 May 2022). Adapted with permission from [6]. Copyright 2003, Sathish S. Vadhiyar.

Listing 1. MPI code modified with calls to SRS in order to include malleabily [6]. Based on: http://cds.iisc.ac.in/faculty/vss/publications/vadhiyar-srs-ppletters2003.pdf (accessed on 7 May 2022).

```
int main(int argc, char** argv){

   MPI_Comm comm = MPI_COMM_WORLD;

   MPI_Init(&argc, &argv);

   // Initialization data and parameters for the application
   // to be reconfigurable.
   SRS_Init();

   // Checking if the application is starting its execution
   // or continued from its previous checkpoint
   restart_value = SRS_Restart_Value();

   if(restart_value == 0){ //application is starting
      // Code for application initialization
   }
   else{ // Application continues from its previous checkpoint and
         // data are retrieved
     // Call to SRS_Read function for each data stored in checkpoint
     // Call to SRS_Read to retrieves iteration to continue
     // execution (iter)
   }

   // Mark the data that will be stored during a checkpointing
   // Call to SRS_Register function for each data that has to be stored
   // in a checkpoint
   // Call to SRS_Register to store iteration where execution will
   // continue (iter)


   for(i=iter; i<global_size; i++){
      // Checkpoint to test if the application has to be stopped
      stop_value = SRS_Check_Stop();
      if(stop_value == 1){ // Execution ends
          MPI_Finalize();
          exit(0);
      }

      // Code for iterative application
   }

   SRS_Finish();
   MPI_Finalize();
   exit(0);
}
```

The authors conclude with an analysis of the overhead of using this library. The experimental results show that the parallel applications transformed into malleable and migratable, were able to be reconfigured incurring at a very low overhead [6]. These experiments were performed on the ScaLAPACK QR factorization application and two clusters that were connected employing the Internet.

3.2. PCM API

The Process Checkpointing and Migration (PCM) API [8] allows iterative MPI applications to be migrated based on the C/R technique [9].

PCM API relies on the reconfiguration capabilities of Internet Operating System (IOS) [10]. IOS is a distributed middleware framework that supports dynamic reconfiguration of large-scale applications via load balancing, and resource-level and application-level profiling. IOS reconfiguration mechanisms provide:

- Analysis of profiled application communication patterns.

- Capture behaviors of the underlying physical resources.
- Reconfiguration of application entities by changing their mappings to other physical resources (migration).

The IOS architecture consists of distributed agents that are capable of interconnecting themselves in different virtual topologies. Moreover, using IOS as runtime, PCM is implemented in the user space to provide portability with MPI-2 standard implementations.

In [11], PCM is extended with routines for splitting and merging MPI processes that eventually enables malleability, being responsible for the spawning and splitting operations, as well as re-arranging communication among processes. Nevertheless, programmers still need to specify the data structures that will be involved in the migration operations regarding the specific ways of subdividing data among processes. Furthermore, programmers also need to guide the split and merge operations for data redistribution. Additionally, PCM API provides four classes of services:

1. Environmental inquiry. Performed by two functions to know the rank and state of each process, one for the state of the application and another one to simplify the data redistribution task.
2. C/R techniques. Performed by two functions to store or load checkpoints to redistribute data. Data are stored with proximity to their destination.
3. Initialization and finalization. Performed by one function to initialize the environment and another one to finalize it.
4. Collective reconfiguration. Performed by four different functions to allow different kinds of reconfigurations using some or all the processes of the application

It also supports data distributions such as block, cyclic, and block-cyclic; however, it is limited to data-parallel programs with a 2D data structure and a linear communication structure.

Users can develop their malleable applications through both: PCM API calls, such as MPI_PCM_Init that initializes internal data structures; a configuration file that has information about the IOS runtime configuration as the port number where accept connections and the location of the PCM daemon.

PCM functions wrap many of the MPI functions/variables. In this regard, coders are expected to use the API following this workflow:

- Check if there is a migration in progress. If so, determine which type of reconfiguration has been scheduled (expansion or shrinkage).
- Spawn new processes and configure them to load data according to the new processes layout.

PCM explores how malleability can be used in C/R applications [12]. Especially, PCM reconfiguration has been implemented and evaluated with a fluid dynamic problem that solves heat diffusion in a solid for testing purposes. The C/R mechanism is leveraged to restart applications with a different number of processes from data stored in checkpoint files.

3.3. Scalable Checkpoint /Restart Extended

Authors in [13] present the scalable C/R (SCR) library, which consists of a novel multi-level checkpoint system that allows us to write checkpoints to RAM memory, flash, or disk on the compute nodes in addition to the parallel file system. This library enables MPI applications to use storage distributed on a system's compute nodes to attain high checkpoint and restart bandwidth.

This technique uses multiple types of checkpoints that have different levels of resiliency and cost in a single application run. The slowest but most resilient level writes the checkpoint file to the parallel file system. Faster but less resilient checkpoint levels utilize local node memory, such as RAM, Flash, or disk to cache these files. So, applications can take frequent inexpensive checkpoints and less frequent and more resilient ones, resulting in better efficiency and reduced load on the parallel system. It also reduces the input/output bandwidth.

SCR library implementation is focused on two key issues:

1. Jobs only need their most recent checkpoint. As soon as the next one is saved in the local storage of the compute node, the previous can be discarded. SCR can also apply a cross-redundancy scheme by caching a local checkpoint file on others nodes. This helps to recover from a failure of a small part of the system. In addition, SCR periodically copies the local checkpoint file to the parallel file system to support failures that disable larger parts of the system.
2. SCR depends on an external service (such as an RMS) to cancel a job that presents a failure. Upon failure, it attempts to recover the most recent checkpoint from a local cache. Assuming it succeeds, SCR can copy that file to the parallel file system and try to restart the execution from the checkpoint. If it fails, it looks for the last file cached in the parallel file system and attempts to restart execution from there.

In [14], authors extend SCR to enable malleability in MPI applications, leveraging the multi-level C/R. In addition, it also permits data to be cached at a specific position in the checkpoint files providing their further redistribution. Originally, SCR was only allowed to retrieve the checkpoint file that matches the MPI rank that wrote the file; however, in [14], it has been modified so that each rank can request and access any checkpoint file on demand. Finally, the malleable application can be restarted with a different number of processes (expand/shrink) performing the required data redistribution.

Authors compare malleability implemented via SCR and User-Level Failure Mitigation (ULFM) library, described next, in Section 3.4. The analysis is focused on malleability in HPC applications to study the effects on execution time as well as resources usage efficiency. The study comprises both techniques applied on two basic applications. These have been a state-of-the-art parallel matrix multiplication algorithm and an application with no computation, but a global matrix shared using a two-dimensional block-cyclic data distribution.

The study involves jobs with long execution times being resized to accommodate short running jobs with higher priority. In this regard, a candidate job is shrunk to free up resources for a high-priority job and then resized back after the latter has finished execution. Thus, the RMS maintains several separate queues according to their priority. The scheduler extracts jobs according to the order of priority and locates the resources needed for their execution. If it is necessary, it will reduce resources to a lower priority job in execution.

Results showed the improvement that malleability can provide in terms of reducing the queuing time for high-priority jobs and also that the ULFM approach enables faster reconfigurations.

3.4. ULFM

User-Level Failure Mitigation (ULFM) (https://fault-tolerance.org/ (accessed on 3 March 2022)) is a proposal for extending the standard MPI with new features that support failure mitigation. These improvements are intended to avoid deadlock in communication operations and return control to the application so it can build its failure tolerance [15,16]. ULFM is the most accepted to integrate this type of application among other projects for MPI and it is highly active in the development of constant advances [3,17].

ULFM incorporates three additional error types:

- MPIX_ERR_PROC_FAILED: This error is returned when a communication operation fails in any process. It is sent to all processes that participated in that operation and have not finished yet. So, the same operation for some processes could return successfully while others do not. For example, the root process in a MPI_Bcast will not detect an error if the operation fails during the reception in the remaining processes. In this regard, processes that still have the communication operation active will receive the error code, but the root will not. From this moment, the failed process will always return this error code for all the operations in its participation.

- MPIX_ERR_REVOKED: Returned by all non-local functions in a communicator/window/file after calling MPI_Comm_revoke/MPI_Window_revoke/MPI_File_revoke for that same handler, respectively.
- MPIX_ERR_PROC_FAILED_PENDING: Returned by non-blocking receiving functions from MPI_ANY_SOURCE, where no send has matched to this operation and a potential send operation from an MPI process has failed. It intends to avoid deadlocks caused by waiting on a failed process or that are on the recovery path.

These errors are only found locally, and not necessarily globally. For non-blocking, one-sided, or I/O functions, errors are notified when processes call synchronous operations.

The main objective of ULFM is to create a recovery path from failures and continue the execution. In this regard, the most relevant functions are described below:

- MPIX_Comm_revoke: Local operation which revokes a communicator. All further non-local operations in this communicator are canceled and return the error MPIX_ERR_REVOKED, which allows other processes to know a problem in this communicator.
- MPIX_Comm_shrink: Creates a new communicator with the same characteristics as a revoked communicator. The new one includes only all non-failed processes.
- int MPIX_Comm_failure_ack(): Local function to acknowledge all errors in a communicator. If there are errors, from this moment all collective operations will return the error MPIX_ERR_PROC_FAILED.

Authors in [16] analyze this approach and show its effectiveness with hardly any degradation in performance in their experiments.

These functions can also be used to implement malleability in users' codes. Although ULFM does not provide any specific mechanism other than those found in the native MPI Standard [18] to expand a job; however, shrinking is enabled with the function MPI_Comm_shrink, which terminates a defined number of processes intentionally. In short, if a job has to be shrunk from n to m processes, $n - m$ processes are forced to fail. Then, invoking MPI_Comm_shrink, the processes are removed from the communicator.

Listing 2 shows how ULFM can be used to achieve malleability. Notice in the listing that there is no data redistribution helper provided by ULFM, as it is not intended for this task; therefore, the application itself has to include the necessary code to carry out this redistribution.

Listing 2. ULFM–Example of ULFM malleability.

```
int main (int argc , char **argv){
  int numP;
  MPI_Comm comm;
  ... // Initialization code
  MPI_Comm_size(comm, &numP);
  ... // Initialization code
  for (...){
    nodeList = get_new_nodelist_somehow();
    // Perform resize only if there is a new node list
    if (nodeList != NULL) {
      MPI_Comm aux_comm;
      if (numP >= nodeList.maxProcs) { // Expand operation
        MPI_Comm_spawn(myapp.bin, nodeList.maxProcs,
                       nodeList.names, &aux_comm);
        ... // Data redistribution code
        exit(0);
      } else { //Shrink operation - ULFM
        ... // Data redistribution code
        kill_N_processes_somehow(numP - nodeList.maxProcs);
        MPIX_Comm_shrink(comm, &aux_comm);
        MPI_Comm_free(comm);
        comm = aux_comm;
      }
    }
    ... // Compute and MPI code
  }
  ... // Finalization code
}
```

In [14], the authors also demonstrate the superiority of ULFM malleable applications in front of SCR malleable counterparts (see Section 3.3), as a reconfiguration is performed in less time with ULFM when expanding and shrinking a job.

4. MPI Malleability

The dynamic creation of processes, in other words, spawning processes from a running application, is available since MPI standard version 2 [18]; however, most RMS operate in ways that make it difficult to add dynamic processes support, which prevents its usage in production systems. For this reason, jobs run with a fixed number of processes and may use load balancing techniques that do not rely on spawning processes. The straightforward integration of processes management in RMS would allow for different load balancing strategies and more efficient use of resources.

The API function provided by the MPI standard to support dynamic processes is MPI_Comm_spawn, which creates processes with an inter-communicator connected to the original invoking communicator. We can also find a variant of this function named MPI_Comm_spawn_multiple, which additionally, supports different binaries or configurations in the same call.

