Efficient Scalable Computing through Flexible Applications and Adaptive Workloads

Sergio Iserte, Rafael Mayo, Enrique S. Quintana-Ortí
Universitat Jaume I (UJI)
Castelló de la Plana, Spain
{siserte,mayo,quintana}@uji.es

Vicenç Beltran, Antonio J. Peña
Barcelona Supercomputing Center (BSC)
Barcelona, Spain
{vbeltran,antonio.pena}@bsc.es

***Abstract*—In this paper we introduce a methodology for dynamic job reconfiguration driven by the programming model runtime in collaboration with the global resource manager. We improve the system throughput by exploiting malleability techniques (in terms of number of MPI ranks) through the reallocation of resources assigned to a job during its execution. In our proposal, the OmpSs runtime reconfigures the number of MPI ranks during the execution of an application in cooperation with the Slurm workload manager. In addition, we take advantage of OmpSs offload semantics to allow application developers deal with data redistribution. By combining these elements a job is able to expand itself in order to exploit idle nodes or be shrunk if other queued jobs could be initiated. This novel approach adapts the system workload in order to increase the throughput as well as make a smarter use of the underlying resources. Our experiments demonstrate that this approach can reduce the total execution time of a practical workload by more than 40% while reducing the amount of resources by 30%.**

***Keywords*-Job Reconfiguration; Dynamic Reallocation; Smart Resource Utilization; Scale-Malleable workload**

I. Introduction

In real HPC facilities, applications run on shared computers where hundreds or thousands of other applications are competing for the same resources. In this scenario, applications are submitted to the system with the shape of parallel jobs that conform the current workload of a system. Adapting the workload to the infrastructure can render considerable improvements in resource utilization and global throughput. A potential approach to obtain the desired adaptivity consists in applying *dynamic job reconfiguration*, which devises resource usage to be potentially changed at execution time.

The adaption of the workload to the target infrastructure brings benefits to both system administrators and end users. While administrators would like to see the throughput rate increased and a smarter resource utilization by the applications, end-users are the direct beneficiaries of the scale-adaptivity as they will not experience strict resource requirements on submission. Although this may prevent the application from being executed in the shortest time, a faster completion time (waiting plus execution time) will compensate for this.

In order to dinamically adapt a workload to the infrastructure we need two main tools: (i) a resource manager system (RMS) capable of modifying the resources assigned to a job; and (ii) a parallel runtime to rescale an application. In our solution, we have connected these components by developing a communication layer between the RMS and the runtime.

In this work we enhance the Slurm Workload Manager [1] to achieve fair dynamic resource assignment while maximizing the cluster throughput. We select Slurm because it is one of the most widely-adopted RMSs worldwide, it is open-source, portable, and highly scalable [2], [3].

To exploit a collection of distributed resources, the vast majority of the scientific applications that run on high performance clusters use the Message Passing Interface (MPI) [4], either directly or on top of programming models or libraries leveraging MPI underneath. *Reconfiguration* is possible in MPI applications via the MPI spawning functionality.

The direct use of MPI to handle migrations, however, requires considerable effort from skilled software developers in order to manage the whole set of data transfers among processes in different communicators. For this purpose, we benefit from the recently-incorporated offload semantics of the OmpSs programming model [5] to ease the malleability process and data redistribution. In addition, we adapt the Nanos++ OmpSs runtime to interact with Slurm. We improve the `Nanos++` runtime to reconfigure MPI jobs and establish direct communication with the RMS. For that, applications will expose "reconfiguring points" where, signaled by the RMS, the runtime will assist to resize the job on-the-fly. We highlight that, although we benefit from the OmpSs infrastructure and semantics for job reconfiguration, our proposal may well be leveraged as a specific-purpose library, and applications using this solution are free to implement on-node parallelism using other programming models such as OpenMP or OmpSs.

Our interest in developing an Application Programming Interface (API) for *Dynamic Management of Resources* (DMR API) is rooted on the lack of performance of Checkpoint/Restart (C/R) mechanisms when applied to dynamic reconfiguration. Figure 1 illustrates our point via a comparison of the time needed for the non-solving stages of the N-body simulation (see Section VII-B4) using the C/R approach versus the DMR API. In particular, the labels of the "spawing" bars reveal an important increment in the cost of spawning processes for C/R with respect to the DMR API (e.g., for 48–24 processes by a factor 63.75×), because of the need to save data to disk to be later reloaded.

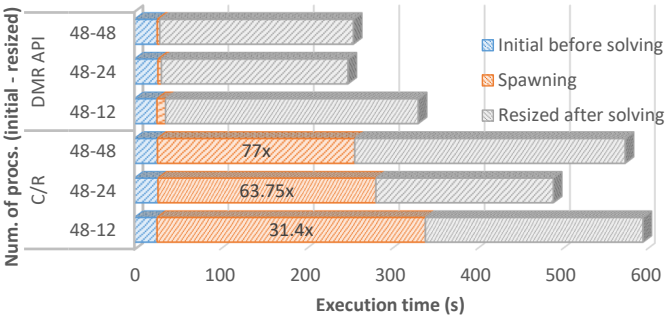


Fig. 1. Execution time of non-solving stages of N-body simulation.

In summary, the main contribution of this paper is a mechanism to accomplish MPI malleability, based on existing components (MPI, OmpSs, and Slurm) that enhances resource usage in order to produce higher global throughput in terms of executed jobs per unit of time. To that extent, we propose (1) an extension of the OmpSs offload mechanism to deal with dynamic reconfiguration; (2) a reconfiguration policy for the RMS to decide whether a job must be expanded or shrunk; and (3) a communication protocol for the runtime to interact with the RMS, based on application-level API calls. Finally, (4) we also provide an extensive evaluation of the framework that demonstrates the benefits of our workload-aware approach.

The rest of this paper is structured as follows: Section II discusses related work. Section III presents an overview of the proposed methodology. Sections IV and V present the extensions developed in the Slurm RMS and the Nanos++ runtime in order to support our programming model proposal discussed in Section VI. The analysis and experimentation involve these three sections: Section VII describes the platform and applications; Section VIII evaluates all the features implemented; and Section IX presents and analyses a realistic use case for our framework, demonstrating the benefits of deploying adaptive workloads in a production cluster. Section X outlines the conclusions and discusses future work.

