




# Job migration in HPC clusters by means of checkpoint/restart

Manuel Rodríguez-Pascual<sup>1</sup> · Jiajun Cao<sup>2</sup> · José A. Moríñigo<sup>1</sup>  · Gene Cooperman<sup>2</sup> · Rafael Mayo-García<sup>1</sup>

Published online: 23 April 2019

© Springer Science+Business Media, LLC, part of Springer Nature 2019

## Abstract

Until now, jobs running on HPC clusters were tied to the node where their execution started. We have removed that limitation by integrating a user-level checkpoint/restart library into a resource manager, fully transparent to both the user and running application. This opens the door to a whole new set of tools and scheduling possibilities based on the fact that jobs can be migrated, checkpointed, and restarted on a different place or in a different moment, while providing fault tolerance for every job running on the cluster. This is of utmost importance in the future generation of exascale HPC clusters, where an increasing degree and complexities of efficient scheduling make it challenging to obtain the required degree of parallelism demanded by the applications.

**Keywords** Checkpoint–restart · DMTCP · Dynamic job migration · Exascale clusters

## 1 Introduction

The recent rise in maturity of checkpoint–restart makes possible new functionality that was not previously practical. In particular, we describe a novel approach toward increased cluster performance and flexibility. The approach makes possible a uniform approach toward checkpointing for the large variety of jobs routinely executed by modern supercomputers, including OpenMP for many-core computing, MPI, and hybrid MPI/GPU computation.

This work has been motivated by the need to overcome deficiencies in the Slurm resource manager, as used at CIEMAT. At CIEMAT supercomputing

---

✉ José A. Moríñigo  
josea.morinigo@ciemat.es

<sup>1</sup> Department of Technology, CIEMAT, Avda. Complutense 40, 28840 Madrid, Spain

<sup>2</sup> Department of Electrical and Computer Engineering, Northeastern University, 360 Huntington Avenue, Boston, MA 02115, USA

facilities, a large collection of open source and proprietary codes is routinely executed, providing a diverse testbed with which to validate our approach to improved cluster performance and flexibility. CIEMAT is the largest Spanish research center for research on energy, the environment, and related technology. In addition to this basic research, there is also an emphasis on health science and high-energy physics. Similar to other large, national institutions (e.g., the U.S. Department of Energy), CIEMAT has included supercomputing facilities since the 1950s, and it has been a pioneer in the use of supercomputing in Spain. As one would expect, there are also a large number of scientists using CIEMAT's supercomputing infrastructure to perform simulations pertinent to a variety of disciplines. The result is a plethora of codes and applications, each with its own particular computational requirements. Most of these codes and applications come from releases widely used by huge communities around the world, and comprising millions of lines of code in Fortran, C, C++, etc.

In this scenario, adding an application-specific fault tolerance capability to each code is not practical. Hence, a uniform paradigm for software resilience must be provided by the HPC administrators. This resilience must be transparent to the codes run by the users. This aim fits closely with the increasing importance of fault tolerance in the coming exascale era. The current work is motivated by a need to go beyond current practice, by deriving new functionality for supercomputer management and performance.

The evolution of HPC hardware follows a clear path: an increasingly large number of processing units per cluster. For many years now, miniaturization and technical improvements have provided a larger number of resources, with limits of the technology and the nature of applications orienting processor design into simple ones with an increasing number of cores [1, 2]. Moreover, as performance varies over computers generations and characteristics, it is common to see heterogeneous clusters composed by resources with different performances, and even islands of clusters with notable differences in network connection among the nodes (see, e.g., the TOP500 list at <https://www.top500.org/>).

This evolution has had a deep influence on the software architecture. In particular, current and future tendencies in HPC design (including exascale-ready applications) come in the form of massively parallel applications or large arrays of serial applications with a post-processing step [3]. These must be executed in an efficient way, but the environment is also of importance to build new platforms that bear in mind the so-called exascale demonstration applications. This co-design pathway is a cornerstone nowadays [4].

There are two major factors that have a negative impact on the scalability of parallel applications: Amdahl's law, which establishes an upper limit on the application's scalability according to the fraction of parallel code, and the overhead of the different subsystems (network, memory) and its influence on the execution time. Both items must be accounted for in the exascale era [5]. They are partially overcome through new mathematical developments and application kernels, which increase the scalability, even at the cost of reducing efficiency. As a result of this variability on application requirements and limits, nowadays we can find a mix of codes with very different degrees of parallelism and execution times [6].

This combination of an increasing number of hardware elements and applications with different requirements and characteristics has resulted in complex and potentially problematic ecosystems. Among others, we identify three major issues related to the management of HPC resources: It is difficult to perform scheduling that achieves a high-throughput level; the administration of the cluster nodes is no longer trivial, as performing maintenance or updating them can have a negative influence on the currently running jobs; and last but not least, the potential for hardware failures to affect the execution of applications grows as the degree of parallelism increases. All of these problems increase both with the degree of parallelism of the applications and with their execution time. So finding solutions in the years to come is obligatory if we want to gain the most from the new generations of hardware.