As it is reported in [19], the current process spawning in MPI Standard involves several limitations:

- The spawn operations are synchronous for all processes involved in the invocation of this operation and the new set of processes created.
  Therefore, they are implemented as collective operations that block the processes they invoke while the new ones are created and initialized.
- These operations produce inter-communication based on disjoint processes groups. The majority of applications consider a single flat communicator (`MPI_COMM_WORLD`), from which other intra-communicator could be defined and used in their MPI codes.
- 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. Processes in a group are not destroyed until all of them invoke `MPI_Finalize`.
- Processes created with spawn operations are run in the same resource allocation by default.

Finally, another important issue to consider in this approach is how to redistribute the data among existing and new processes, which is usually performed on the memory of each process during resizing. As a consequence, without proper management, it could affect the performance of the application.

This section describes various efforts that deal with these issues.

4.1. Elastic Execution of MPI Applications

In [19], the authors propose an extension to the MPI Standard 2.0 to better support moldable and malleable applications on large-scale distributed memory systems.

This consists of a new API that extends the MPICH library and allows the dynamic reshaping of the processes' layout.

Following, the new functions are described:

- MPI_Init_adapt: Initializes MPI likewise MPI_Init does, but it has an additional output parameter that informs about how processes are created. If the parameter contains joining, it means that the process was created during the execution. While if it contains new, it means that the process was created when the execution started.

- MPI_Probe_adapt: With this routine, processes can ask the RMS whether a reconfiguration is required when a synchronization point is reached. There is an alternative interface of this routine, which is useful for fault tolerance applications.
- MPI_Comm_adapt_begin: This routine starts the adaptation phase. It retrieves the inter-communicator, equivalent to the generated by spawn routines, and a flat intra-communicator, which will be *MPI_COMM_WORLD* when the adaptation is completed. When shrinking, the intra-communicator of the processes that have to be terminated is set to *MPI_COMM_NULL*. They still are connected to the other processes with the inter-communicator. Management of these two communicators will allow data redistribution required during the adaptation.
- MPI_Comm_adapt_commit: This routine notifies the RMS that the reconfiguration has ended. MPI_COMM_WORLD is changed by the new_comm_world obtained in the previous routine, in which the processes to be destroyed are excluded.

An example of how to implement malleability using these functions is showcased in Listing 3.

Listing 3. Elastic MPI–Structure of a simple malleable application using the elastic execution [19].

```
int main (int argc , char **argv){
  MPI_Init_adapt(&argc , &argv , &local_status );
  for (...){
    MPI_Probe_adapt(&adapt , . . . ) ;
    if ( local status == MPI_ADAPT_STATUS_JOINING
         || adapt==MPI_ADAPT_TRUE){
            MPI_Comm_adapt_begin (...);
           // Redistribution code
          MPI_Comm_adapt_commit(...);
    }
    // Compute and MPI code
  }
}
```

SLURM Integration

Despite the different alternatives when it comes to RMS, the proposed framework is integrated into SLURM [20]. Since other malleability solutions leverage SLURM as well, at this point, it is worth it to describe its architecture:

- The SLURMCTLD daemon is a single centralized component (running on the management node), which is responsible for monitoring each compute node state and for allocating resources to jobs. The rest of the nodes (compute nodes) run the SLURMD daemon, on which the SLURMCTLD notifications arrive. These daemons periodically communicate to the SLURMCTLD to exchange node and job state information.
- Jobs are included in a priority-ordered queue, concerning the criteria of the defined scheduling policy. When a job is selected to be executed, the job is initiated in a compute node, executing an SRUN instance. This instance notifies the other compute nodes that a SLURMSTEPD daemon has to be launched in each node. SLURMSTEPD daemons launch the local processes and interact with them via the Process Management Interface (PMI).

Figure 2 depicts the interactions among SLURM daemons.

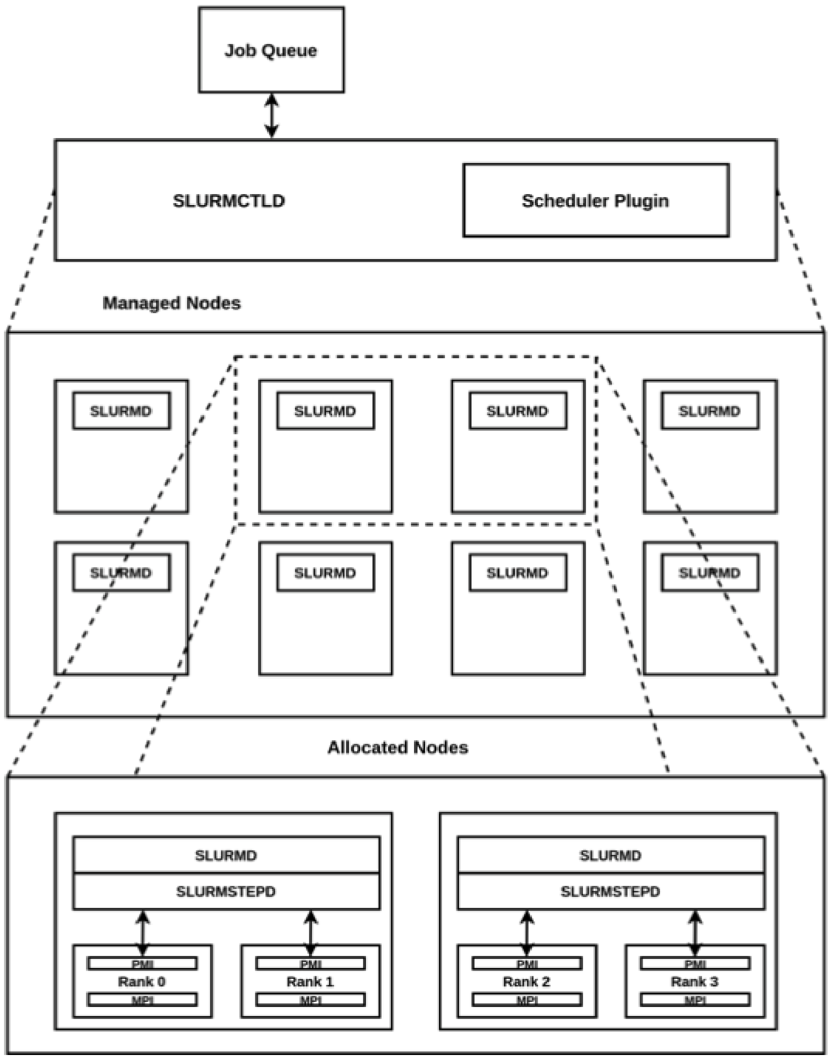

Figure 2. Elastic MPI—overview of the interactions between the SLURM daemons [21]. Source: https://mediatum.ub.tum.de/doc/1539159/file.pdf (accessed on 7 May 2022). Reprinted with permission from [21]. Copyright 2020, Mohak Chadha.

To incorporate elasticity, some modifications to the design and functionality of SLURM are proposed in [19] to support its malleability framework [21,22].

- The batch script language is extended by the following parameters:
  - -min-nodes-invasic defines the minimum number of nodes, such that the job is properly executed.
  - -max-nodes-invasic defines the maximum number of nodes for the job.
  - -node-constraints includes additional constraint on the number of cores. Some values are odd, even, power-of-two, cubic, . . .
- SLURMCTLD is divided in two components:
  - Elastic Runtime Scheduler (ERS) manages expand/shrink operations for malleable jobs. It is also responsible to manage the parameters on which the reconfiguration decision is founded, such as runtime performance or power consumption.
  - Adaptive Batch Scheduler (ABS) is responsible for efficient job scheduling and dynamic reconfiguration decisions for running jobs. It maintains two separate priority-ordered queues, one for rigid and the other for malleable applications.

These components are executed concurrently in two independent threads of the node.

The interaction between them is event-triggered and occurs every *SchedulerTick* second (defined in the configuration file) when a job is submitted or after job completion.

- When an SRUN is launched in a node, a unique port number to a particular thread is communicated to the ERS, on which specific messages will be received.
- Thus, the reconfiguration of a job can be summarized in the following steps:
  1. ERS sends a reconfiguration signal to the SRUN instance.
  2. If a shrinking is notified by ERS, SRUN informs the corresponding SLURMD daemons about the processes to terminate.
  3. If an expansion is triggered, SRUN notifies the corresponding SLURMD daemons about the number of processes to create.
  4. When children processes are executing MPI_Probe_adapt, SLURMD notifies SRUN and the state of the job is changed to *ADAPTING*.
  5. After the execution of MPI_Comm_adapt_commit in all processes, SRUN sends a reconfiguration complete message to ERS.
  6. Finally, ERS sends to SRUN the updated job credentials and updates the job state back to *RUNNING*.

Figure 3 depicts how these stages are followed.

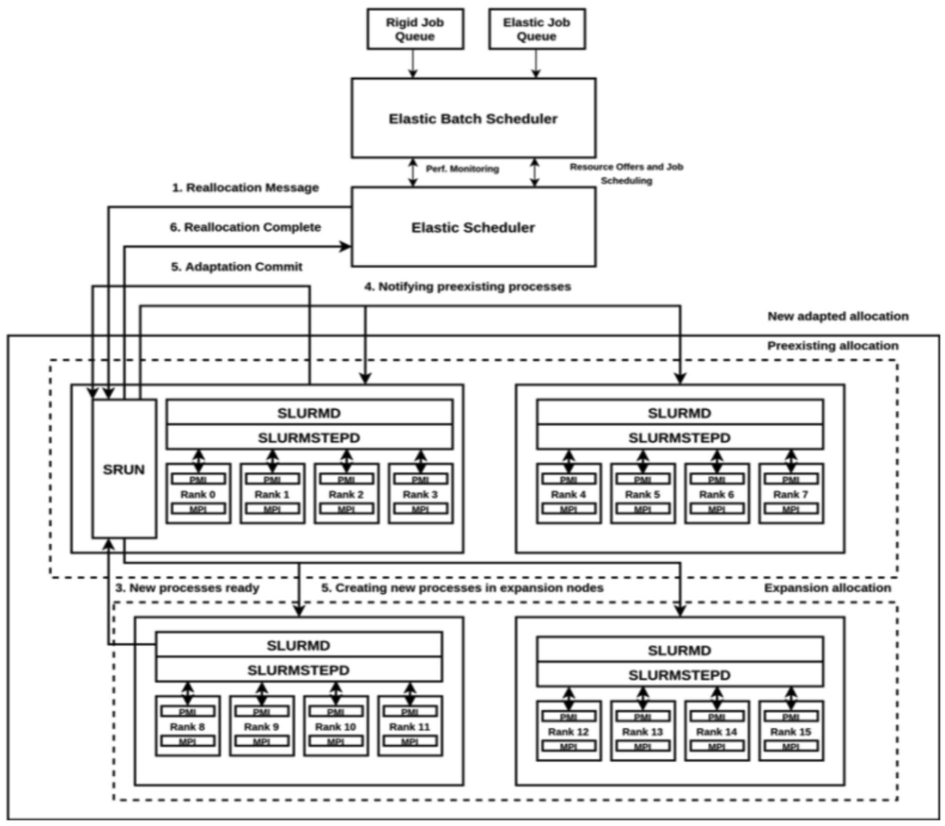

Figure 3. Elastic MPI—overview of interactions between different SLURM daemons during a reconfiguration with the elastic execution [21]. Source: https://mediatum.ub.tum.de/doc/1539159/file.pdf (accessed on 7 May 2020). Reprinted with permission from [21]. Copyright 2020, Mohak Chadha.