II. Related Work

In general, a job (application) can be classified in one of the following types: *rigid*, *moldable*, *malleable* and *evolving* [6]. These classes depend in the number of concurrent processes during the execution of a job, so that we collapse them into two categories:

- Fixed: the number of parallel processes remains constant during the execution (rigid and moldable applications).
- Flexible: the number of processes can be reconfigured on-the-fly, allowing distinct numbers of parallel processes in different parts of the execution (malleable and evolving applications) or job malleability. This action is known as *dynamic reconfiguration*.

The first steps toward malleability were in *shared-memory* systems exploiting the flexibility of applications. In [7] the authors leveraged moldability together with preemptive policies, such as equi-partitioning and folding. These policies can interrupt active jobs in order to redistribute processors among the pending jobs.

Checkpointing mechanisms have been used in the past to save the application state and resume its execution with a different number of processes, or simply to migrate the execution to other processes. The work in [8] explores how malleability can be used in checkpoint/restart applications. There, a checkpoint–and–reconfigure mechanism is leveraged to restart applications with a different number of processes from data stored in checkpoint files. Storing and loading checkpoint files, however, poses a nonnegligible overhead versus runtime data redistribution.

In [9], the authors combine Scalable Checkpoint/Restart for MPI (SCR) [10] with the User Level Failure Migration (ULFM) MPI standard proposal for fault-tolerance [11]. A resizing mechanism based on CHARM++ is presented in [12]. The authors of that work demonstrate the benefits of resizing a job in terms of both performance and throughput, but they do not address the data redistribution problem during the resizing.

The authors of [13] rely on an MPI implementation called EasyGrid AMS in order to adjust automatically the size of a job. Another similar approach is found in [14], where a performance-aware framework based on the Flex-MPI library [15] is presented. That work leverages job reconfiguration in order to expand/shrink a job targeting execution performance. For that purpose, the framework monitors execution, predicts future performance, and balances the load.

In the literature we can also find several works that combine malleability with resource management. ReSHAPE [16] integrates job reconfiguration techniques with job scheduling in a framework that also considers the current performance of the execution. Complementary research using this framework analyzes its impact on individual performance and throughput in small workloads [17], [18]. That solution, however, requires all applications in the cluster to be specifically-developed to be flexible under the ReSHAPE framework. In a more recent work, they present a more in-depth study discussing the ReSHAPE behavior with a workload of 120 jobs [19].

An additional important contribution is [20], where a batch system with adaptive scheduling is presented. The authors in this paper enable the communication between the RMS Torque/Maui and Charm++ as a parallel runtime. Charm++ applications are presented as automatically malleable thanks to checkpointing.

Compared with previous work, we present a holistic throughput-oriented reconfiguration mechanism based on existing software components that is compatible with unmodified non-malleable applications. Our experiments are run on a platform with 65 nodes, for large workloads with up to 400 jobs. The numbers exceed the 50- node platforms and 230-job workloads in the current state- of-the-art. Furthermore, in contrast with previous studies, we configure our workloads not only leveraging synthetic applications.

III. Methodology Overview

Slurm exposes an API that may be used by external software agents. We use this API from the Nanos++ OmpSs runtime in

order to design the job resize mechanism. Thus, Slurm's API allows us to resize a job following the next steps:

- Job A has to be expanded
  1) Submit a new job B with a dependency on the initial job A. Job B requests the number of nodes NB to be added to job A.
  2) Update job B, setting its number of nodes to 0. This produces a set of NB allocated nodes which are not attached to any job.
  3) Cancel job B.
  4) Update job A and set its number of nodes to NA+NB.
- Job A has to be shrunk
  1) Update job A, setting the new number of nodes to the final size (NA is updated).

After these steps, Slurm's *environment variables* for job A are updated. These commands have no effect on the status of the running job, and the user remains responsible for any malleability process and data redistribution.

The framework we leverage consists of two main components: the RMS and the programming model runtime. The RMS is aware of the resource utilization and the queue of pending jobs. When an application is in execution, it periodically contacts the RMS, through the runtime, communicating its rescaling willingness (to expand or shrink the current number of allocated nodes). The RMS inspects the global status of the system to decide whether to initiate any rescaling action, and communicates this decision to the runtime. If the framework determines that a rescale action is due, the RMS, the runtime, and the application will collaborate to continue the execution of the application scaled to a different number of nodes (MPI processes).

IV. Slurm Reconfiguration Policy

We designed and developed a resource selection plug-in responsible for reconfiguration decisions. This plug-in realizes a node selection policy featuring three modes that accommodate three degrees of scheduling freedom:

1) Request an Action: Applications are allowed to "strongly suggest" a specific action. For instance, to expand the job, the user could set the "minimum" number of requested nodes to a value that is greater than the number of allocated nodes. However, Slurm will ultimately be responsible for granting the operation according to the overall system status.

2) Preferred Number of Nodes: One of the parameters that applications can convey to the RMS is their *desired* number of nodes to execute a specific computational stage. If the desired size corresponds to the current size, the RMS will return "no action". If a "preference" is requested, and there is no outstanding job in the queue, the expansion can be granted up to a specified "maximum" (line 2 of Algorithm 1). Otherwise, if the desired value is different from the current allocation, (lines 6 and 10 of Algorithm 1) the RMS will try to expand or shrink the job to the preferred number of nodes.

3) Wide Optimization: The cases not covered by the preceding methods (Algorithm 1, line 13) are handled as follows:

A job is expanded if there are sufficient available resources to fulfill the new requirement of nodes and either (1) there is no job pending for execution in the queue, or (2) no pending job can be executed due to insufficient available resources. By expanding the job, we can expect it to finish its execution earlier and release the associated resources.

A job is shrunk if there is any queued job that could be executed by taking this action. More jobs in execution should increase the global throughput. Moreover, if the job is going to be shrunk, the queued job that has triggered the shrinking event will be assigned the maximum priority in order to foster its execution.