Thus, it is still an open problem how to maximize cluster throughput while at the same time keeping a reasonable level of user's satisfaction (avoiding starvation, providing flexibility, implementing QoS, etc.). New approaches and scheduling techniques have been proposed [7, 8], but we are still far from finding a perfect and universal solution.

The maintenance and update of clusters is usually performed with scheduled downtimes. This standard practice leads to a period where the cluster is underutilized. This period includes: the downtime period itself; plus some time before it (when users will not be running anything to avoid losing their partial results); and sometime after it (until jobs have been submitted again and the cluster returns to full usage).

With regard to resilience, fault tolerance is currently achieved, at least in production environments, with the use of application-level checkpoints: modifying the application to save the state at given points during the execution. However, the use of application-level checkpointing/restart (C/R) libraries finds only limited use, due to the limited capabilities of most libraries and the lack of integration with *resource managers* (RMs). The lack of widespread use of application-level checkpoint–restart has several reasons. First, there is a significant cost in terms of development time for both implementation and maintenance. Second, an application-level approach places extra responsibility on the users: detect failures, manage temporary state files, and so on. Third, integrating application-level checkpoint methodologies in large codes with thousands of modules used by a heterogeneous community is far from trivial.

But perhaps the most insidious issue of all in application-level checkpointing approaches on large codes is that after some initial implementation effort, it actually works. Then, a new developer adds an additional module and fails to augment the application-level checkpointing code, which has now become somewhat complex. Most users do not notice the problem initially, since most users are not immediately using the features of the new module. But slowly, over time, the users are trained to believe that the built-in application-level checkpointing cannot be trusted except for the simplest of inputs.

The work presented here is aimed at addressing these issues through the use of C/R techniques completely transparent to the user, while integrating this with a RM. This is so because counting on a C/R library fully integrated in the RM allows one to save the state of all running jobs and restore them in case of problems. This way, we can overcome hardware failures and reduce their impact on the job execution time.

The question next arises if more potential outcomes can be achieved beyond simple resilience, once that the capability of saving the state of a job and restoring is available. In particular, if the job scheduler is aware of this possibility, it can be employed in the decision-making process, thus allowing one to modify the allocation of jobs in real time. For example, jobs can be moved inside the cluster to concentrate them in the minimum possible number of nodes, or jobs can be distributed evenly within a partially filled cluster. Other uses come from preemption, removing a running job from its resource to allocate another job with a higher priority.

The same approach can be applied to system administration: Any maintenance operation on a node can be performed immediately, by checkpointing all jobs running on that node and placing them back in the job queue. The users do not need to be notified, since the checkpoint and later restart will be completely transparent to them. Since there is no need to shut down the whole service, the impact of the process is greatly reduced.

All of the previous can result in a better computing and/or energy efficiency in the cluster and represents a new concept of novelty in this work as paves the way for adding more customized artificial intelligence capabilities in the RM depending on the cluster characteristics and general use. Thus, in order to support this new approach, the C/R library must satisfy certain minimum requirements in order to provide an optimum scenario according to the analysis carried out as part of this work:

- First, it must be able to save the state of all or most applications running on the cluster.
- Second, it must be transparent at the application level. The end user should not have to recompile or modify the application in any way, and the checkpoint and restart must be completely transparent to that end user.
- And third, of course, the overhead induced by the checkpoint library should be kept to a minimum.

The RM should also include some specific capabilities. In particular, it should include C/R support in the form of an API, so that a set of commands can be used to perform the checkpoint and restart operations. That API will then be called by a newly created scheduling algorithm, aware of the possibility of migrating jobs and using it in the decision-making process. Also, if we want to create user-level tools using migration, that API should be accessible from inside the RM. In addition, this API should be able to manage different C/R libraries and tools transparently.

Sadly, none of the existing RMs provide the required full support for a C/R library. We have therefore chosen the Slurm resource manager [9] due to its sophisticated features for extensibility as well as their vast use in the TOP500 list and modified it to include support for the user-level, transparent DMTCP C/R library [10] (see deeper analysis on why this library has been selected below).

Slurm is a well-known cluster management and job scheduling system for Linux clusters that requires no kernel modifications for its operation. It has been adopted by a wide set of administrators around the world for managing their supercomputers. In addition, it has a strong and active community of developers. Hence, this work has targeted Slurm due to the large potential impact. Nevertheless, one of

the main reasons for selecting Slurm is that the architecture of this RM provides a plug-in capability for integrating new functionality. This is used here to add a checkpoint–restart plug-in to Slurm, thus enabling new scheduling algorithms.

The architecture used here is able to manage: nodes (the compute resource in Slurm); partitions (grouping nodes into logical sets; jobs, or allocations of resources assigned to a user for a specific time slot); and job steps (sets of, possibly parallel, tasks within a job). Thus, several parameters can be defined in these partitions (job time limit, job size limit, user access permissions, etc.). These parameters enable the second objective of this work: the use of C/R to provide new features, including more accurate scheduling algorithms, dynamic migration of tasks, preemption, and transparent maintenance operations to the user.