The scheduling policies can be based on different criteria. In [22], the evaluation on a cluster of 6480 compute nodes using the standard benchmark and synthetic elastic applications were made. In this experimentation, the authors used two different scheduling strategies:

- The performance-aware strategy considers the MTCT ratio (the time spent in MPI calls versus the time doing relevant computation), as an intuitive heuristic.

- The power-aware strategy requires the correct management of the RAPL counters, specifically updating automatically system files that store energy values.

All this information is stored and managed by ERS component.

4.2. ReSHAPE

ReSHAPE [23] is a coupled solution for malleability that implements a runtime, an API, a RMS, and the communication layer among them.

Unlike other solutions, ReSHAPE does not need to know in advance the data partition sizes for the reconfigurations. This framework calculates partitions based on the runtime performance of the application. Furthermore, ReSHAPE redistributes data among processes using message passing mechanisms which avoid data load/store in the disk.

ReSHAPE framework consists of two main components: an application scheduling and monitoring module and a programming model for resizing applications.

- Firstly, the scheduler allocates resources and gathers jobs performance data to make appropriate decisions. Particularly, the scheduling module is based on the DQ/GEMS project [24], which was extended with job resizing support [25].
- Secondly, the programming model includes the resize library and the API. The library consists of algorithms for mapping processors and communications, and redistributing data. This library is built on top of (BLACS) [26] (Basic Linear Algebra Communication Subprograms), the ScaLAPACK communication library [27], which is extended with support for dynamic process management. The API provides the methods to establish communication with the scheduler module.

All in all, the strong integration of ReSHAPE modules forces users to specifically develop applications for this exclusive system. In this regard, applications need to be compiled with the resize library to enable the scheduler to dynamically add or remove processors.

During a reconfiguration, the application execution control is transferred to the resize library, which maps the new set of processors and redistributes the data. Then, the resize library returns control back to the application following with its next iteration. Users are expected to define global data structures and variables so that they can be redistributed among children processes after the reconfiguration. The steps taken during a resize are the following:

- Application requests a resize.
- The resize library conveys the performance data to the scheduler, which in turn, gathers the application's past performance.
- Scheduler takes an action: expand, shrink, or nothing.
- If it expands, new processes are spawned and added to the context. If it shrinks, processes are terminated, and a new context is created.
- If any action is taken, data have to be redistributed among the new process layout.

In [28], the authors use ReSHAPE to implement malleability in some applications of the Nasa Parallel Benchmark (http://www.nas.nasa.gov/Software/NPB (accessed on 3 March 2022)). Furthermore, in [29], the authors also developed a malleable version of a classical molecular dynamics code, LAMMPS (http://lammps.sandia.gov (accessed on 3 March 2022)), leveraging its native C/R methods.

4.3. Flex-MPI

Flex-MPI [30] is a library, built on top of MPICH [18], to improve *Single Program Multiple Data* (SPMD) application performance using a performance-aware dynamic reconfiguration where users set performance objectives and application constraints.

When an application is executed using Flex-MPI, information related to performance is automatically gathered and analyzed to decide if a reconfiguration would benefit the performance. Furthermore, Flex-MPI also takes care of load-balancing issues, selecting how data will accordingly be partitioned and redistributed among processes.

Flex-MPI is composed of the following modules, which work together under a reconfiguration policy:

- Monitoring: Takes care of gathering data from the application using the library PAPI [31] to retrieve information from hardware counters, and PMPI to collect information from MPI calls. Metrics are collected for each process and are preserved during a context switch.
- Computational prediction model (CPM): Predicts how the application will perform from gathered data when executing on different processor configurations. The result is the expected computation and communication time for the following iterations. It also calculates the expected time for the creation/finalization process and data redistribution.
- Dynamic process management: Controls process creation and termination. Processes are divided between *initial*, which are in the first set of processes created, or *dynamical*, which are created in execution time. Flex-MPI will always have at least an *initial* number of processes working and this group can not be terminated.
  To create a process, the function MPI_Comm_spawn is called with an MPI_Info object to select where the process will be bound. Processes are created one by one to allow fine-grained control over the processes' number and, therefore, satisfy performance constraints. This makes the termination of processes different to the *initial* ones easier; therefore only *dynamic* processes can be removed [30].
  A new process is created with a new inter-communicator connected to the main group of processes, which in turn, is merged into an intra-communicator with the function MPI_Intercomm_merge. So, a new main processes group is started where the recently created process is added, knowing this operation as a "merge" method [32]. The intra-communicator in Flex-MPI is called XMPI_COMM_WORLD and is updated after each process creation or finalization.
  Processes are created only if these criteria are met:
  - There are idle processors in the system.
  - Current performance does not meet the user-given performance objective.
  - CPM predicts a performance improvement to satisfy the user-given performance objective.
- In order to remove a dynamic process, XMPI_COMM_WORLD is released and recreated without the process that is being terminated, which ends the compute and calls MPI_Finalize. The rest of the processes recreate XMPI_COMM_WORLD by invoking the API function MPI_Group_incl, without the removed process.
  Processes are removed only if all this criterion is met:
  - Current performance does not meet the user-given performance objective.
  - CPM predicts a performance reduction that satisfies the user-given performance objective.
- Load balancing: Ensures that data are partitioned among processes having a workload according to the processor they are bound. This module is based on techniques described at [33,34], and creates a partition concerning the computing power of each processor.
- Data redistribution: Redistributes data according to the model set by the load balancing module. The distribution is performed completely by this module and users do not need to describe the data structures; however, Flex-MPI only handles the following types of data: one-dimensional, two-dimensional, and dense or sparse matrices with block-based one-dimensional domain decomposition.

The reconfiguration policy, which controls the previous modules, follows these steps to decide if a resize is necessary:

1. Activation of the monitoring module.
2. Computation of an iteration.
3. Gathering data from the monitoring module.

4. CPM predicts the application performance for the next iterations.
5. Checking if the predicted application performance for the next iteration meets the user performance objective by comparing if the difference is bigger than a given threshold. If it is met, the application continues normally. If not, new resources are requested to the RMS.
6. The CPM calculates different options that meet the goal from the available resources and selects the one that fits best the requirements. Results return the new process quantity and their layout.
7. In the case of an expansion, the following modules are activated:
   (a) Dynamic process management module, to create new processes.
   (b) Load balance module, to select how data will be partitioned in the new layout.
   (c) Data redistribution module, to communicate data accordingly to the previous step.

   When shrinking, the dynamic process management module will be activated after the data have been redistributed among the remaining processes, and then, unnecessary processes will be finalized.
8. Notification of the new layout to the RMS.

Listing 4 shows an example code of how to use Flex-MPI API. Before the computation loop, it can be seen that data are partitioned among processes and registered in Flex-MPI, so it can be performed automatically in future resizes.

For *dynamic* processes, they also obtain their data partitions from other processes. At the beginning of every iteration in the computation loop, function XMPI_Monitor_init() starts gathering application performance metrics for posterior resize evaluations. Finally, at the end of every iteration, it is considered a resize XMPI_Eval_reconfiguration. In the case of shrinking, processes check who should finalize invoking XMPI_Get_Process_status and terminating.

Listing 4. Flex-MPI–Example of use of Flex-MPI [30]. Based on: https://core.ac.uk/download/pdf/41830176.pdf (accessed on 7 May 2022).

```
int main (int argc , char **argv){

  ... // Initialization code
  XMPI_Get_Wsize(...); // Distribute data among processes
  XMPI_Register(...); // Register data structures

  XMPI_Get_shared_data(...);
  // Obtain data from other processes (Dynamic processes only)

  for (...){
    XMPI_Monitor_init(); // Gather application data

    // Compute and MPI code

    XMPI_Eval_reconfiguration(...); // Considers a resize
    XMPI_Get_Process_status(...); // Should the process continue
    execution?
    if(status == EMPI_REMOVED) break; // This process is no longer
    required
  }
  ... // Finalization code
}
```

Classic iterative applications such as Jacobi [35], CG, or EpiGraph [36] have been implemented and evaluated with Flex-MPI. Results showcase how CPM effectively estimates application performance previous resizing and, therefore, improves the efficiency of resource utilization and cost-efficiency of application's executions.

In [37], CPM is extended to allow energy-efficiency constraints, where configurations with better Power Usage Effectiveness (PUE) ratios are favored. Results show that compacting the workload in fewer nodes can reduce energy consumption while maintaining the performance objective.

4.4. DMR

The Dynamic Management of Resources (DMR) [38] malleability framework consists of two main components: an RMS and a parallel distributed runtime based on MPI. DMR provides the communication layer between them and allows malleable application execution in a cluster workload.

SLURM is the RMS used by DMR and, as in other solutions, it is responsible for monitoring the resource utilization and jobs requests. SLURM has been extended with the capability of scheduling malleable jobs and managing dynamic resources.

In a nutshell, job malleability in DMR works as follows:

- When an application is running, it has to periodically contact the RMS in order to show its rescaling willingness.
- This communication is established in a synchronization point, defined by users, where the reconfiguration operation can be initiated.
- Then, SLURM evaluates the global system status to decide whether to perform that reconfiguration and communicate that decision to the runtime.
- If this call resolves to resize the job, SLURM reallocates the resources and returns the resulting number of processes, which can mean to expand or shrink the job.
- Finally, the runtime, with the guidelines of the application, redistributes the data between parent and children, and the job continues the execution with the new process layout.

Furthermore, the current implementation of DMR accommodates three scheduling modes with different degrees of freedom when requesting a resize action:

- Applications are allowed to **strongly suggest** a specific operation. For instance, to expand a job, users are allowed to define the “minimum” number of requested nodes to a value greater than the number of allocated nodes of that job.
- Determine a **sweet spot** number of processes to run an application. Users can configure a malleable application with a preferred size from which the execution cannot be scaled down; however, the scheduler can decide to grant an expansion up to its specified maximum.
- **Wide expansion**:
  - Expand a job as long as there are sufficient available resources if:
    - There is no job pending for execution in the queue.
    - No pending job can be executed due to insufficient available resources.
  - Shrink a job if there is any queued job that could be initiated.

DMR counts with two different approaches: using an API or a library. The API is based on the OmpSs programming model [39] and uses their compiler directives for the user code. The library is a module that can be loaded during the compilation of an MPI program providing the logistics to support malleability. They are described below:

- The DMR API [40] implements a communication layer between the OmpSs runtime (Nanos++ (https://pm.bsc.es/nanox) (accessed on 5 May 2022)) and SLURM that allows the reconfiguration of MPI applications (see Figure 4). This API is based on the OmpSs off-load semantics [41] for automatically managing processes and redistributing data among them.
  Nanos++ runtime is extended with the logic to handle job reconfigurations. In this regard, a new routine, named dmr_check_status, is provided to the users. This function is responsible for triggering the reconfiguration. Moreover, the authors in [42] also present an asynchronous version of the function, such that when an action is scheduled, DMR sets up the new process layout while the application continues its

execution. This version deploys the reconfiguration in the next iteration, to reduce the effective reconfiguration time.
Since DMR API relies on OmpSs, the reconfiguration is conducted by `#pragma` directives, with which, users can define data dependencies and the communication pattern among processes.
Listing 5 showcases the use of this directive where data dependencies are explicitly indicated. After the `#pragma` users indicate the resuming function for continuing with the execution.