Algorithm 1 Slurm Reconfiguration Policy

```
1:  if preferred then
2:      if am I the only job in the queue? then
3:          action ← expand.
4:          processes ← jobMaxProcs.
5:      else
6:          if can I expand to preferred? then
7:              action ← expand.
8:              processes ← max_procs_to(preferred).
9:          else
10:             if can I shrink to preferred? then
11:                 action ← shrink.
12:                 processes ← preferred.
13: else
14:     if are there pending jobs in the queue? then
15:         if can other job run with my resources? then
16:             action ← shrink.
17:             processes ← min_procs_run(targetJobId).
18:             set_max_priority(targetJobId).
19:         else
20:             action ← expand.
21:             processes ← max_procs_to(jobMaxProcs).
22:     else
23:         action ← expand.
24:         processes ← jobMaxProcs.
```

V. Nanos++ Runtime Extensions

We implemented the necessary logic in Nanos++ to reconfigure jobs in tight cooperation with the RMS. In this section we discuss the extended API and the resizing mechanisms.

A. The Dynamic Management of Resources (DMR) API

We designed the DMR API with two main functions: `dmr_check_status` and its asynchronous version `dmr_icheck_status`. These routines instruct the runtime (Nanos++) to communicate with the RMS (Slurm) in order to determine the resizing action to perform: "expand", "shrink", or "no action". The asynchronous counterpart schedules the next action for the next execution step, at the same time that

the current step is executed. Hence, by skipping the action scheduling stage, the communication overhead in that step is avoided. Details of the API overhead have to be avoided due to lack of space.

Thus, in case an action is to be performed, these functions spawn the new set of processes and return an opaque handler. This API is exposed by the runtime and it is intended to be used by applications. These functions present the following input arguments.

- Minimum number of processes to be resized to.
- Maximum number of processes. This prevents the application from growing beyond its scalability capabilities.
- Resizing factor (E.g.: a factor of 2 will expand/shink the number of processes to a value multiple/divisor of 2).
- Preferred number of processes.

The output arguments return the new number of nodes and an opaque handler to be used in the task offloading directives.

An additional mechanism implemented to reach a fair balance between performance and throughput is the "checking inhibitor". This introduces a timeout during which the calls to the DMR API are ignored. This knob is mainly intended to be leveraged in iterative applications with short iteration intervals. The inhibition period can be tuned by means of an environment variable (`NANOX_SCHED_PERIOD`).

B. Automatic Job Reconfiguration

The runtime will perform the following actions in order to leverage the Slurm resizing mechanisms (see Section IV) by means of its external API:

1) Expand: A new *resizer* job (RJ) is first submitted requesting the difference between the current and total amount of desired nodes. This enables the original nodes to be reused. There is a dependency relation between the RJ and the original job (OJ). In order to follow better the RMS decisions, RJ is set to the maximum priority, facilitating its execution.

The runtime waits until JR moves from "pending" to "running" status. If the waiting time reaches a threshold, RJ is canceled and the action is aborted. This situation may occur if the RMS assigns the available resources to a different job during the scheduling action. This is more likely to occur in the asynchronous mode because an action then can experience some delay during which the status of the queue may change. Once OJ is reallocated, the updated list of nodes is gathered and used in a call to `MPI_Comm_spawn` in order to create a new set of processes.

2) Shrink: The shrinking mechanism is slightly more complex than its expansion counterpart because Slurm will have to kill all processes executing in the released nodes. To prevent premature process termination, we need a synchronized workflow to guide the job shrinking. Hence, the RMS sets a *management node* in charge of receiving an acknowledgment from all other processes. These ACKs will signal that they finished their offloading tasks and the node is ready to be released.

After a scheduling is complete, the DMR call returns the expand-shrink action to be performed and the resulting number of nodes. The application is responsible for stating the appropriate data dependencies and triggering the tasks offloading to the new set of processes (see Section VI next).

```
1  void main(int argc, char **argv) {
2    ...
3    int t = 0;
4    MPI_Comm_get_parent(&parentComm);
5    if (parentComm == MPI_COMM_NULL) {
6      init(data);
7    } else {
8      MPI_Recv(parentComm, data, myRank);
9      MPI_Recv(parentComm, &t, myRank);
10   }
11   compute(data, t);
12   ...
13 }
14
15 void compute(data, t0) {
16   for (t=t0; t<timesteps; t++) {
17     nodeList = get_new_nodelist_somehow();
18     if (nodelist != NULL) {
19       MPI_Comm_spawn(myapp.bin, nodeList, &newComm);
20       MPI_Send(newComm, data, myRank);
21       MPI_Send(newComm, t, myRank);
22       exit(0);
23     }
24     compute_iter(data, t);
25   }
26 }
```

Listing 1. Pseudo-code of job reconfiguration using bare MPI.

VI. Programming Model

In this section we review our programming model approach to address dynamic reconfiguration coordinated by the RMS. The programmability of our solution benefits from relying the OmpSs offload semantics versus directly using MPI. Next, we illustrate this via a practical example.

A. Benefits of the OmpSs Offload Semantics

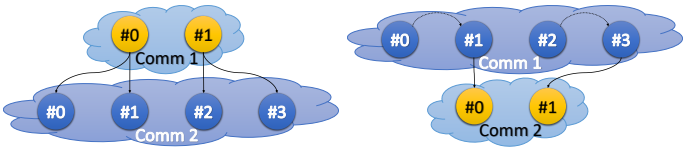
To showcase the benefits of the OmpSs offload semantics, we review the specific case of migration. This analysis allows us to focus on the fundamental differences between programming models because it does not involve data redistribution among a different number of nodes (which is of similar complexity in both models). We show a complete example in Section VI-B.

a) MPI Migration: Listing 1 contains an excerpt of pseudo-code with a program which directly uses MPI calls. In this case, we assume some mechanism is available to determine the new node list in line 17.

b) OmpSs-based Migration: The same functionality is attained in Listing 2 by leveraging our proposal on top of the OmpSs offload semantics. This includes a call to our extended API in line 11. At a glance, our proposal exposes higher-level semantics, increasing code expressiveness and programming productivity. In addition, communication with the RMS is implicitly established in the call to the runtime in line 11, which pursues a potential increase in overall system resource utilization. Data transfers are managed by the runtime with the directive in line 13. Last, the "taskwait" (line 15) follows the original semantics of the clause and with it, the

```
1  void main(void) {
2    ...
3    int t = 0;
4    init(data);
5    compute(data, t);
6    ...
7  }
8
9  void compute(data, t0) {
10   for (t=t0; t<timesteps; t++) {
11     action = dmr_check_status(&newNnodes, &handler);
12     if (action) {
13       #pragma omp task inout(data) onto(newComm,
             myRank)
14       compute(data, t)
15       #pragma omp taskwait
16     } else
17       compute_iter(data);
18   }
19 }
```

Listing 2. Pseudo-code of job reconfiguration using OmpSs.