In the rest of this document, we first describe the broad framework enabling this research: a full, transparent integration between Slurm and DMTCP. This is used to explore novel features on top of this integration:

- Creating a tool for system administration;
- Exploring how existing preemption mechanisms in Slurm are enhanced by this new possibility;
- Creating a new paradigm for batch jobs, the “eternal job”; and
- Proposing two scheduling algorithms employing live job migration as a demonstrator for more advanced artificial intelligence capabilities.

Following this, we include a results section describing the performance of this integration and conclude the article with some ideas about open problems and future work.

In the rest of this document, Sect. 2 presents an extensive review of related work for checkpoint–restart and current approaches in practice in HPC centers. Section 3 presents our approach to the use of checkpoint–restart for more efficient software resilience. Section 4 presents an experimental evaluation within the CIEMAT environment. Section 5 discusses a conclusion and future work.

## 2 Related work

As mentioned earlier, joint opportunistic user scheduling and power allocation are topics of major importance in modern HPC systems [11]. Among other aspects, it can be designed in order to achieve throughput optimization and fair resource sharing [12]. Beyond those, there is a link between fault tolerance and improvement in cluster throughput that this work is tackling, i.e., enhance the computational efficiency in a cluster profiting from checkpointing methodologies in order to design better scheduling algorithms, perform job preemption techniques, make administration maintenance operations transparently to the user, etc.

A description follows of the related work regarding checkpointing/restart libraries as well as live job migration.

## 2.1 Status of checkpoint/restart libraries

With the increasing use of parallelism in HPC, checkpoint libraries are becoming increasingly more valuable.

Checkpointing can be accomplished either at the system level (transparently to the application), or at the application level (integrated into the application). While the first type is easier to apply for the end user, the latter is typically more efficient [13].

In system-level checkpointing, the state of a computation is saved by an external entity. The complete process information has to be stored, including memory contents, open files, CPU, content of registers, and so on. On restart, the state is carefully restored, so that the execution can seamlessly continue at the same point where it was interrupted.

System-level checkpoint solutions can either be implemented inside the kernel or at the user level. The former has the advantage that the checkpointer has full access to the target process as well as its resources, while user-level checkpointers have to find other ways to gather this information. On the flip side, user-level schemes are typically more portable and easier to deploy.

Regarding current projects, BLCR [14] is primarily a kernel-based implementation of checkpoint/restart, employing a Linux kernel module. It is fast and efficient. However, the kernel module must be re-compiled and possibly re-tuned for each particular Linux kernel version. Another weakness of BLCR is that it does not support the SysV enhancements, such as System V shared memory. Many MPI implementations employ System V shared memory as an optimization for message passing among MPI ranks on the same node. While most MPI implementations can be configured to avoid making use of System V shared memory, this is non-optimal. At the same time, coordinated checkpointing and rollback recovery for MPI-based parallel applications has been provided by integrating BLCR with LAM and other implementations of MPI through a checkpoint–restart service specific to each MPI implementation [15].

The updated DMTCP version [16] is a strictly user-space, system-level checkpoint. It makes use of a simple, yet powerful idea in order to be able to capture all state related to the running application: A DMTCP library is injected into each running process, and that library starts a DMTCP-specific checkpoint thread. With this configuration, DMTCP can monitor all activities in the process. As a drawback, this approach adds a thin software layer. In most applications, the overhead due to this software layer is usually smaller than the jitter, or variation in time, when the application is re-run. However, a notable exception is its support for InfiniBand; the overhead can be measured at a fraction of 1%. This non-negligible overhead will be determined later in this work.

CRIU [17] is a promising new project, but it is still not able to checkpoint parallel or distributed applications. It does, however, provide the possibility of checkpointing Docker containers along with some other interesting features [18]. A similar conclusion can be made for the compiler-based lightweight memory checkpointing (LMC) [19], which has demonstrated low performance overhead with strictly bounded memory usage at runtime as demonstrated on server applications.

The alternative to system-level checkpointing is application-level checkpointing, where it is the application that writes its own state into a checkpoint file and reads it on the restart stage. While this requires explicit code inside the application and hence is no longer transparent, it gives the application the ability to decide when checkpoints should be taken and what should be contained in the checkpoint (see, e.g., works on incremental checkpointing applied to application level [20, 21]).

In the case of application-level checkpoint, probably the most extensive solution is scalable checkpoint–restart [22]. SCR is a multi-level checkpointing, which allows applications to take both frequent inexpensive checkpoints and less frequent, more resilient checkpoints, aiming to obtain a better efficiency and reduced load on the parallel file system. In order to do so, they perform checkpoints and save the resulting image in increasingly safer, but slower, places in the memory hierarchy. They have performed an impressive work on data allocation, thus minimizing data movement around nodes, I/O serialization in network disks, and interesting replication techniques to overcome different set of failures with minimal effort and time loss. The fault tolerance interface (FTI) [23] has a similar approach to the problem, differing in the way that the library deals with checkpoint files and metadata. A new project supported by Argonne National Laboratory is underway: VeloC. This represents a merging of the design of SCR and FTI, and a prototype integration with DMTCP has been demonstrated.