Listing 5. DMR–Pseudo-code of job reconfiguration using the DMR API [38].

```

void compute(double *data, int dataSize, int step) {
  for (t = step; t < TIMESTEPS; t++) {
    action =  dmr_check_status(&newComm);
    if (action) {
      #pragma omp task in(data) onto(newComm)
      compute(data, dataSize, t);
    } else
      // Computation
  }
}

```

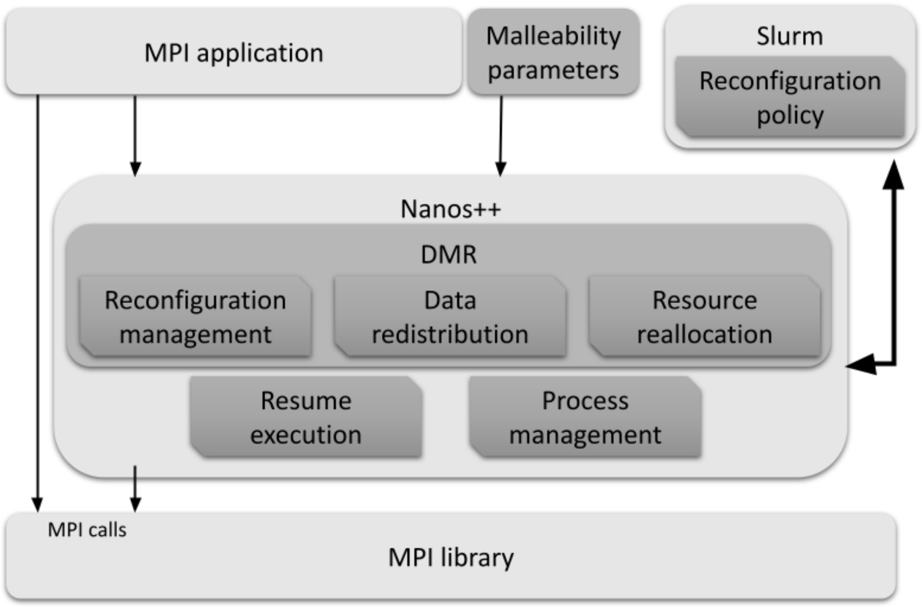

Figure 4. DMR API architecture.

- On the other hand, although the DMR API provides a highly usable interface, irregular applications, such as consumer–producer, may be hard to implement with #pragma directives since not every process features the same data structures.
  DMRlib [43] is designed to ease malleability adoption by application developers. Built on top of the DMR API, DMRlib hides all the interactions among the application, the runtime, and RMS (see Figure 5). For this purpose, the library is in charge of the whole job reconfiguration procedure, honoring the malleability parameters provided by the user.
  DMRlib usage is based on a macro that triggers and handles the whole reconfiguration process. The macro, DMR_RECONFIG, expects five arguments corresponding to five function names:
  - *compute*: Function that will be executed when the reconfiguration ends (usually the same function where the macro is invoked).

– *send_expand*: Function that implements the algorithm for sending data from parent processes to child processes.
– *recv_expand*: Function that implements the algorithm for receiving data in child processes from parent processes.
– *send_shrink*: Similar to the *send_expand* but for shrinking.
– *recv_shrink*: Similar to the *recv_expand* but for shrinking.

Listing 6 showcases how DMRlib could be used in a user code that implements malleability.

Listing 6. DMR–Enabling malleability using DMRlib in a user code [38].

```
void compute(double *data, int data_size, int step) {
    DMR_Set_parameters(minimum, maximum, preferred);
    for (int t = step; t < TIMESPTEPS; t++) {
        DMR_RECONFIG(compute(data, data_size, t),
                     send_expand(data, data_size),
                     recv_expand(&data, &data_size),
                     send_shrink(data, data_size),
                     recv_shrink(&data, &data_size));
        // Computation
    }
}

```

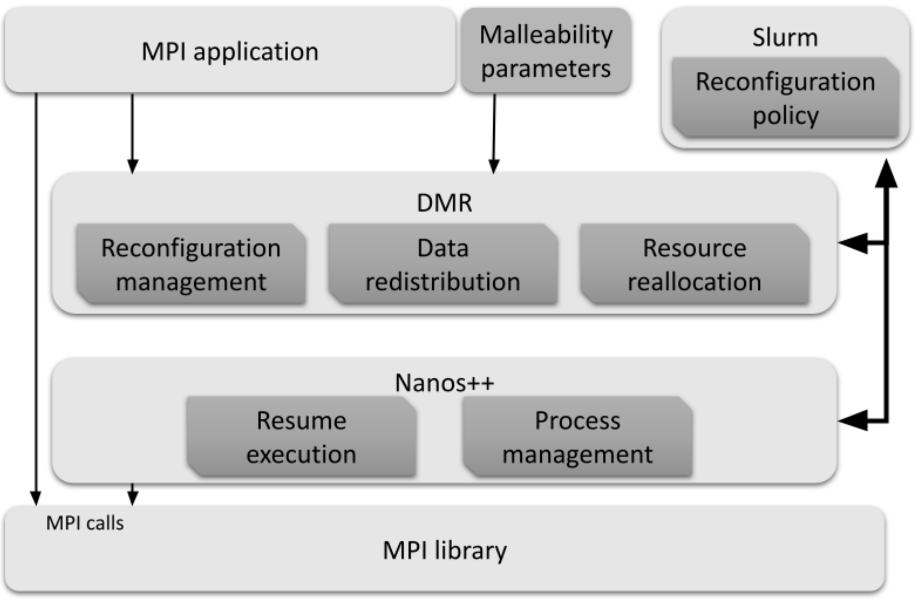

Figure 5. DMRlib architecture.

DMR has been proved an interesting solution to implement malleability in a wide range of applications, not only the traditional iterative applications widely used in malleability, such as Jacobi, CG, N-body [44], LAMMPS, but also producer–consumer bioinformatic applications [45]. Furthermore, DMR has been the first malleability framework to implement reconfiguration in GPU-enabled applications [46].

5. Charm++ Malleability

CHARM++ is a parallel object-oriented programming paradigm designed to enhance programmer productivity delivering high performance. Its key feature is over-decomposition, in other words, to divide the problem into a large number of *chares* (basic unit of computation), which are assigned to physical processors.

Chares can be seen as MPI processes, which communicates through asynchronous remote method invocation, using *entry methods*, which are executed atomically but never block waiting for messages; therefore, CHARM runtime (*Converse*) is a message-driven scheduler that gathers pending messages from a prioritized queue and executes the entry method of the selected object indicated in the message.

A relevant feature in CHARM++ is that data of each chare have to be located in global memory space. CHARM++ forbids the use of global variables; therefore, if a code requires this kind of data, they should be clubbed in a specific global memory part, and all chares have to receive a pointer to that area.

This section describes the steps taken in CHARM++ to support malleability in cluster systems.

5.1. Charm++ Integration into an RMS

CHARM++ leverages Torque/Maui as an RMS to manage and reallocate resources. This integration implied modification in both tools in order to enable malleability.

The Torque/Maui RMS is composed of a set of daemons that are mapped in the different nodes of the cluster. The communication among them allows the correct scheduling of parallel jobs [47]. The main features of scheduling are described below:

- The front-end includes the `pbs_server` daemon and Maui scheduler, whereas the compute nodes execute `pbs_mom` daemon.
- Clients submit jobs using command `qsub` in which the required resources are specified.
- Jobs are incorporated into the corresponding queue of the server.
- Maui chooses the "most suitable" job in the queue to be executed, according to the established scheduling policies.
- A node assigned to the parallel is referred to as *mother superior*, which receives the list of reserved nodes.
- All nodes perform a join operation before parallel job execution starts.

Figure 6 shows the order in which these steps are taken.

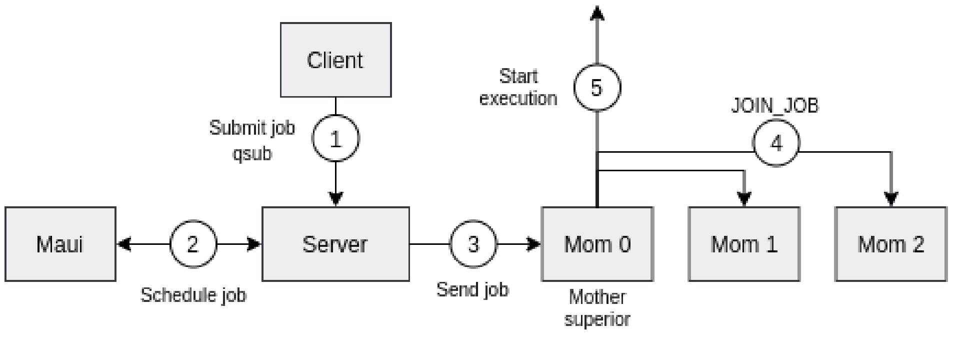

Figure 6. Charm++ and RMS—workflow of the Torque/Maui batch system [47]. Based on: http://charm.cs.illinois.edu/newPapers/15-12/paper.pdf (accessed on 7 May 2022). Adapted with permission from [47]. Copyright 2014, Suraj Prabhakaran.

The integration of CHARM++ in the Torque/Maui RMS to enable malleability required new features to be incorporated [47,48]:

- qsub is extended with an extra option (-L) to specify the upper bound rescaling.
- The communication between runtime and pbs_mom daemons had to be defined:
  - A Converse Client–Server (CCS) interface is designed in CHARM++ to control malleability actions via TCP/IP communication.
  - A CCS client is integrated into each pbs_mom daemon, while a CCS server will run for each job in the pbs_server.
  - The *mother superior* assigns a unique port to communicate the CCS server and a specific parallel job.
- In the scheduler, Maui instructions allow to expand/shrink the resources on the server, according to this algorithm:
  - The scheduler notifies the pbs_server that a resource reconfiguration is required.

 – The pbs_server incorporates the modifications and forwards the information to *mother-superior* node.

- The pbs_mom daemon of the *mother-superior* reallocates the resource according to the received information.
- When the reconfiguration is completed, a *CCSExpand/CCSShrink* message is sent to the CCS server. Notice that *CCSExpand* can be immediately sent, but *CCShrink* has to be sent only when the data from removed nodes are fully retrieved.

Figures 7 and 8 depict how a job expansion and shrinkage are performed, respectively.

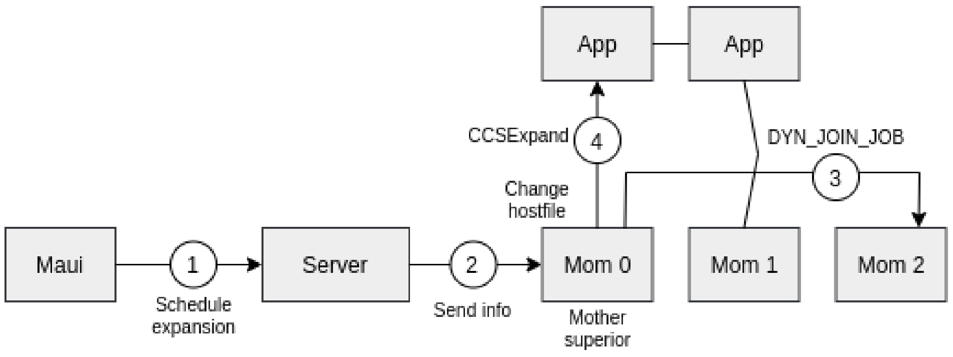

Figure 7. Expanding a job by adding nodes 2 and 3 [47]. Based on: http://charm.cs.illinois.edu/newPapers/15-12/paper.pdf (accessed on 7 May 2022). Adapted with permission from [47]. Copyright 2014, Suraj Prabhakaran.