(a) Expand. (b) Shrink.

Fig. 2. Data transfers.

initial processes terminate, letting the execution continue in the processes of the new communicator.

B. A Practical Example

The excerpt in Listing 3 is derived from that showcased in Section VI-A to discuss malleability. In this case the application must drive the task redistribution according to the resizing action. The mapping `factor` indicates the number of processes in the current set that are mapped to the processes in the new configuration (see Figure 2). This example implements homogeneous distributions, where we always resize to a multiple or divisor of the current number of processes. Our model, however, supports arbitrary distributions.

For the "expand" action (line 8), the original processes must partition the dataset. For instance, in Figure 2a, the processes split the dataset into two subsets, mapping each half to a process in the new configuration. The data transfers are performed by the runtime according to the information included in the task offloading directive (line 12).

The "shrink" action, on the other hand, involves preliminary explicit data movement. The processes in the original set are grouped into "senders" and "receivers". This initial data movement is illustrated in the example in Figure 2b.

VII. Experimental Setup

A. Platform

Our evaluation was performed on the Marenostrum Supercomputer at *Barcelona Supercomputing Center*. Each compute node in this facility is equipped with two 8-core Intel Xeon E5-2670 processors running at 2.6 GHz with 128 GB of RAM.

```
1  void compute(data, t0) {
2    for (t=t0; t<timesteps; t++) {
3      action = dmr_check_status(&newNnodes, &handler);
4      if (!action)
5        compute_iter(data);
6      else {
7        if (action == "expand") {
8          factor = newNnodes / worldRanks;
9          for (i=0; i<factor; i++) {
10           dest = myRank * factor + i;
11           subdata = part_data(factor, data);
12   #pragma omp task inout(subdata) onto(handler,dest)
13           compute(subdata, t);
14           #pragma omp taskwait
15         }  // End for
16       } else if (action == "shrink") {
17         factor = worldRanks / newNnodes;
18         sender = (myRank % factor) < (factor - 1);
19         if (sender) {
20           dst = factor * (myRank / factor + 1) - 1;
21           MPI_Isend(comm, data, dst);
22         } else {  // Receiver
23           for (i=1; i<=factor; i++) {
24             src = myRank - factor + i;
25             MPI_Irecv(comm, &alldata, src);
26           }  // End for
27         }  // End if (sender)
28         MPI_Waitall();
29         if (!sender) {
30           dest = myRank / factor;
31   #pragma omp task inout(alldata) onto(handler,dest)
32           compute(alldata, t);
33           #pragma omp taskwait
34         }  // End if (!sender)
35       }  // End if (action == ...)
36     }  // End if (action)
37   }  // End for
38 }  // End compute()
```

Listing 3. Pseudo-code of a malleable application.

The nodes are connected via an InfiniBand Mellanox FDR10 network. For the software stack we used MPICH 3.2, OmpSs 15.06, and Slurm 15.08.

Slurm was configured with the *backfill* job scheduling policy. Furthermore, we also enabled job priorities with the policy *multifactor*. Both were configured with default values.

B. Flexible Applications

For our experimentation we used one synthetic and three real applications, described below:

1) Flexible Sleep (FS): This iterative synthetic application performs a *sleep* in each step (iteration). The time of the step depends on the number of processes deployed in that iteration, so if the job is resized, the sleep time is modified assuming perfect linear scalability. Apart from the *sleep* that simulates the computation time, the application also manages an array of doubles, distributed among the ranks. This array conforms the data-dependency for OmpSs, redistributed in each reconfiguration.

2) Conjugate Gradient (CG): The CG method[1] is an iterative algorithm for the numerical solution of sparse systems of linear equations that produces a solution after a finite number of steps. For our experimentation the method will perform a specific number of iterations in order to have control of the execution time.

[1]https://en.wikipedia.org/wiki/Conjugate_gradient_method

The data of the application propagated in each iteration step of CG is constrained to a matrix flat-stored and four vectors. This version is implemented using OpenMP+MPI, and each MPI process works on a block of rows of the matrix and the corresponding elements from the vectors. The local matrix-vector products are parallelized with OpenMP.

We have applied our OmpSs-based extensions in order to ease the creation of new processes while maintaining the data dependencies after the resizing procedure. The five data structures in CG conform the data-dependencies between iterations in the OmpSs programming model, and they are redistributed when a rescaling is necessary.

During a resize, the data in the matrix and vectors must be redistributed according to the new number of MPI processes.

3) Jacobi: The Jacobi method[2] is an iterative and embarrassingly-parallel algorithm for the solution of a system of linear equations.

Our OpenMP+MPI version of this solver is based on the implementation presented in [21]. The program layout is similar to the CG implementation. In this application, we also have a flat matrix, but only two vectors. These three structures conform the data-dependencies for OmpSs and they are all distributed among the processes.

4) N-body: The N-body problem[3] simulates the individual motions of a group of objects interacting with each other by means of a given force.

We have used an OmpSs+MPI version of this simulator where each process stores a subset of particles, while the intra-node parallelism is exploited by OmpSs.

The amount of work of N-body per iteration is considerably larger than that present in the remaining two real applications described in this section. Apart from computing the position and forces of its own particles, each process exchanges its local subset of particles with the other processes. At the end of the iteration, all the processes have worked with the whole set of particles.

The data-dependency in this particular case is dictated by an array of particles with information about position, velocity, mass and weight. This array is split or merged when an scale-up or down is respectively scheduled.

C. Workload Configuration

The workloads were generated using the statistical model proposed by Feitelson [6], which characterizes rigid jobs based on observations from logs of actual cluster workloads. These include the distribution of job sizes in terms of number of processors, the correlation of runtime with parallelism, and the number of repeated runs. Among others, we found four customizable parameters to be especially relevant:

- `Jobs`: Number of jobs to be launched.
- `Job size`: Number of nodes determined by a complex discrete distribution.
- `Runtime`: Fixed following a hyperexponential distribution based on the `job size`.