Both system-level and application-level approaches have advantages and disadvantages. While system-level checkpoints provide full transparency to the user and require no special mechanism or consideration inside an application, this transparency is missing in application-level checkpointers. On the other hand, this transparency comes at the cost of high implementation complexity and storage overhead for the checkpointing software. Nevertheless, for many users who are primarily interested in obtaining scientific results without the need for an in-depth study of the underlying code base, the transparent user-level checkpointing approach is a major step forward.

## 2.2 Live job migration

The possibility of moving “something” being executed on a particular computational resource to another one is not new at all. It has been around for at least 25 years. Thus, in the lowest level of abstraction, task migration is widely used inside multi-core processors to distribute tasks among the existing cores, distributing and balancing the load [24]. This is now an active line of research for virtualized environments [25–27].

At a higher abstraction layer, Charm++ [28] is a parallel object-oriented programming language based on C++ with the ability of applying it to fine grain parallelism and using it to checkpoint individual threads or active objects in a large-scale system to achieve load balancing.

A methodology can be proposed for dynamic job reconfiguration based on the use of MPI\_spawn in order to dynamically adjust the number of nodes in actively

running MPI jobs in Slurm [29]. After introducing some small modifications in the code to indicate where the execution can be arbitrarily distributed, their framework gathers all the available resources and increases the degree of parallelism of the application at runtime. This approach, however, requires modifications of the source code suitable only for certain kind of applications, thus preventing a wider adoption.

Also, the inclusion of resilience capabilities into the MPI standard through the proposed user-level failure mitigation (ULFM) has enabled the implementation of resilient MPI applications [30]. Its low overhead when tolerating failures in one or several MPI processes has been shown [31]. This solution is built on top of Compiler for Portable Checkpointing (CPPC), an application-level checkpointing tool for MPI applications. Thus, the proposed development transparently makes MPI applications resilient by instrumenting the original application code; this does require a previous customization of the code to be checkpointed.

Regarding full jobs, the HTCondor scheduling system [32] applies checkpoint/restart to complete jobs (although only serial ones) to achieve a better utilization of compute clusters for high-throughput computing. Since most of the serial jobs are components of larger Monte Carlo simulations, achieving fault tolerance is less important.

There also exists a formal approach that can be applied to multi- and many-core chips. Several algorithms and mechanisms have been proposed [33, 34], although their impact is beyond the scope of this work.

Similarly, the most widely used VM (virtual machine) managers (Xen, OpenVZ, KVM, VirtualBox, etc.) support live migration of full VMs. This possibility is regularly employed by platforms such as OpenNebula [35] and OpenStack [36] to rearrange running VMs inside the clusters, usually with the objective of concentrating the VMs in the fewest possible number of resources to reduce energy consumption. In the case of software containers, this technique has still not been widely adopted, and to the authors' knowledge, only Docker + CRIU offers this possibility [37].

Of course, the technologies employed vary depending on the software layer, but the underlying ideas are always roughly the same.

From the existing work presented in this section, it can be inferred that the checkpointing of jobs in local clusters is an ongoing work that has already provided useful results and production-ready tools. Live migration of different elements (tasks, virtual machines, etc.) is a mature technology, suitable for many situations. However, there is still a lack of a real and effective connection between these two areas in HPC environments. Thus, fault tolerance manager (FTM) for coordinated checkpoint files is able to provide users automatic recovery from failures when losing computing nodes [38], though it is particularly useful in infrastructure-as-a-service cloud platforms environments, and is based on the RADIC architecture. Finally, task migration for fault tolerance in heterogeneous multi-cluster systems exists, but was developed primarily for grid computing infrastructures [39].

Motivated by the issues described above, the present work presents a transparent, novelty approach to live migration of tasks based on jobs checkpointing and restart, which incurs in low overhead. Under this approach, more accurate scheduling, dynamic migration of tasks, flexible preemption and maintenance operations are possible, hence opening a new set of capabilities to be seamlessly performed



by system administrators. Thus, such a design presents not only a new successfully tested engineering solution, but rather a new and novel approach and concept to better exploit supercomputers not previously foreseen to the authors' knowledge.

### 3 Checkpoint/restart for job migration

In this section, we present our solution for seamlessly integrating a checkpoint/restart library into a resource manager. Then, we describe the tools and scheduling algorithms that can be created based on this new capability, thanks to job migration.

#### 3.1 Software stack

For this work, we have chosen Slurm as the resource manager and DMTCP as the checkpoint library.

Slurm is one of the most widely employed resource managers for supercomputers. In particular, it is the workload manager on about 60% of the TOP500 supercomputers according to their main developer, SchedMD. It is an open-source tool, and so we are able to dig into its internals and modify them according to our needs. Further, due to Slurm's modular and plug-in-based design, these modifications of the internals are kept to a minimum.

DMTCP is chosen as the checkpoint library. As discussed in "Related Work" section, it is, to the authors' knowledge, the only checkpoint library that provides the needed requirements for this project: open source; fully transparent to the users' and applications; support for parallel applications (MPI and OpenMP); stable; and continuing active support by its development team. In addition, petascale-level checkpointing has been demonstrated by DMTCP through a new mechanism for virtualization of the InfiniBand UD (unreliable datagram) mode and for updating the remote address on each UD-based send. Results have demonstrated low overhead in tests with real applications and benchmarks running on more than thirty thousand MPI processes and CPU cores [16]. An extrapolation of those results to future SSD-based storage systems shows that this approach will remain practical in the exascale generation.