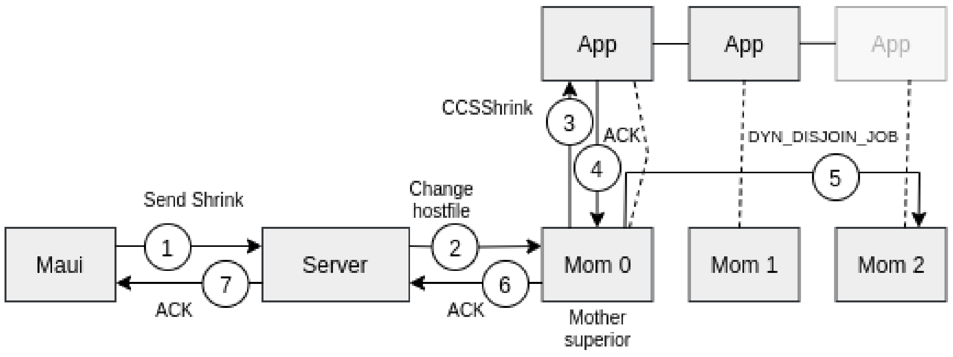

Figure 8. Charm++ and RMS—shrinking a job by removing nodes 2 and 3 [47]. Based on: http://charm.cs.illinois.edu/newPapers/15-12/paper.pdf (accessed on 7 May 2022). Adapted with permission from [47]. Copyright 2014, Suraj Prabhakaran.

The authors complete their study with a performance evaluation of malleable and evolving jobs analysis in a workload using standard benchmark and synthetic applications.

5.2. Adaptive MPI

Adaptive MPI (AMPI) [49,50] is presented as a regular MPI implementation (such as MPICH or OpenMPI) that allows MPI developers to leverage CHARM++'s dynamic runtime system. In this regard, MPI applications, with zero to minimal changes, are granted support for process virtualization, the overlap of communication and computation, load balancing, and fault tolerance [51,52].

There are different options to implement CHARM++ chares: processes, kernel-threads, and user-threads. AMPI has chosen the latter because it provides full control over scheduling. Moreover, with user threads, it is also possible to maintain information on the communication pattern among chares, and their computational and memory requirements.

As has been introduced before, each chare has local data and references to global memory that are stored in the stack of the thread. The success of the migration is achieved by copying data and references from one thread to another and keeping consistent the allocations in the new mapping. In order to convert an MPI code to AMPI, global and static variables have to be transformed as private variables in each chare [49,50]. AMPI proposes the use of an scalable variant of the *isomalloc* functionality PM^2 [53]. This technique consists of allocating the thread's stack, in such a way that it spans the same reserved virtual addresses across all the processors. Thus, the total virtual address space is split among physical processors with the same virtual addressing. For this reason, during migrations, memory locations do not change among threads.

Finally, it should be noted that the management of the virtual address space is based on the use of *mmap* and *munmap* functionality of Linux. Thus, the allocation routines are intercepted and handled by the runtime which locates the memory reservation in the corresponding virtual address.

Natively, CHARM++ operations are non-blocking. To enable blocking, operations are expected to be split into two different chares. The first one implements the same operation in a non-blocking manner, whereas the second contains the task after blocking, thus, when the first is completed, the next starts. The use of blocking MPI operations could considerably alter the structure of the program, therefore, to avoid this problem,

AMPI includes specific implementations of these routines to facilitate their use.

AMPI includes several routines as extensions to the MPI standard, in which AMPI_Migrate is the most relevant for load balancing and malleability. This is a collective operation where all chares have to execute with the same parameters. Then, the runtime analyzes the load of the physical processors, and the corresponding load balancing strategy is applied. As a result, chares could be scheduled for migration, and consequently, the job rescaled.

AMPI_Migrate has a single parameter (MPI_Info), which determines the operation. There are four different possibilities. Thus, two built-in MPI_Info objects, called AMPI_INFO_LB_SYNC and AMPI_INFO_LB_ASYNC, which, respectively, determine whether the load balancing is at a synchronization point (see Listing 7).

Moreover, MPI_INFO_CHKPT_IN_MEMORY is another built-in MPI_Info object that forces to perform a checkpointing in memory. Finally, a specific MPI_Info object can be created with the key "ampi_checkpoint" to enable checkpointing in non-volatile memory.

Listing 7 shows a usage example to perform a migration in a malleable application leveraging AMPI_Migrate.

There exists an extend catalog (https://charm.readthedocs.io/en/latest/ampi/05-examples.html (accessed on 13 April 2022)) of examples and applications for AMPI.

Listing 7. AMPI–Structure of a simple AMPI malleable application.

```
int main (int argc , char **argv){
  ... // Initialization code
  for (...){
    MigrationAnalisisRequired = ...;
    if (MigrationAnalisisRequired) {
      AMPI_Migrate( AMPI_INFO_LB_SYNC );
    }
    ... // Compute and MPI code
  }
  ... // Finalization code
}
```

6. Other Malleability Solutions

In this section, we present some other interesting efforts on malleability that cannot directly fit in the previous sections since they are not based on MPI, are not intended for cluster computing, or are ad hoc malleability implementations.

6.1. Invasive Malleability

Invasive computing paradigm [54] suggest a resource-aware programming model, where the application can dynamically fit to the available resources, e.g., processing elements, memory, and network connections. The request of these resources is named *invade*.

Such resources invasion can be indicated by constraints. Next, application uses the invaded resources for a certain computation. This action is named *infect*. When the resources are not needed anymore, application frees them. This is known as *retreat*.

Applications developed by this paradigm must be extended in three ways: (i) applications must state its constraints and give hints to resource manager, (ii) resource adaptations must be supported by the application, and (iii) application must handle data migration, which results from resource adaptations. A resource manager is necessary to make decisions on how to allocate resources based on application constraints during execution.

Originally, invasive computing (http://www.invasic.de (accessed on 13 April 2022)) was applied on a multiprocessor system-on-chip, with custom hardware [55] and operating system [56]. Its main goal is to write applications in a cooperating and resource-aware manner, where resource requirements are explicitly communicated to the system. Leveraging its global view, the system decides for the best action, and order the hardware and applications to adapt. Most of these works have been developed using the programming language X10.

X10 [57] is an object-oriented programming language in the *PGAS family* [58] that supplies important tools to support the field of scientific computing with parallelization considered from the very beginning.

One of its most interesting feature for malleability is the flexible treatment of concurrency, distribution, and locality, within an integrated type system. X10 also extends the PGAS model with asynchrony, yielding the *APGAS programming model* [59]. This feature introduces *places* as an abstraction for a computational context with a locally synchronous view of shared memory. An X10 computation runs over a large collection of *places*. Each place hosts some data and runs one or more *activities*.

The main features of X10 are as follows:

- Activities within the same place use shared memory, whereas elsewhere activities must communicate with other means.
- An activity may synchronously (and atomically) use one or more memory locations in the place in which it resides, leveraging current symmetric multiprocessor (SMP) technology.
- An activity may shift to another place to execute a statement block. X10 provides weaker ordering guarantees for inter-place data access, enabling applications to scale.
- Multiple memory locations in multiple places cannot be accessed atomically. Immutable data need no consistency management and may be freely copied by the implementation between places.
- One or more clocks may be used to order activities running in multiple places. Distributed arrays, `DistArrays`, may be distributed across multiple places and support parallel collective operations.

Authors in [60] have developed an invasive application near to malleable applications using this programming language. It consists of a resource manager, as well as a multigrid solver, with dynamical resource demands. It can be executed on common HPC many-core and cluster systems. Such a solver has to modify its scalability behavior during the execution and requires data migration upon reallocation due to distributed memory systems, in the same way as a malleable application behaves. Their experimental results reflect improved application throughput thanks to the dynamic adaptation of resources.

An invasive application features three steps [61] as it is shown in Listing 8:

1. A call to `invade` claims resources specified by various *constraints*. This set of resources is called a *claim*.
2. A call to `infect` uses claimed resources for executing *i-lets* on them. *i-lets* are self-contained units of computation that run until completion.
3. The *retreat* gives the claim's resources back to the system.
4. Optionally, *reinvade* can change constraints in between and allows the system to adapt a claim. This method is one of the major mechanisms for resource-aware programming and allows for a reallocation of resources with two possibilities as shown in Listing 9.

On the one hand, it allows the RMS to reallocate resources in front of a change of situation. On the other hand, this method also offers the possibility to change the resource constraints for an application providing a new specification.

An example of an invasive architecture for HPC systems is depicted in Figure 9, where the X10 invasive framework is responsible for resource management. Applications request resources with appropriate constraints by *invade* and its claim is set for each application. The resource layer represents invaded resources assigned exclusively to an application without interfering with others.

Listing 8. Invasive Malleability–The basic idea of invasive programming [60]. Based on: https://pp.ipd.kit.edu/uploads/publikationen/bungartz13invasive.pdf (accessed on 13 April 2022). Adapted with permission from [60]. Copyright 2013 Hans-Joachim Bungartz.

```
parallel_function () {
   // Application code executed by each compute resource allocated
}
// Setting resources available to the application from the constraints.
// Define claim
claim = Claim.invade(constraints)

// Allocate those resources indicating function to run
claim.infect(parallel_function)

// Free resources not used
claim.retreat()
```

Listing 9. Invasive Malleability–reinvade method allows resources reallocation [60]. Based on: https://pp.ipd.kit.edu/uploads/publikationen/bungartz13invasive.pdf (accessed on 13 April 2022). Adapted with permission from [60]. Copyright 2013 Hans-Joachim Bungartz.

```

// First Setting resources available for the application and allocation
claim = Claim.invade(constraints)
claim.infect(parallel_function)

// New resource allocation and allocation
claim2 = claim.reinvade(Other_constraints)
claim.infect(parallel_funtion)

// Define other resource needs
claim3 = claim.reinvade(Other_Constraints_new)
```

An extension of this work is presented in [62] where asynchronously malleable applications are introduced. They can be reconfigured without specifying synchronization points. The work describes how master–slave applications meet the requirements for asynchronous malleability and how invasive computing supports that. In an asynchronously malleable invasive application, the system can decide at any time to resize a *claim*.

For example, requiring an asynchronously malleable claim of four to ten processing elements means the system can resize to five or nine processing elements at any time. In contrast to normal claims, an asynchronously malleable claim must be adaptable even within an active infect phase. The application is responsible for starting and terminating *i-lets* concerning added and removed computational units.

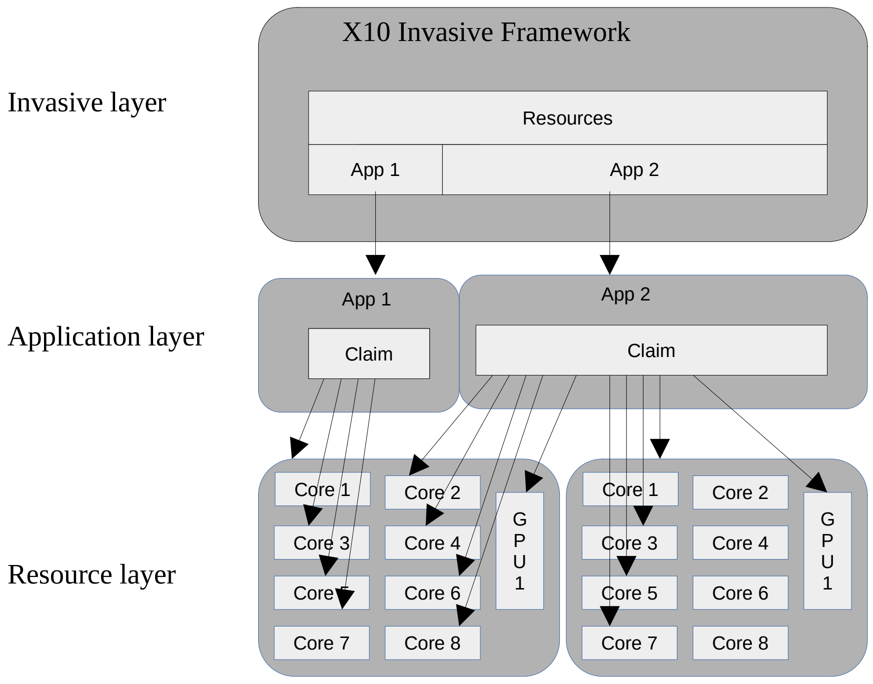

Figure 9. Invasive Malleability—invasive architecture developed whit X10 invasive framework [60]. Based on: https://pp.ipd.kit.edu/uploads/publikationen/bungartz13invasive.pdf (accessed on 13 April 2022). Adapted with permission from [60]. Copyright 2013, Hans-Joachim Bungartz.