[2]https://en.wikipedia.org/wiki/Jacobi_method
[3]https://en.wikipedia.org/wiki/N-body_problem

TABLE I
CONFIGURATION PARAMETERS FOR THE APPLICATIONS

Application	Iterations	Number of Processes			Scheduling period
		Minimum	Maximum	Preferred	
FS	25	1	20	-	-
CG	10000	2	32	8	15 seconds
Jacobi	10000	2	32	8	15 seconds
N-body	25	1	16	1	-

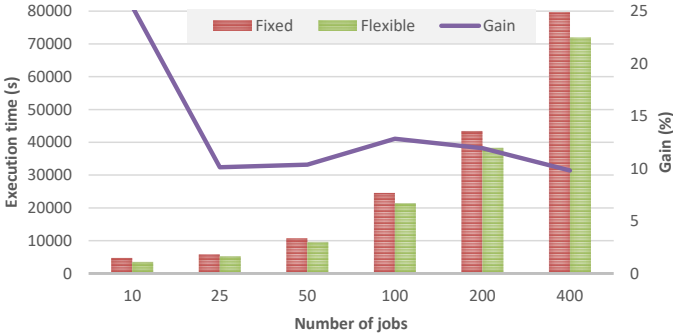


Fig. 3. Comparison between fixed and flexible workloads with different number of jobs.

- `Arrival`: Inter-arrival times of jobs modeled using a Poisson distribution.

For every job the `shrink-expand factor` was set to 2.

VIII. PRELIMINARY STUDY

In this section we present an in-depth analysis of the framework's features using synthetic flexible jobs. For this purpose, we used the FS application configured to perform 2 steps and to transfer 1 GB of data during the reconfiguration.

In order to test thoroughly the features of our solution, we performed 4 different experiments comprising: synchronous and asynchronous scheduling, heterogeneous workload, and micro-steps.

A. Testbed

For this study we only used the FS application; we generated several workloads of different size (number of `jobs`), assigning up to 20 nodes to each job (the number of available nodes in this experiment) with the parameter `job size`; the maximum `runtime` was set to 60 seconds for each step; and the average `arrival` time was 10 seconds. Table I details the rest of the FS parameters for the reconfigurations. The array was determined to have 1 GB of data transferred in each iteration. Furthermore, by not giving a `preferred` value the RMS has more freedom to reallocate resources.

B. Synchronous Reconfiguration Scheduling

For the first test, we launched workloads of different sizes in terms of number of jobs. Figure 3 depicts the execution time for this experiment. Each workload has both a fixed as well as a flexible version. The line "Gain" in that chart indicates the reduction of the execution time (in %) attained by the flexible workload with respect to the fixed workload.

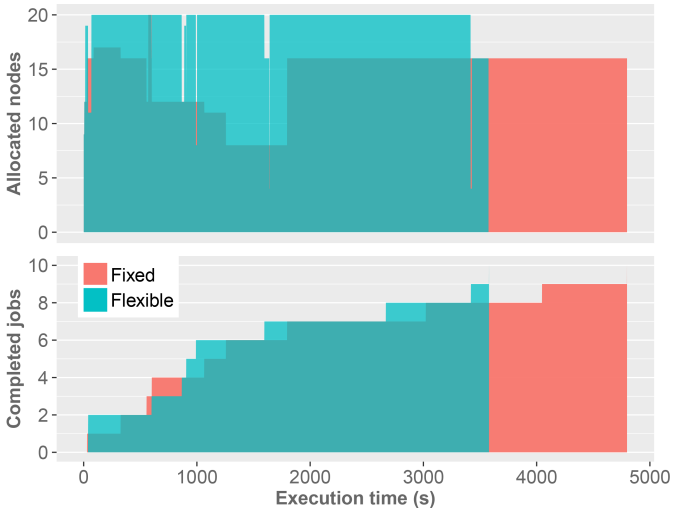


Fig. 4. Evolution in time for the 10-job workload.

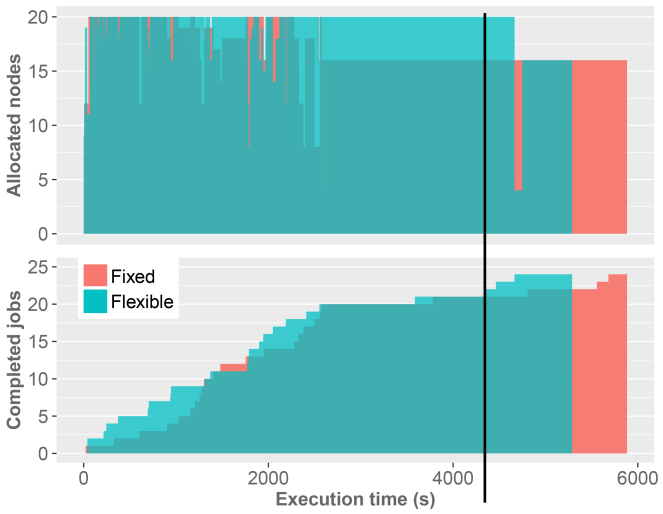


Fig. 5. Evolution in time for the 25-job workload.

Except for the 10-job workload, we can appreciate a gain in the interval 10-15% for the execution time, though the benefit decreases as the workload grows. Nonetheless, this occurs because we are evaluating a finite workload. Under these conditions, the scheduler is able to backfill jobs and fulfill resources, but malleability cannot bring a higher resource utilization. In a more realistic scenario, involving a much larger workload, the throughput is always higher for the flexible workloads. For instance, bottom plots in Figures 4 and 5 show that the productivity attained by the flexible workload is always higher than that offered by the fixed workload.

Figure 4 reports an almost-full allocation of resources during the flexible execution, exposing that the remarkable gain is due to the augment in resource usage.