Note, also, that this project is not tied to DMTCP. The modular design of Slurm allows one to change the checkpoint library without affecting the rest of the tool, since the library is modularized through a well-defined API. Thus, a different, future library could be adopted by simply modifying a configuration file.

#### 3.2 Developments for job migration

The integration of Slurm and DMTCP was performed using an existing checkpoint API present in Slurm. This was implemented by creating a shell wrapper for each of the three checkpoint functions (start, checkpoint and restart). There were some

challenges due to concurrency issues when starting MPI jobs, but these were solved through a lock mechanism on files in a shared folder.

It is important to note that this plug-in has a behavior opposite to the original Slurm design for checkpointing. Originally, Slurm was designed to start a job with checkpoint support only if requested by the user through a command-line flag during submission. Since we want to support *all* running jobs in order to more broadly support job migration, we changed this default behavior to start the application with checkpoint support providing that the user does not disable checkpointing during submission through a command-line flag.

Aside from the plug-in, the changes made in Slurm were kept to a minimum. The structure, APIs, and existing functionality were maintained for compatibility with the official Slurm version. The modifications only add optional extensions to the existing API calls.

### 3.3 New tools and functionality in Slurm enabled by job migration

The following section is devoted to describe the tools and functionality enabled by the availability of job migration inside clusters. We first present smigrate, a tool for cluster system administration that employs job migration to idle nodes in a fast and secure way, and, then, two different scheduling algorithms using job migration to dynamically reallocate running tasks. Note that the objective of this section is not the creation of complex tools and algorithms, but to demonstrate how the implemented job migration inside clusters can enable a new and wide set of additional tools for cluster administration.

For the sake of completeness, it is worth noticing that optimization of task assignments on parallel computers is carried out in Slurm by a best fit algorithm based on Hilbert curve scheduling or a fat tree network topology [40].

#### 3.3.1 Smigrate—a job management tool

The nodes composing a cluster need to be maintained (i.e., removed from active service on the cluster to perform operations related to system administration) on a regular basis. Reasons include, but are not limited to, software updates, reconfigurations, network issues, changes on the hardware, etc. In normal usage, users are notified of these updates in advance, since part or all of the cluster will be out of service.

This maintenance process greatly harms the computation throughput. Although the update itself may be short, users are warned in advance not to submit long jobs and the cluster remains not fully occupied immediately after the update, since users have not yet submitted new jobs. Moreover, if the maintenance is urgent and there is no time to notify users in advance, then the jobs running on those nodes are simply killed and the corresponding computation time is lost. This is especially harmful in the case of parallel applications.

As an alternative, job migration can be used to avoid the loss of computation time. If a particular slot, node or part of a cluster must receive maintenance, then the tasks running there can be migrated. With this goal in mind, we have created smigrate, an application that allows administrators and users to manually migrate jobs.

The process is as follows:

- The user executes `smigrate` indicating the node to be emptied of jobs.
- `smigrate` asks Slurm if the user has enough privileges to do so, and exits if not.
- The node is marked as “DRAIN.” This is a Slurm state indicating that Slurm is operative, but not accepting any more jobs, and will go offline as soon as the currently running jobs complete. The state is changed to “DRAIN” without the need for additional checks, thus avoiding race conditions.
- `smigrate` checks if all the jobs on the node are checkpointable (see Section [DEVELOPMENTS FOR JOB MIGRATION] for details), and if not, then `smigrate` sets the node back to “AVAILABLE” and exits.
- The user can use a flag to decide what to do if there is a parallel application running on several nodes, one of which is the node marked “DRAIN.” The flag is checked here, setting the node back to “AVAILABLE” and exiting if the migration is not desired.
- At this point, there are no issues preventing the node to be emptied of jobs.
- All jobs running on the node are checkpointed and placed back in the job queue. Their priority is set to maximum, so that they will be executed on the first available nodes that satisfy the job requirements.
- The node has now been emptied of jobs and is ready for maintenance.

After the desired tasks have been performed on the node and it is ready to go back into production, it remains to update its status to “AVAILABLE.” Slurm will then place the node into the resource queue again and submit the corresponding restart jobs.

Last, it is important to note that this process is not limited to one node at a time. Instead, an arbitrary set of nodes can be emptied of jobs at the same time.

The `smigrate` tool is implemented in C language as Slurm is and can be downloaded from [41], where the reader can find the whole bunch of programs (source, object, header, makefile, etc.), a wiki page with instructions about how to install and execute it, and even a demo video.

### 3.3.2 Preemption and “eternal” jobs

The preemption policy establishes that a running job may be canceled so that another one with higher priority can use their resources.

Without the possibility of checkpoint/restart, the usage of preemption mechanisms was extremely limited as it implies losing all the computational effort invested in the job being canceled. The ability to use checkpoint/restart represents here a game changer, as these low-priority jobs can be re-queued and their execution continued just as the higher-priority job completes, or when the first resource becomes available.