6.2. EasyGrid Application Management System

In [63], the authors leveraged EasyGrid framework [64] to transform a traditional moldable MPI application into an autonomic malleable version that inherits the benefits of both malleability [12] and autonomy [65]. As such, applications will thus be capable of adjusting the number of their processes, if appropriate, and/or reallocating them to be adapted to the applications' requirements, resources availability, or resource failures in the system. The most notable point is that this reconfiguring capability does not depend on an external system nor the ability of the programmer, but an application management system.

EasyGrid was designed to generate a system-aware version of the application capable of using the `Grid` resources available to the user in the most appropriate manner. The EasyGrid methodology uses a middleware oriented to the application. This means the service middleware is part of, and specific to, each particular grid application. A *application management system* (AMS) distributed within each system-aware application will be will be responsible for the efficient use of resources.

The AMS EasyGrid uses Local Area Multicomputer library (LAM), an predecessor of modern MPI standard implementations. It transforms applications into system-aware versions, capable of adapting their execution under changes in the grid environment. AMS does not create all application processes at once, but rather according to an application-specific scheduling policy. Since this is transparent to users' MPI programs, AMS must also manage all communications between processes.

EasyGrid AMS integrates functional layers for monitoring processes and resources, and for dynamic reshaping the processes layout of a job. It employs over-provisioning, in particular, to harness heterogeneous resources, and temporal process partitioning where long-running processes are divided into a series of subprocesses; therefore, processes are only created when they are ready for execution.

Figure 10 depicts how to generate an EasyGrid system-aware application from MPI source code.

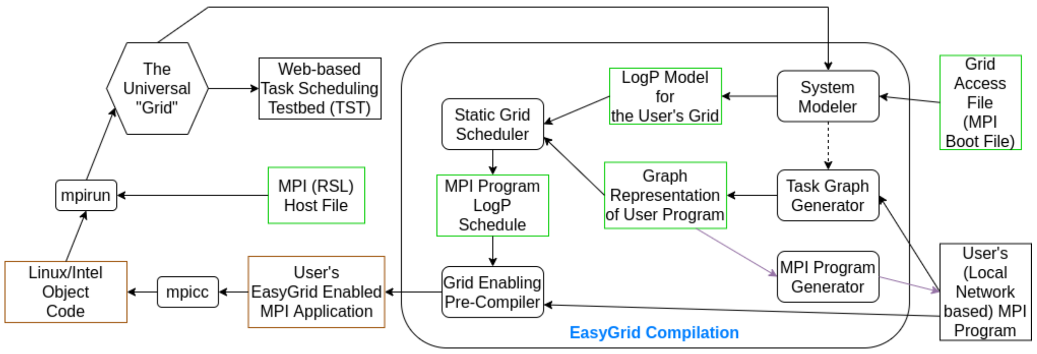
The Universal "Grid"
Web-based Task Scheduling Testbed (TST)
mpirun
MPI (RSL) Host File
Linux/Intel Object Code
mpicc
User's EasyGrid Enabled MPI Application
Static Grid Scheduler
LogP Model for the User's Grid
System Modeler
Grid Access File (MPI Boot File)
MPI Program LogP Schedule
Graph Representation of User Program
Task Graph Generator
MPI Program Generator
Grid Enabling Pre-Compiler
User's (Local Network based) MPI Program
EasyGrid Compilation

Figure 10. EasyGrid AMS—framework [64]. Based on: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.536.1496&rep=rep1&type=pdf (accessed on 13 April 2022). Adapted with permission from [64]. Copyright 2004, Cristina Boeres.

The authors in [63] extend EasyGrid AMS to support process malleability. For this purpose, AMS, periodically, evaluates the need to rescale a job based on the monitoring information collected during its execution. Thus, to incorporate malleability in this environment, a new set of functions were added in order to:

- Define reconfiguration points where applications can be rescaled.
- Design the reconfiguration mechanism that calculates the new processes layout in the next time step based on information gathered in previous ones.
- Define the moment during the execution when the reconfiguration mechanism will be called (invocation point).

Experimental evaluations demonstrated that the EasyGrid AMS enhanced version allows the creation of a malleable version of an application (in this particular case, N-body simulation), and it is capable of automatically rescaling it by reconfiguring processes and reallocating resources. In addition, results presented a low overhead that decreases as the problem size increases.

6.3. COMPSs-Based Solution

In [66], the authors develop an ad hoc malleable solution for Alya [67], a CFD software. The work leverages the COMPSs (COMP Superscalar) runtime [68,69] along with the TALP library [70] to enable performance-aware malleability. The solution is based on the Alya native support of C/R to dynamically resize the number of resources.

COMPSs is a programming model and runtime system that allows users to write applications in a sequential paradigm, which is later used in runtime to exploit parallelism inherent to the sequential code. Its objective is to avoid users from addressing concurrency, so they are not expected to deal with threading, messaging, or data redistributions. Users are still required to annotate which functions should be executed as asynchronous parallel tasks.

The runtime was developed initially for grid computing [69] and later on for cloud computing [71]. Nevertheless, the malleable solution in [66] only uses a portion of its features, as only are needed those to communicate with the RMS to allocate or free resources for the user application Alya. The version of COMPSs used is PyCOMPSs [72].

The application Alya has been parallelized with MPI and has an internal feature to create a checkpoint to the filesystem by using MPI-IO functions. Later on, it can be restarted with any amount of processes and data will be redistributed accordingly.

Alya is also compiled along with a library called TALP, which monitors the application parallel efficiency in runtime, and if it does not reach a user-defined communication efficiency threshold, Alya will ask COMPSs for more or fewer resources, depending on the performance results.

The communication between Alya and COMPSs is established via a file. COMPSs periodically reads if Alya has written the possibility for a resize, indicating that it, with the help of TALP, has detected an efficiency improvement. When a resize is scheduled, it is checked whether the job will be expanded or shrunk. If it is expanded:

- COMPSs communicates with the RMS to obtain new resources.
- Meanwhile, the application continues its execution.
- When the resources are available, COMPSs sends the signal *SIGTERM* to Alya, which indicates the application to create a checkpoint and stop its execution.
- COMPSs launches again the application with the new resources layout and the created checkpoint.

If the job is shrunk:

- COMPSs sends the signal *SIGTERM* to Alya, which indicates the application to create a checkpoint and stop its execution.
- COMPSs launches again the application without the unneeded resources and the created checkpoint.
- The application continues its execution while COMPSs frees the unneeded resources.

This solution has only been tested with the CFD application Alya, which has been parallelized purely with MPI.

7. Discussion

This section summarizes and compares the previously described malleability solutions. Table 1 compiles the most interesting features of the described malleability solutions regarding the motivation of this manuscript and the importance of process malleability in the upcoming exascale systems. The table contains information about the programming paradigm (column 4) presented to users for the application implementation; how the data transfers are performed (column 5); in which RMS the solution is integrated, if any (column 6); if solutions support performance and power aware decisions, and fault tolerance (columns 7, 8, and 9, respectively).

Table 1. Process malleability solution feature comparison.

Type	Solution	Sec.	Prog. Paradigm	Data Transfers	RMS	Perf. Aware	Power Aware	Fault tol.
C /R	SRS	Section 3.1	MPI	Implicit	No	No	No	Yes
	PCM API	Section 3.2	MPI (ad hoc)	Implicit †	No	No	No	Yes
	SCR	Section 3.3	MPI (ad hoc)	Explicit	Yes	No	No	Yes
	ULFM	Section 3.4	MPI (ad hoc)	Explicit	No	No	No	Yes
Dynamic Resize	Elastic MPI	Section 4.1	MPI (ad hoc)	Explicit	Slurm	Yes	Yes	Yes
	ReSHAPE	Section 4.2	ad hoc	Implicit	ad hoc	Yes	No	No
	Flex-MPI	Section 4.3	MPI (ad hoc)	Implicit	No	Yes	Yes	No
	DMR API	Section 4.4	OmpSs	Implicit	Slurm	No	No	No
	DMRlib	Section 4.4	MPI	Implicit †	Slurm	No	No	No
	CHARM++	Section 5.1	CHARM++	Implicit	Torque	Yes	No	Yes
	AMPI	Section 5.2	MPI	Implicit †	Torque	Yes	No	Yes
	Invasive computing	Section 6.1	X10	Explicit	ad hoc	Yes	No	No
	Easygrid AMS	Section 6.2	LAM/MPI	Implicit	AMS (ad hoc)	Yes	No	Yes
	Alya malleable	Section 6.3	COMPSs	Implicit	Slurm	Yes	No	Yes

Notice that MPI (ad hoc) programming paradigm refers to modified implementations of MPI that fit the needs of the specific solutions. With regard to the data transfers, two categories have been defined:

- Explicit: Users are expected to fully implement the data redistribution among processes.
- Implicit: Data redistribution among processes is performed and is transparent to users. Thus, user applications are not expected to be modified. For instance, users can leverage data structure registers or data dependencies of the paradigm to enable

automatic data redistributions; however, several solutions are marked as *Implicit*†, which means that automatic data redistributions are performed in many cases, but not in any case. For instance, PCM API automatic data transfers are limited to 2D linear redistributions, or DMRlib does not provide an specific mechanism for non-standard redistribution patters.

Nearly all of the solutions tend to provide an implicit data redistribution mechanism as this could be a starting barrier for users to use malleability. Simplifying data redistribution as much as possible makes malleability more attractive to incoming users, but it also proves a challenge for malleability developers.

Many of the solutions provide fault tolerance capabilities as they were initially created with this intention in mind. In [4], nearly half of the projects use checkpoint techniques to provide fault tolerance, since they consider it a need for the potential increase in failures in Exascale systems. This describes the obligation of future malleability solutions to provide at least basic fault tolerance techniques in order to reach to a broader audience.

The adoption of performance, or power, aware techniques are interesting in terms of making smart decisions when reconfiguring jobs. Instead of letting users manually define the malleability boundaries, the runtime is able to evaluate resize actions depending on the application performance compared to previous registered performances or power limitations stated by the system administrator. These techniques provide a better resource utilization of the application and, at the same time, of the system, which is one of the key points malleability that researchers would like to improve; therefore, newer solutions should ensure that the performed resizes induce a better resource utilization by monitoring the applications in execution time within the defined power budget.

It is interesting to further describe the case of the Alya malleable solution. It appears to be one of the most complete, since it implements automatic data transfers, is compatible with Slurm, takes decisions based on performance metrics, and is resilient with C/R methods. Although this solution is not generalist and it has been implemented only for Alya, it has been included in the survey as an inspiration source of future developments in the field of process malleability.

8. Conclusions

This paper was created to provide an updated state-of-the-art account of the most significant efforts in process malleability. Its main goal was to introduce and describe how different frameworks address process malleability, their integration into an RMS, and how they can be used in our codes, as long as documentation has been publicly published.

Process malleability in HPC follows a long journey, starting from manual C/R methods, to more complex solutions integrated into RMS with autonomous data redistribution among processes, and with awareness to performance and efficiency metrics.

As we have seen, despite the variety of solutions, leveraging process malleability is not as straightforward as running your existent MPI codes and expecting to be automatically rescaled. It always requires some adaptation in the code and/or in the system, which supposes a critical entry barrier for malleability being widespread in HPC systems.

Particularly, data redistribution is presented as one of the most challenging issues for processes reconfiguration, not only with algorithms that improve computational load balance [73,74], but also with usable codes that perform optimal data redistribution without changes in the original codes [38]. This also allows us to use load-balancing techniques to ensure better resource utilization. Upcoming solutions should keep this trend, as not all final users in exascale systems will be willing to modify their applications to add their own data redistribution mechanisms.