In contrast, Figure 5 depicts the behavior of a workload of 25 jobs, for which we observe a narrower gain. The problem here arises on the last job (LJ) just in the vertical black line in that figure. Specifically, in that instant LJ is using 4 nodes

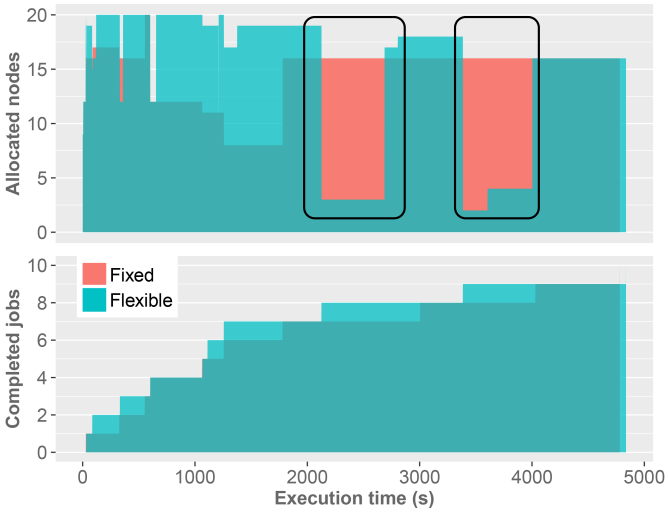


Fig. 6. Asynchronous scheduling of the 10-job workload.

Fig. 7. Comparison between fixed and flexible workloads using asynchronous selection of action.

while the penultimate job (PJ) before completion, owns the rest of resources. When PJ finishes, 16 nodes are released, but until the next check, LJ cannot be expanded. At that point, the scheduler decides to expand the job to its maximum, in this experiment 16 nodes. At the end of the timeline, no more jobs can use the spare resources. This is the same situation that appears in the fixed workload. The consequence is that there is no further improvement, as the flexible policy had already obtained the gain from the first reallocation of resources.

C. Asynchronous Reconfiguration Scheduling

In this test we evaluated the asynchronous version of the scheduling. Again, we compare a fix workload and its flexible counterpart, but now the decision is made asynchronously. Here, we remind that the asynchronous scheduling takes a decision in a specific step but the action takes place in the next step. In the meantime, the status of the system may change. Therefore, the conditions found when the action is applied in the next step might not be the same that were present when the RMS decided the future action. In this situation, enforcing outdated actions may result in an inefficient use of resources.

Let us analyze the effect of adopting outdated decisions for the asynchronous scheduling in the 10-job workload. In this

Fig. 8. Execution times of workloads of 100 jobs with different rates of flexible (Showing the top of the chart, Y axis is not starting in 0).

Fig. 9. Execution time for the different inhibition periods (bars) and the gain respect the fixed workload (percentage on the right of the bars).

example the flexible workload performs worse than the fixed execution. We can explain the cause analyzing Figure 6. If we focus on the resource allocation evolution (top of Figure 6), two relevant gaps can be identified between seconds 2000-3000 and 3000-4000. At the beginning of the first gap in Figure 6, there are 3 jobs in execution that occupy a total of 19 nodes: one (J1) with 16, one (J2) with 2, and a last one (J3) with 1. Moreover, the RMS has decided to expand J3 to 2 nodes in the next step. Now, J1 is finished, leaving the other 2 active jobs using 3 nodes (J3 has not expanded yet). J2 is resolved to expand and acquires 16 nodes (around second 2500). J3 experiences a longer step, carrying out its pending action of expanding to 2, yielding an overall of 18 allocated nodes. If J3 had checked the resources at that moment, it would have been expanded to 4 nodes, but the asynchronous scheduling was negotiated earlier, when the conditions were different. In addition to realizing the expansion, the scheduler decides that J3 will expand to 4 nodes in the next step.

The beginning of the second gap indicates the completion of J2. A few seconds later J3 expands to 4 nodes (instead of doing it to 16 if the scheduling had been synchronous).

Despite the lack of good results for small workloads, the larger workload completion times reveal a higher gain. In that type of situation the malleability overcomes the initial problem described above.

As we did in the synchronous benchmark (Figure 3), if we dismiss the small executions (10-to-50-job workloads), we can observe around a 6%-gain, with the improvement decreasing as we add more jobs to the workload.

This test reveals that so far, there is no need of using an asynchronous scheduling. Hence, the rest of the experiments will exclusively use the synchronous mode.

D. Heterogeneous Workloads

In this benchmark we mixed flexible and fixed jobs in the same workload in order to study their interaction. The workloads were composed of 100 jobs and the percentage of flexible jobs determined the probability of a job being flexible. We raised the ratio of flexible jobs between 0% and 100% in steps of 25%. Figure 8 depicts the execution times for these configurations. In general, we can appreciate that the execution time decreases as the ratio of flexible jobs grows. The results show a 10%-gain with only a 50%-rate of flexible jobs, and up to a 12%-improvement when all of them are flexible.

E. Checking Period Inhibitor in Micro-steps Applications

For this test we reduced the time step in the model to an average of 2 seconds in order to investigate the importance of the overhead incurred by the scheduling process. Again, we generated workloads with a different number of jobs and executed them as both fixed and flexible workloads. For the flexible workloads, we enabled the *checking inhibitor* (see Section V-A) to prevent that each iteration triggers a check. Figure 9 depicts the variation of malleability for a fixed execution. The group at the top ("Flexible"), represents an execution without the *checking inhibitor* mechanism. The rest of the groups in the chart (bellow "flexible"), show the execution time when configuring the inhibitor period to: 2, 5, 10 and 20 seconds (from top to bottom). The conclusion is that this gain is virtually negligible or even negative in some cases. Hence, enabling periods of *checking inhibition* can avoid a burst of communications between the runtime and the RMS, reducing the overhead. In this particular example, setting a period of 5 seconds between scheduling, not only offers better results than the fixed workload, but also outperforms a simple flexible workload.

IX. Experimental Results

A. Testbed

For the experiments, we have generated workloads of different sizes, configuring the job submissions to the queue at their best performance.

We first evaluate the individual scalability of the three real applications. These tests identify two parallel behaviors:

- High scalability (CG and Jacobi). In this case both applications have a similar behavior, with the highest speed-up attained for 32 processes. However, from 8 processes on, the difference gain between tests drops below 10%, so we consider 8 processes as a "sweet configuration spot" for these two applications.
- Constant performance (N-body). In contrast, this application reaches its maximum performance for 16 processes.