Preemption can now be used for a wider set of purposes, as the drawbacks that kept the mechanism from a wider adoption have disappeared with the use of checkpoint/restart. In particular, we have identified and tested three different use cases.

A basic usage for preemption is the support for queues with different priorities. In this use case, Slurm is configured so that if there are pending jobs on high-priority queues, the ones running on low-priority queues are preempted and re-queued. In this way, urgent jobs can begin their execution as soon as possible, with the only

cost being to delay the execution of low-priority jobs. Of course, users could abuse this system by submitting all their jobs with high priority, but the Slurm quota system makes it straightforward to avoid this situation.

A similar situation happens in clusters where resources are limited to a specific set of nodes, such as Xeon Phi accelerators being present only in some nodes. Currently there are two alternatives: leave these nodes with special resources idle for jobs requiring those resources; or use them for any job, with the ones with special requirements having to wait. By using preemption, we can execute any kind of job on these nodes and obtain full usage from the cluster, moving these jobs away when there is a specific job requiring the resource.

Perhaps the most interesting use case, only enabled by the use of checkpoint/restart-based preemption, is the creation of *Eternal Jobs*. The underlying idea is that some users have a computational demand that exceeds the available resources, especially in the shared environments typical of clusters. A solution is the creation of a low-priority queue of serial jobs that are preempted whenever a more important job arrives. There is no need to set a particular length for these jobs, hence the qualification of “eternal”: They simply run whenever the cluster is not fully occupied, and they are preempted when new “normal” jobs arrive. In this way, the cluster increases its usage and the most demanding users can be assigned more CPU time, at the sole cost of yielding to other jobs with a higher priority.

Together, these use cases demonstrate a new approach to the cluster administration with more flexible tools, through the use of checkpoint/restart.

### 3.3.3 New scheduling algorithms

Traditional scheduling algorithms determine where and when a job should run. This is decided taking into account several factors: job information provided by the user; state of the cluster; and future demand based on the pending job queue. After a decision is taken and the job starts its execution, the process has finished.

Live job migration adds another dimension to the scheduling process: the possibility of altering the execution of a job by saving its state, canceling it, and then restoring it on a different physical location and/or in a different moment. This way, the scheduler can adapt to either changes in the infrastructure or on demand.

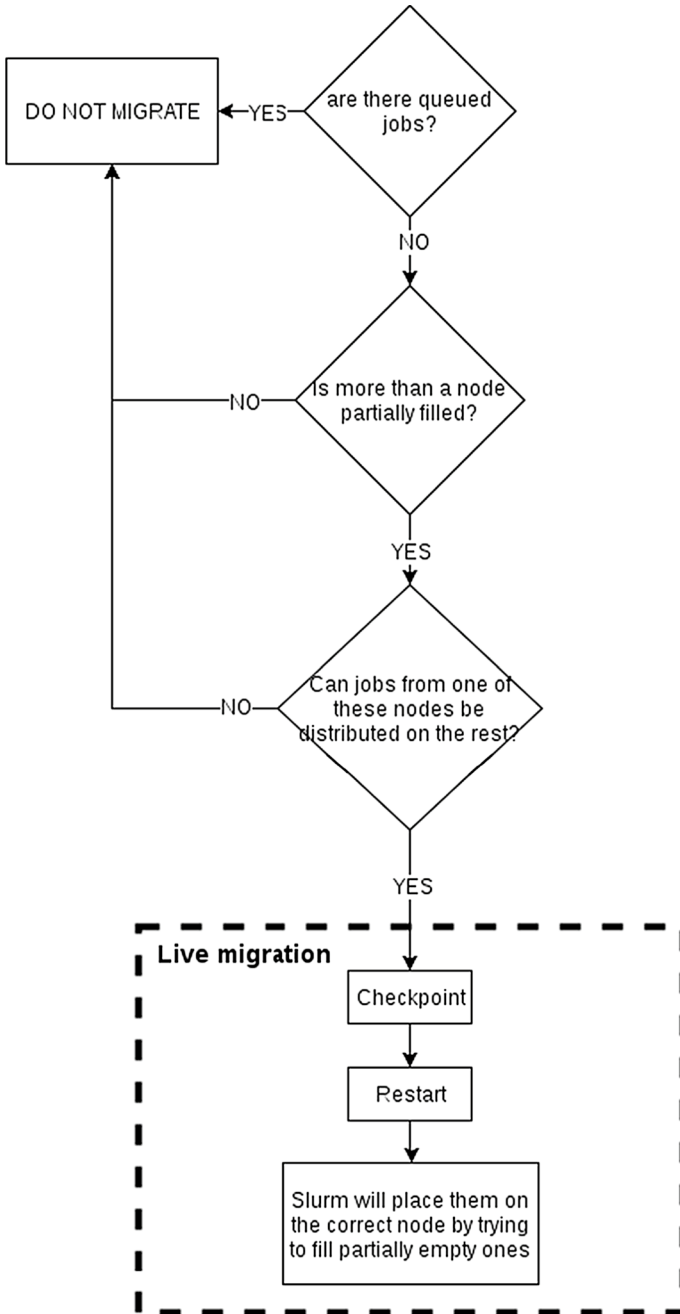
Although creating sophisticated scheduling algorithms is out of the scope of this work, we consider of high interest to demonstrate these new capabilities. For this sake, we have designed two algorithms with different behaviors.