All in all, ideally, a malleability solution not only should implement completely implicit data transfers, be compatible with most popular HPC RMS, consider for taking reconfiguration decisions, and provide fault tolerance mechanisms, but also be incorporated into the standard of an HPC ubiquitous programming paradigm such as MPI.

In this regard, distributed malleability has still a long path ahead, particularly if topics such as data locality for redistribution [73], asynchronous process management [75], in-memory C /R fault tolerance methods [76], or intranode load balancing [77] are included and combined with malleability.

Author Contributions: Conceptualization, J.I.A., M.C., S.I., I.M.-Á. and R.M.; Formal analysis, J.I.A., M.C., S.I., I.M.-Á. and R.M.; Investigation, J.I.A., M.C., S.I., I.M.-Á. and R.M.; Methodology, J.I.A., M.C., S.I., I.M.-Á. and R.M.; Resources, J.I.A., M.C., S.I., I.M.-Á. and R.M.; Supervision, J.I.A., M.C., S.I. and R.M.; Validation, J.I.A., M.C., S.I. and R.M.; Visualization, S.I.; Writing—original draft, J.I.A., M.C., S.I., I.M.-Á. and R.M.; Writing—review & editing, J.I.A., M.C., S.I., I.M.-Á. and R.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research 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.Martin-Alvarez was supported by the predoctoral fellowship ACIF/2021/260 both from Valencian Region Government and European Social Funds.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Dongarra, J.; Luszczek, P. TOP500. In *Encyclopedia of Parallel Computing*; Padua, D., Ed.; Springer: Boston, MA, USA, 2011; pp. 2055–2057. [CrossRef]
2. Balaprakash, P.; Buntinas, D.; Chan, A.; Guha, A.; Gupta, R.; Narayanan, S.H.K.; Chien, A.A.; Hovland, P.; Norris, B. Exascale workload characterization and architecture implications. In Proceedings of the 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Austin, TX, USA, 21–23 April 2013; pp. 120–121. [CrossRef]
3. Losada, N.; González, P.; Martín, M.J.; Bosilca, G.; Bouteiller, A.; Teranishi, K. Fault tolerance of MPI applications in exascale systems: The ULFM solution. *Future Gener. Comput. Syst.* **2020**, *106*, 467–481. [CrossRef]
4. Bernholdt, D.E.; Boehm, S.; Bosilca, G.; Gorentla Venkata, M.; Grant, R.E.; Naughton, T.; Pritchard, H.P.; Schulz, M.; Vallee, G.R. A survey of MPI usage in the US exascale computing project. *Concurr. Comput. Pract. Exp.* **2020**, *32*, e4851. [CrossRef]
5. Feitelson, D.G.; Rudolph, L. Toward Convergence in Job Schedulers for Parallel Supercomputers. In *Job Scheduling Strategies for Parallel Processing*; Springer: Berlin/Heidelberg, Germany, 1996; Volume 1162, pp. 1–26.
6. Vadhiyar, S.S.; Dongarra, J.J. SRS: A framework for developing malleable and migratable applications for distributed systems. *Parallel Process. Lett.* **2002**, *2*, 291–312. [CrossRef]
7. Plank, J.S.; Beck, M.; Elwasif, W.R.; Moore, T.; Swany, M.; Wolski, R. The Internet Backplane Protocol: Storage in the Network. In Proceedings of the Network Storage Symposium, Portland, OR, USA, 14–19 November 1999.
8. El Maghraoui, K.; Szymanski, B.K.; Varela, C. An Architecture for Reconfigurable Iterative MPI Applications in Dynamic Environments. In Proceedings of the International Conference on Parallel Processing and Applied Mathematics, Columbus, OH, USA, 14–18 August 2006; pp. 258–271.
9. Plank, J.S.; Beck, M.; Kingsley, G.; Li, K. Libckpt: Transparent Checkpointing under Unix. In Proceedings of the Usenix Winter Technical Conference, Orleans, LA, USA, 16–20 January 1995; pp. 213–223.
10. Desell, T.; El Maghraoui, K.; Varela, C. Load Balancing of Autonomous Actors over Dynamic Networks. In Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04), Big Island, HI, USA, 5–8 January 2004; Volume 9, p. 90268.1.
11. El Maghraoui, K.; Desell, T.J.; Szymanski, B.K.; Varela, C.A. Dynamic Malleability in Iterative MPI Applications. In Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid), Rio de Janeiro, Brazil, 14–17 May 2007; pp. 591–598.
12. El Maghraoui, K.; Desell, T.J.; Szymanski, B.K.; Varela, C.A. Malleable Iterative MPI Applications. *Concurr. Comput. Pract. Exp.* **2009**, *21*, 227–240. [CrossRef]
13. Moody, A.; Bronevetsky, G.; Mohror, K.; de Supinski, B.R. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC10), Washington, DC, USA, 13–19 November 2010.
14. Lemarinier, P.; Hasanov, K.; Venugopal, S.; Katrinis, K. Architecting Malleable MPI Applications for Priority-driven Adaptive Scheduling. In Proceedings of the 23rd European MPI Users' Group Meeting (EuroMPI), Edinburgh, UK, 25–28 September 2016; pp. 74–81.

15. Bland, W.; Bosilca, G.; Bouteiller, A.; Hérault, T.; Dongarra, J. A Proposal for User-Level Failure Mitigation in the MPI-3 Standard. 2012. Available online: https://www.semanticscholar.org/paper/A-Proposal-for-User-Level-Failure-Mitigation-in-the-Bland-Bosilca/3e697ef781bbce707e3729185a062fb3f580309e (accessed on 13 April 2022).
16. Bland, W.; Bouteiller, A.; Herault, T.; Hursey, J.; Bosilca, G.; Dongarra, J.J. An Evaluation of User-Level Failure Mitigation Support in MPI. In *Recent Advances in the Message Passing Interface*; Träff, J.L., Benkner, S., Dongarra, J.J., Eds.; Springer: Berlin/Heidelberg, Germany, 2012; pp. 193–203.
17. Bland, W.; Bouteiller, A.; Herault, T.; Bosilca, G.; Dongarra, J. Post-failure Recovery of MPI Communication Capability: Design and Rationale. *Int. J. High Perform. Comput. Appl.* **2013**, *27*, 244–254. [CrossRef]
18. Gropp, W. *MPICH2: A New Start for MPI Implementations. Recent Advances in Parallel Virtual Machine and Message Passing Interface*; Kranzlmüller, D., Volkert, J., Kacsuk, P., Dongarra, J., Eds.; Springer: Berlin/Heidelberg, Germany, 2002; p. 7.
19. Comprés, I.; Mo-Hellenbrand, A.; Gerndt, M.; Bungartz, H.J. Infrastructure and API Extensions for Elastic Execution of MPI Applications. In Proceedings of the 23rd European MPI Users' Group Meeting, Edinburgh, UK, 25–28 September 2016; Association for Computing Machinery: New York, NY, USA, 2016; pp. 82–97.
20. Yoo, A.B.; Jette, M.A.; Grondona, M. SLURM: Simple Linux Utility for Resource Management. In *Job Scheduling Strategies for Parallel Processing*; Feitelson, D., Rudolph, L., Schwiegelshohn, U., Eds.; Springer: Berlin/Heidelberg, Germany, 2003; pp. 44–60.
21. Chadha, M. Adaptive Resource-Aware Batch Scheduling for HPC Systems. Master's Thesis, Technische Universität München, Munich, Germany, 2019.
22. Chadha, M.; John, J.; Gerndt, M. Extending SLURM for Dynamic Resource-Aware Adaptive Batch Scheduling. 2021. Available online: http://xxx.lanl.gov/abs/2009.08289 (accessed on 13 April 2022).
23. Sudarsan, R.; Ribbens, C.J. ReSHAPE: A Framework for Dynamic Resizing and Scheduling of Homogeneous Applications in a Parallel Environment. In Proceedings of the International Conference on Parallel Processing, Xi'an, China, 10–14 September 2007.
24. Tadepalli, S. GEMS: A Fault Tolerant Grid Job Management System. Master's Thesis, Virginia Polytechnic Institute, Blacksburg, VA, USA, 2003.
25. Swaminathan, G. A Scheduling Framework for Dynamically Resizable Parallel Applications. Ph.D. Thesis, Virginia Tech, Blacksburg, VA, USA, 2004.
26. Dongarra, J.J.; Whaley, R.C. *A User's Guide to the BLACS v1.1*; Technical Report 94; University of Tennessee: Knoxville, TN, USA, 1997.
27. Blackford, L.; Choi, J.; Cleary, A. ScaLAPACK: A linear algebra library for message-passing computers. In Proceedings of the SIAM Conference, Stanford, CA, USA, 13–15 July 1997; pp. 1–15.
28. Sudarsan, R.; Ribbens, C. Scheduling Resizable Parallel Applications. In Proceedings of the International Symposium on Parallel & Distributed Processing, Rome, Italy, 23–29 May 2009.
29. Sudarsan, R.; Ribbens, C.J.; Farkas, D. Dynamic Resizing of Parallel Scientific Simulations: A Case Study Using LAMMPS. In Proceedings of the International Conference on Computational Science (ICCS), Baton Rouge, LA, USA, 25–27 May 2009; pp. 175–184.
30. Martín, G.; Singh, D.E.; Marinescu, M.C.; Carretero, J. Enhancing the Performance of Malleable MPI Applications by Using Performance-aware Dynamic Reconfiguration. *Parallel Comput.* **2015**, *46*, 60–77. [CrossRef]
31. Mucci, P.; Moore, S.; Deane, C.; Ho, G. PAPI: A Portable Interface to Hardware Performance Counters. In Proceedings of the Department of Defense HPCMP Users Group Conference, Monterey, CA, USA, 7–10 June 1999; pp. 7–10.
32. Radcliffe, N.; Watson, L.; Sosonkina, M. A Comparison of Alternatives for Communicating with Spawned Processes. In Proceedings of the 49th Annual Southeast Regional Conference, Kennesaw, GA, USA, 24–26 March 2011; Association for Computing Machinery: New York, NY, USA, 2011; pp. 132–137.
33. Martín, G.; Marinescu, M.C.; Singh, D.E.; Carretero, J. FLEX-MPI: An MPI Extension for Supporting Dynamic Load Balancing on Heterogeneous Non-dedicated Systems. In Proceedings of the Euro-Par Parallel Processing, Aachen, Germany, 26–30 August 2013; pp. 138–149.
34. Dongarra, J. Overview of PVM and MPI. 1998. Available online: http://www.netlib.org/utk/people/JackDongarra/pdf/pvm-mpi.pdf (accessed on 3 March 2022).
35. Jajaga, E.; Kllobocishta, J. MPI Parallel Implementation of Jacobi. In Proceedings of the ICT Innovations, Ohrid, Republic of Macedonia, 12–15 September 2012; pp. 449–458. ISSN 1857-7288. Available online: https://www.researchgate.net/publication/255719047_MPI_Parallel_Implementation_of_Jacobi (accessed on 3 March 2022).
36. Martín, G.; Marinescu, M.C.; Singh, D.E.; Carretero, J. Leveraging social networks for understanding the evolution of epidemics. *BMC Syst. Biol.* **2011**, *5*, 1–16. [CrossRef]
37. Carretero, J.; Garcia Blas, J.; Singh, D.; Isaila, F.; Fahringer, T.; Prodan, R.; Bosilca, G.; Lastovetsky, A.; Symeonidou, C.; Pérez-Sánchez, H.; et al. Optimizations to enhance sustainability of MPI applications. In Proceedings of the 21st European MPI Users' Group Meeting, Kyoto, Japan, 9–12 September 2014. [CrossRef]
38. Iserte, S. High-Throughput Computation through Efficient Resource Management. Ph.D. Thesis, Universitat Jaume I, Castelló de la Plana, Spain, 2018. [CrossRef]