However, in this case, the gain does not exceed 10% with respect to the sequential run, so a single process is considered as the "sweet spot".

From the perspective of computational cost, CG and Jacobi comprise "short" iterations that complete in less than 2 seconds, while N-body executes costly iterations, in the scale of minutes. For this reason, for CG and Jacobi we enabled the *scheduling period inhibitor* featured by the runtime, in order to reduce the amount of communications with the RMS.

Table I displays the configuration employed in the experimentation. The job submission of each application is launched with its "maximum" value, reflecting the user-preferred scenario of a fast execution. Each workload is composed of a set of randomly-sorted jobs (with a fixed seed) which instantiate one of the three real applications (33% of jobs of each application class). Furthermore, the inter-arrival time among submissions is generated using the statistical model proposed by Feitelson [6], which characterizes rigid jobs based on observations from logs of actual cluster workloads.

B. Results

Figure 10 depicts the execution time of each workload size comparing both configuration options: fixed and flexible. The labels at the end of the "flexible" bars report the gain compared with the fixed version. Table II details the measures extracted from the executions. In the first row, we compare the average resource utilization for fixed and flexible workloads. This rate corresponds to the average time when a node has been allocated by a job compared to the workload execution time. These results indicate that the flexible workloads reduce the allocation of nodes around 30%, offering more possibilities for queued jobs.

The second row of Table II shows the average waiting time of the jobs for each workload. These times are illustrated in Figure 11, together with the gain rate for flexible workloads. The reduction around 60% makes the job waiting time a crucial measure to keep in mind from the perspective of throughput. In fact, this time is responsible for the reduction in the workload execution time.

The last two rows of Table II present two more aggregated measures of all the jobs in the workload: The first one is the average execution time; the second is this execution time plus the waiting time of the job, referred to as completion time. The experiments show that jobs in the flexible workload are affected by the scale-down of their number of processes. However, this is compensated by the waiting time which benefits the completion time.

In order to understand the events during a workload execution, we have chosen the smallest workload to generate detailed charts and offer an in-depth analysis.

The top and the bottom plots in Figure 12 represent the evolution in time of the allocated resources and the number of completed jobs. It also shows the number of running jobs for fixed and flexible workloads (blue and red lines respectively). The figures demonstrate that the flexible workload utilizes fewer resources; furthermore, there are more jobs running

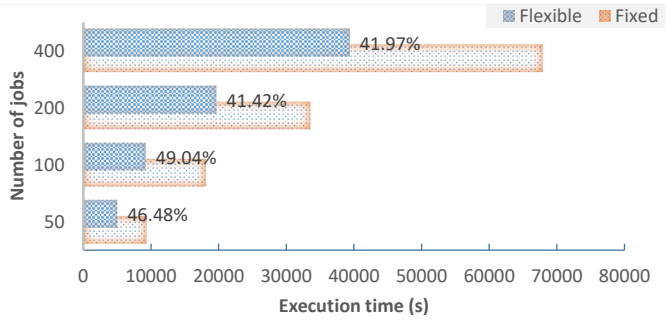


Fig. 10. Workloads execution times (bars) and the gain of flexible workloads (bar labels).

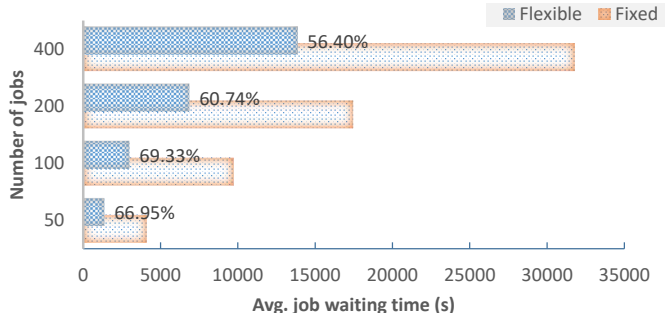


Fig. 11. Average waiting time for all the jobs of each workload (bars) and the gain of flexible workloads (bar labels).

concurrently (top chart). For both configurations, jobs are launched with the "sweet spot" number of processes; the fixed jobs obviously do not vary the amount of assigned resources, while in the flexible configuration, they are scaled-down as soon as possible. This explains the reduction on the utilization resources. For instance, in the second half of the flexible shape in Figure 12 (top), there are 5 jobs in execution which allocate 40 nodes. The next eligible job pending in the queue needs 32 nodes to start. Therefore, unless one of the running jobs finishes, the pending job will not start and the allocation rate will not be higher. When a job eventually finishes and releases 8 nodes, the scheduler initiates the job requesting 32 nodes. Now, the allocated nodes are 64 (the green peaks in the chart); however, as the job prefers 8 processes, it will be scaled-down.

At the beginning of the trace in the bottom of Figure 12, the throughput of the fixed workload is higher, but this occurs because the first jobs are completed earlier (they have been launched with the best-performance number of processes). Meanwhile, in the flexible workload, many jobs are initiated (blue line) and, as soon as they start to finish, the throughput experiences a boost.

X. Conclusions and Future Work

We have improved the state-of-the-art in *dynamic job reconfiguration* by targeting the global throughput of a high performance facility. For this purpose, we have taken advantage of first-class tools to design this new approach that introduce a dynamic reconfiguration mechanism for malleable jobs, composed of two modules: the runtime and the resource

TABLE II
SUMMARY OF MEASURES FROM ALL THE WORKLOADS.

	50 jobs		100 jobs		200 jobs		400 jobs	
	fixed	flexible	fixed	flexible	fixed	flexible	fixed	flexible
Avg. resource utilization rate	98.71 %	68.67 %	97.39 %	71.91 %	98.38 %	73.54 %	98.98 %	73.92 %
Avg. job waiting time	4115.02 s.	1359.92 s.	9750.34 s.	2990.6 s.	17466.2 s.	6856.8 s.	31788.39 s.	13861.03 s.
Avg. job execution time	620.26 s.	900.3 s.	586.64 s.	858.16 s.	520.58 s.	825.88 s.	532.14 s.	843.19 s.
Avg. job completion time	4735.28 s.	2260.22 s.	10336.98 s.	3848.76 s.	17986.78 s.	7676.67 s.	32320.53 s.	14704.22 s.