### 3.3.4 Scheduling for job compaction

The first algorithm is devoted to job compaction. Its objective is concentrating the jobs running on the cluster in as few nodes as possible, leaving the rest of the infrastructure idle. This can be performed for several reasons, like to make the infrastructure available for parallel tasks or to reduce power consumption by switching down or reducing the voltage of empty nodes.

Algorithm 1 describes this process from a high-level point of view.

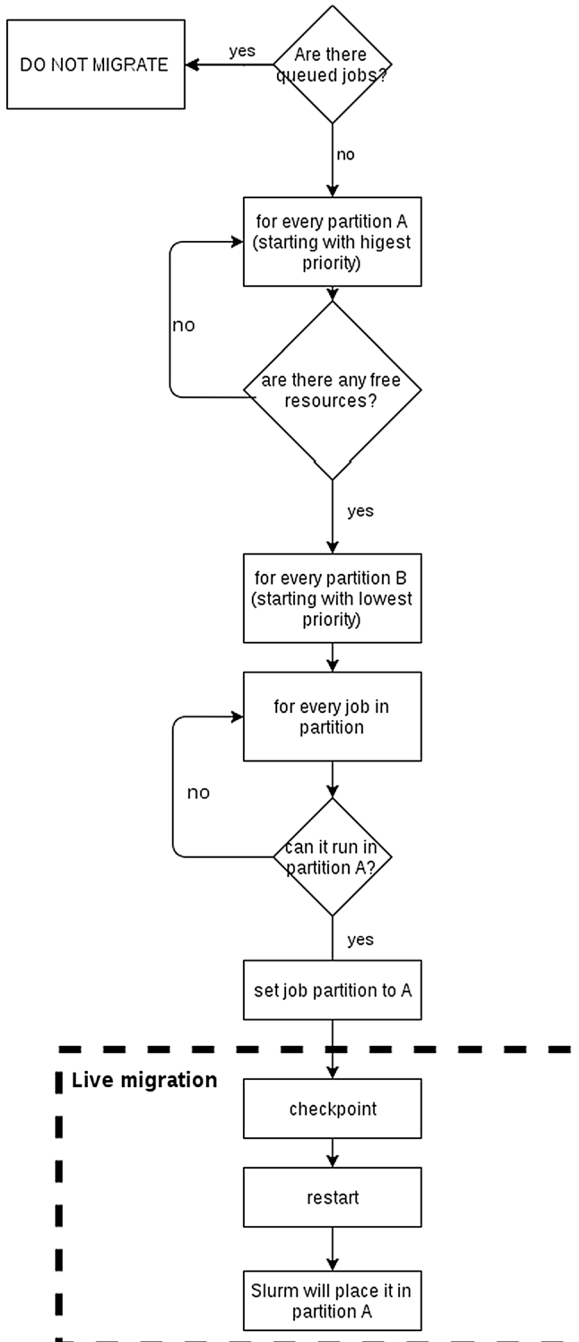
**Algorithm 1:** Job Compaction scheduling



The behavior of this algorithm is straightforward: If there are queued jobs in the cluster, we can safely assume that Slurm will automatically try to place them filling partially filled

nodes; if not, the algorithm sees if the compaction makes sense and, if so, checkpoints the jobs from the node to empty and restarts them in the rest of the partially filled ones.

### Algorithm 2: Scheduling with priorities



A similar concept can be employed to migrate jobs among resources with different priorities. This tool can be very useful on facilities where certain nodes present different characteristics than others.

The first and most obvious example is a heterogeneous cluster where certain nodes are more powerful or have less power consumption than others. In this case, it makes sense to use these nodes as much as possible, while leaving the less powerful or more consuming ones idle. In particular, application performance over KNC/KNL of non-ported jobs will not be optimum.

Another straightforward use is when certain resources are scarce, as depicted in “preemption and eternal jobs” use case, but with a proactive approach: as soon as it is possible, empty the scarce resources so they are available when needed.

This approach can also be handy in an island of clusters. In periods where the island is not fully used, jobs can be concentrated and a full island emptied and shut down. Also, by concentrating smaller jobs in a reduced number of islands, larger parallel jobs can be executed on a single island and not split among several ones, thus reducing communication overhead.

## 4 Experimental results

In this section, we will describe a series of experiments performed to analyze the performance of the proposed solution and its usefulness on different conditions. With respect to how these results have been obtained from the operational point of view of the solution presented in this work, Ref. [41] contains a wiki page and link to a demonstrating video as it has been previously mentioned.

The experiments presented here were performed on a small local cluster, ACME, which includes eight computing nodes. Each node consists of a Supermicro X10DRT-P motherboard with two 8-core Xeon E5-2640 v3@2.60 and 32 GB DDR4 RAM. Regarding storage, it includes a HD (1 TB) for scratch and temporary files and network storage for the user’s home directory and the non-OS applications (scientific codes). Measured latency is 1.5 microseconds for the HD and 7.35 microseconds for the network storage, roughly an order of magnitude larger than RAM latencies.