39. Bueno, J.; Martinell, L.; Duran, A.; Farreras, M.; Martorell, X.; Badia, R.M.; Ayguade, E.; Labarta, J. Productive Cluster Programming with OmpSs. In *Euro-Par 2011 Parallel Processing: 17th International Conference, Euro-Par 2011, Bordeaux, France, 29 August–2 September 2011, Proceedings, Part I*; Jeannot, E., Namyst, R., Roman, J., Eds.; Springer: Berlin/Heidelberg, Germany, 2011; pp. 555–566. [CrossRef]
40. Iserte, S.; Mayo, R.; Quintana-Ortí, E.S.; Beltran, V.; Peña, A.J. DMR API: Improving cluster productivity by turning applications into malleable. *Parallel Comput.* **2018**, *78*, 54–66. [CrossRef]
41. Sainz, F.; Bellon, J.; Beltran, V.; Labarta, J. Collective Offload for Heterogeneous Clusters. In Proceedings of the 2015 IEEE 22nd International Conference on High Performance Computing (HiPC), Washington, DC, USA, 16–19 December 2015; pp. 376–385.
42. Iserte, S.; Mayo, R.; Quintana-Ortí, E.S.; Beltran, V.; Peña, A.J. Efficient Scalable Computing through Flexible Applications and Adaptive Workloads. In Proceedings of the 46th International Conference on Parallel Processing Workshops (ICPPW), Bristol, UK, 14–17 August 2017; pp. 180–189. [CrossRef]
43. Iserte, S.; Mayo, R.; Quintana-Orti, E.; Pena, A. DMRlib: Easy-coding and Efficient Resource Management for Job Malleability. *IEEE Trans. Comput.* **2020**, *70*, 1443–1457. [CrossRef]
44. Aarseth, S. From NBODY1 to NBODY6: The Growth of an Industry*. *Publ. Astron. Soc. Pac.* **1999**, *111*, 1333–1346. [CrossRef]
45. Iserte, S.; Martínez, H.; Barrachina, S.; Castillo, M.; Mayo, R.; Peña, A.J. Dynamic reconfiguration of noniterative scientific applications. *Int. J. High Perform. Comput. Appl.* **2018**, *33*, 804–816. [CrossRef]
46. Iserte, S.; Rojek, K. An study of the effect of process malleability in the energy efficiency on GPU-based clusters. *J. Supercomput.* **2019**, *76*, 255–274. [CrossRef]
47. Prabhakaran, S.; Iqbal, M.; Rinke, S.; Windisch, C.; Wolf, F. A Batch System with Fair Scheduling for Evolving Applications. In Proceedings of the 2014 43rd International Conference on Parallel Processing, Minneapolis, MN, USA, 9–12 September 2014; pp. 351–360. [CrossRef]
48. Prabhakaran, S.; Neumann, M.; Rinke, S.; Wolf, F.; Gupta, A.; Kale, L.V. A Batch System with Efficient Adaptive Scheduling for Malleable and Evolving Applications. In Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium, Hyderabad, India, 25–29 May 2015; pp. 429–438. [CrossRef]
49. Bhandarkar, M.; Kale, L.V.; de Sturler, E.; Hoeflinger, J. Object-Based Adaptive Load Balancing for MPI Programs. In Proceedings of the International Conference on Computational Science, San Francisco, CA, USA, 28–30 May 2001; LNCS 2074, pp. 108–117.
50. Huang, C.; Lawlor, O.; Kalé, L.V. Adaptive MPI. In Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), College Station, TX, USA, 2–4 October 2003; LNCS 2958, pp. 306–322.
51. Kale, L.; Bhandarkar, M.; Jagathesan, N.; Krishnan, S.; Yelon, J. Converse: An interoperable framework for parallel programming. In Proceedings of the International Conference on Parallel Processing, Bloomingdale, IL, USA, 12–16 August 1996; pp. 212–217. [CrossRef]
52. Bhandarkar, M.; Brunner, R.; Kalé, L. Run-time support for adaptive load balancing. In *Parallel and Distributed Processing–15 IPDPS 2000 Workshops, Proceedings*; Rolim, J., Ed.; Springer: Berlin/Heidelberg, Germany, 2000; pp. 1152–1159. [CrossRef]
53. Antoniu, G.; Bougé, L.; Namyst, R. An efficient and transparent thread migration scheme in the PM2 runtime system. In *Parallel and Distributed Processing*; Rolim, J., Mueller, F., Zomaya, A.Y., Ercal, F., Olariu, S., Ravindran, B., Gustafsson, J., Takada, H., Olsson, R., Kale, L.V., Eds.; Springer: Berlin/Heidelberg, Germany, 1999; pp. 496–510.
54. Teich, J.; Henkel, J.; Herkersdorf, A.; Schmitt-Landsiedel, D.; Schröder-Preikschat, W.; Snelting, G., Invasive Computing: An Overview. In *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*; Hübner, M., Becker, J., Eds.; Springer: New York, NY, USA, 2011; pp. 241–268. [CrossRef]
55. Pujari, R.K.; Wild, T.; Herkersdorf, A.; Vogel, B.; Henkel, J. Hardware assisted thread assignment for RISC based MPSoCs in invasive computing. In Proceedings of the 2011 International Symposium on Integrated Circuits, Singapore, 12–14 December 2011; pp. 106–109. [CrossRef]
56. Oechslein, B.; Schedel, J.; Kleinöder, J.; Bauer, L.; Henkel, J.; Lohmann, D.; Schröder-Preikschat, W. OctoPOS: A Parallel Operating System for Invasive Computing; In Proceedings of the EuroSys 2011 Workshop on Systems for Future Multi-Core Architectures (SFMA), Salzburg, Austria, 10 April 2011.
57. Saraswat, V.; Bloom, B.; Peshansky, I.; Tardieu, O.; Grove, D. X10 Language Specification, Version 2.3. 2012. Available online: http://x10.sourceforge.net/documentation/languagespec/x10-230.pdf (accessed on 3 March 2022).
58. Calin, G.; Derevenetc, E.; Majumdar, R.; Meyer, R. A Theory of Partitioned Global Address Spaces. *arXiv* **2013**. arXiv:1307.6590
59. Saraswat, V.; Almasi, G.; Bikshandi, G.; Cascaval, C.; Cunningham, D.; Grove, D.; Kodali, S.; Peshansky, I.; Tardieu, O. The asynchronous partitioned global address space model. In Proceedings of the First Workshop on Advances in Message Passing. Toronto, ON, Canada, 6 June 2010; pp. 1–8.
60. Bungartz, H.J.; Riesinger, C.; Schreiber, M.; Snelting, G.; Zwinkau, A. Invasive computing in HPC with X10. In Proceedings of the third ACM SIGPLAN X10 Workshop, Seattle, WA, USA, 16–19 June 2013; ACM: New York, NY, USA, 2013; pp. 12–19. [CrossRef]
61. Zwinkau, A.; Buchwald, S.; Snelting, G. *Invade X10 Documentation v0.5*; Technical Report; Karlsruhe Institute of Technology: Karlsruhe, Germany, 2013.
62. Buchwald, S.; Mohr, M.; Zwinkau, A. Malleable invasive applications. In *Software Engineering (Workshops)*; Technische Universität Ilmenau: Ilmenau, Germany, 2015; Volume 1337, pp. 123–126.

63. Ribeiro, F.S.; Nascimento, A.P.; Boeres, C.; Rebello, V.E.F.; Sena, A.C. Autonomic Malleability in Iterative MPI Applications. In Proceedings of the Symposium on Computer Architecture and High Performance Computing, Washington, DC, USA, 23–26 October 2013; pp. 192–199.
64. Boeres, C.; Rebello, V. EasyGrid: Towards a framework for the automatic Grid enabling of legacy MPI applications. *Concurr. Pract. Exp.* **2004**, *16*, 425–432. [CrossRef]
65. Nascimento, A.P.; Sena, A.C.; Silva, J.A.; Vianna, D.Q.C.; Boeres, C.; Rebelló, V.E.F. Autonomic application management for large scale MPI programs. *Int. J. High Perform. Comput. Netw.* **2008**, *5*, 227–240. [CrossRef]
66. Houzeaux, G.; Badia, R.M.; Borrell, R.; Dosimont, D.; Ejarque, J.; Garcia-Gasulla, M.; López, V. *Dynamic Resource Allocation for Efficient Parallel CFD Simulations*; Technical Report; Barcelona Supercomputing Center: Barcelona, Spain, 2021.
67. Vázquez, M.; Houzeaux, G.; Koric, S.; Artigues, A.; Aguado-Sierra, J.; Arís, R.; Mira, D.; Calmet, H.; Cucchietti, F.; Owen, H.; et al. Alya: Multiphysics engineering simulation toward exascale. *J. Comput. Sci.* **2016**, *14*, 15–27. [CrossRef]
68. Badia, R.M.; Conejero, J.; Diaz, C.; Ejarque, J.; Lezzi, D.; Lordan, F.; Ramon-Cortes, C.; Sirvent, R. COMP Superscalar, an interoperable programming framework. *SoftwareX* **2015**, *3–4*, 32–36. [CrossRef]
69. Tejedor, E.; Badia, R.M. COMP Superscalar: Bringing GRID superscalar and GCM together. In Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID), Lyon, France, 19–22 May 2008. [CrossRef]
70. Lopez, V.; Ramirez Miranda, G.; Garcia-Gasulla, M. 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, Stockholm, Sweden, 25 June 2021; Association for Computing Machinery: New York, NY, USA, 2021; pp. 3–10. [CrossRef]
71. Lordan, F.; Tejedor, E.; Ejarque, J.; Rafanell, R.; Álvarez, J.; Marozzo, F.; Lezzi, D.; Sirvent, R.; Talia, D.; Badia, R.M.; et al. ServiceSs: An Interoperable Programming Framework for the Cloud. *J. Grid. Comput.* **2014**, *12*, 67–91. [CrossRef]
72. Tejedor, E.; Becerra, Y.; Alomar, G.; Queralt, A.; Badia, R.M.; Torres, J.; Cortes, T.; Labarta, J. PyCOMPSs: Parallel computational workflows in Python. *Int. J. High Perform. Comput. Appl.* **2017**, *31*, 66–82. [CrossRef]
73. Cao, Q.; Bosilca, G.; Losada, N.; Wu, W.; Zhong, D.; Dongarra, J. Evaluating Data Redistribution in PaRSEC. *IEEE Trans. Parallel Distrib. Syst.* **2022**, *33*, 1856–1872. [CrossRef]
74. Garcia, M.; Labarta, J.; Corbalan, J. Hints to Improve Automatic Load Balancing with LeWI for Hybrid Applications. *J. Parallel Distrib. Comput.* **2014**, *74*, 2781–2794. [CrossRef]
75. Martín-Álvarez, I.; Aliaga, J.I.; Castillo, M.I.; Mayo, R.; Iserte, S. Malleability Implementation in a MPI Iterative Method. In Proceedings of the 2021 IEEE International Conference on Cluster Computing (CLUSTER), Portland, OR, USA, 7–10 September 2021; pp. 801–802. [CrossRef]
76. Zheng, G.; Xiang Ni.; Kale, L.V. A scalable double in-memory checkpoint and restart scheme towards exascale. In Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012), Boston, MA, USA, 25–28 June 2012; pp. 1–6. [CrossRef]
77. Garcia, M.; Corbalan, J.; Labarta, J. LeWI: A Runtime Balancing Algorithm for Nested Parallelism. In Proceedings of the 2009 International Conference on Parallel Processing, Piscataway, NJ, USA, 22–25 September 2009; pp. 526–533. [CrossRef]