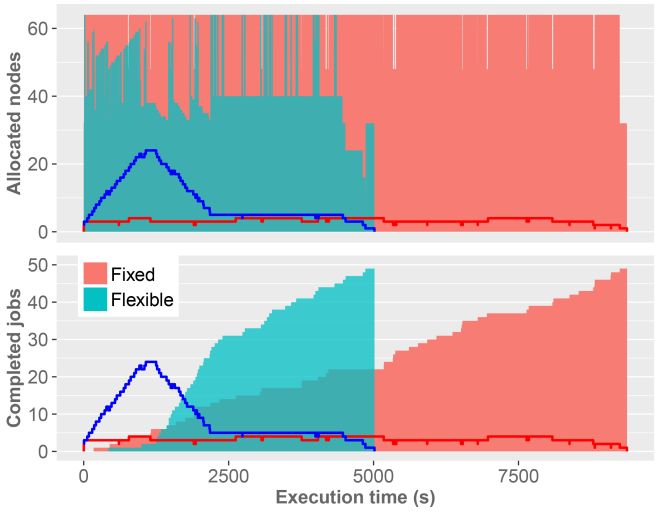


Fig. 12. Evolution in time for the 50-job workload. Blue and Red lines represent the running jobs for fixed and flexible policies.

manager. Those two elements collaborate in order to resize jobs on-the-fly to favor the global throughput of the system.

As we prove in this paper, our approach can significantly improve resource utilization while, at the same time, reducing the wait-time for enqueued jobs, and decrease the total execution time of workloads. In fact, resource utilization could still be improved if the job submission was not rigid, but flexible by giving a range of number of nodes required instead of a fixed value. Although this is achieved at the expense of a certain increase in the job execution time, we have reported that, depending on the scalability of the application, this drawback can be negligible.

ACKNOWLEDGMENT

This work is supported by the Project TIN2014-53495-R and TIN2015-65316-P from MINECO and FEDER. Antonio J. Peña is cofinanced by MINECO under Juan de la Cierva fellowship number IJCI-2015-23266. Special thanks to José I. Aliaga for the conjugate gradient code.

REFERENCES

[1] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: Simple Linux utility for resource management," in *9th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, 2003, pp. 44–60.
[2] "Slurm Workload Manager," http://slurm.schedmd.com.
[3] "The Top500 List," https://www.top500.org.
[4] Message Passing Interface Forum, "MPI: A message-passing interface standard version 3.1," http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf, Tech. Rep., Jun. 2015.
[5] F. Sainz, J. Bellon, V. Beltran, and J. Labarta, "Collective offload for heterogeneous clusters," in *22nd International Conference on High Performance Computing (HiPC)*, 2015.
[6] D. G. Feitelson and L. Rudolph, "Toward convergence in job schedulers for parallel supercomputers," in *Job Scheduling Strategies for Parallel Processing*, vol. 1162/1996, no. 5, 1996, pp. 1–26.
[7] J. Padhye and L. Dowdy, "Dynamic versus adaptive processor allocation policies for message passing parallel computers: An empirical comparison," in *Job Scheduling Strategies for Parallel Processing (IPPS)*, 1996, pp. 224–243.
[8] K. El Maghraoui, T. J. Desell, B. K. Szymanski, and C. A. Varela, "Malleable iterative MPI applications," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 3, pp. 393–413, Mar. 2009.
[9] P. Lemarinier, K. Hasanov, S. Venugopal, and K. Katrinis, "Architecting malleable MPI applications for priority-driven adaptive scheduling," in *Proceedings of the 23rd European MPI Users' Group Meeting (EuroMPI)*, 2016, pp. 74–81.
[10] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC10)*, Nov. 2010.
[11] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra, "Post-failure recovery of MPI communication capability: Design and rationale," *International Journal of High Performance Computing Applications*, vol. 27, no. 3, pp. 244–254, Jun. 2013.
[12] A. Gupta, B. Acun, O. Sarood, and L. V. Kalé, "Towards realizing the potential of malleable jobs," in *21st International Conference on High Performance Computing (HiPC)*, 2014.
[13] F. S. Ribeiro, A. P. Nascimento, C. Boeres, V. E. F. Rebello, and A. C. Sena, "Autonomic malleability in iterative MPI applications," in *Symposium on Computer Architecture and High Performance Computing*, 2013, pp. 192–199.
[14] G. Martín, D. E. Singh, M. C. Marinescu, and J. Carretero, "Enhancing the performance of malleable MPI applications by using performance-aware dynamic reconfiguration," *Parallel Computing*, vol. 46, pp. 60–77, Jul. 2015.
[15] G. Martín, M. C. Marinescu, D. E. Singh, and J. Carretero, "FLEX-MPI: an MPI extension for supporting dynamic load balancing on heterogeneous non-dedicated systems," in *Euro-Par Parallel Processing*, Aug. 2013, pp. 138–149.
[16] R. Sudarsan and C. J. Ribbens, "ReSHAPE: A framework for dynamic resizing and scheduling of homogeneous applications in a parallel environment," in *International Conference on Parallel Processing*, 2007.
[17] R. Sudarsan and C. Ribbens, "Scheduling resizable parallel applications," in *International Symposium on Parallel & Distributed Processing*. IEEE, May 2009.
[18] R. Sudarsan, C. J. Ribbens, and D. Farkas, "Dynamic resizing of parallel scientific simulations: A case study using LAMMPS," in *International Conference on Computational Science (ICCS)*, 2009, pp. 175–184.
[19] R. Sudarsan and C. J. Ribbens, "Combining performance and priority for scheduling resizable parallel applications," *Journal of Parallel and Distributed Computing*, vol. 87, pp. 55–66, 2016.
[20] S. Prabhakaran, M. Neumann, S. Rinke, F. Wolf, A. Gupta, and L. V. Kale, "A batch system with efficient adaptive scheduling for malleable and evolving applications," in *2015 IEEE International Parallel and Distributed Processing Symposium*, May 2015, pp. 429–438.
[21] E. Jajaga and J. Kllobocishta, "MPI parallel implementation of Jacobi," in *ICT Innovations*, 2012, pp. 449–458.