### 4.1 DMTCP overhead

There are several factors that have an influence on the time employed for a checkpoint/restart operation. Among them, some of the most important are the size of the application in terms of memory and temporary files, the storage speed, and the Slurm configuration. This section is devoted to measuring this overhead, so its influence in the total execution time of an application could be determined.

The Slurm resource manager is designed to control the execution of thousands of jobs at the same time. To do so, it includes a sophisticated cache mechanism to store the job state, thus greatly reducing the database overhead. This information is stored and processed at a fixed rate. Until this happens, applications

checkpointed and canceled cannot be restarted, so this adds an overhead of half of the `PURGE_JOB_INTERVAL` parameter, where `PURGE_JOB_INTERVAL` is the frequency with which the cache is processed. This value is 5 min by default, although in this article it has been reduced to 1 min with no harm. The exact value must be fixed according to the cluster size and workload.

We performed an analysis in order to determine the time employed by DMTCPC to checkpoint and restart an application. To do so, we ran an MPI test application with different memory sizes and numbers of tasks, thus determining the impact of size and degree of parallelism. These values were calculated with a script that ran this application, checkpointed it, canceled it, and then tried to restart it until the operation was allowed by Slurm (meaning that the cache had been cleaned). This operation was repeated ten times for every experiment and the values averaged. The same application was executed without DMTCPC support and no checkpoint in order to obtain a base value that was deleted from the previous one, ensuring that it only corresponds to C/R related overhead.

Table 1 reflects the results obtained. It is worth noticing that this table embraces all the existing overheads, including the one created by `PURGE_JOB_INTERVAL` (30 s on average). As can be seen, checkpoint/restart overhead goes from about a minute in serial applications to ten minutes in the case of large, highly parallel codes. It is important to note that this is an extreme case, as it means saving the state of everything running on the cluster to a network storage and recovering it again. This overhead is much smaller when the size of each task is 50 MB instead of one GB. So it is clear that the bottleneck is I/O and not DMTCPC or Slurm.

As mentioned before, DMTCPC has demonstrated good results in several extensive scalability analyses performed on some of the largest machines on the planet [16]. The results are consistent with the ones provided here and show an adequate scalability of the library on very large infrastructures, thereby demonstrating its feasibility for this project.

## 4.2 smigrate

As has been previously stated, smigrate is proposed for cluster administration. It allows emptying a node using job migration and so it can be maintained by the system administrators.

Before smigrate, the node had to be marked as “draining,” so that Slurm would not assign any other job to the node and removed it from the list of available nodes. The average time to empty a node can be assumed to follow a binomial distribution on the average execution time  $T$  and the number of jobs per node  $N$ , i.e.,

$$T \sim 1 - 1/2^N$$

This value largely varies among clusters depending on the number and size of jobs, but according to an analysis of the traces submitted to Parallel Workload Archive [42], it can be deduced that it is on the order of a few hours.



**Table 1** Overhead obtained by using DMTCP on C/R purposes

	Number of tasks	Application size in memory		
		50 MB	100 MB	1000 MB
		Measured overhead [s]		
1	51	53	89	
2	51	54	97	
4	52	55	115	
8	53	59	144	
16	56	66	202	
32	52	79	343	
64	66	98	398	
128	99	157	608	

Comparing that value with that presented in Table 1, we can see that our approach obtains performance gains of up to several orders of magnitude and in every single case is significantly faster.

### 4.3 Preemption and eternal jobs

Once C/R is implemented and fully integrated into Slurm, job preemption is completely straightforward. The same happens with the Eternal Job paradigm, which can be considered a natural consequence of preemption.

```

PartitionName=eternalJobs
Nodes=<node list>
PriorityTier=1
PreemptMode=CHECKPOINT (...)
PartitionName=normal
Nodes = <node list>
PriorityTier=2 (...)

```

With the following configuration on `slurm.conf`, jobs running on partition “eternalJobs” will be checkpointed and re-queued every time that a job running on partition “normal” is assigned to the same node. By making both `<node list>` equal, but sorted the opposite way (this is, if one `<node list>` reads {a,b,c}, the other reads {c,b,a}), eternal jobs are firstly assigned to the nodes that normal jobs use last, i.e., that are only used when the cluster is under full usage.

It is important to consider that, while conceptually simple and already taking advantage of support in Slurm, this technique was not really usable until the integration of DMTCP into Slurm was implemented in this work, as there was a lack of a transparent, universal, and reliable C/R library.

#### 4.4 New scheduling algorithms: scheduling for job compaction

Although creating sophisticated scheduling algorithms is out of the scope of this work, we consider of high interest to demonstrate these new capabilities. For this sake, we have designed two algorithms with different behaviors that will serve as a proof of concept and initial successful test of the new possibilities that this work opens up. In what follows, the idea for designing the algorithms, i.e., the goal they are pursuing, is briefly described.

Next we will show an analysis of the two previously presented different scheduling algorithms that makes use of job migration: the scheduling for job compaction and the scheduling with priorities.

In order to measure the effect of job migration in the occupancy of nodes, we have executed a set of identical workloads with and without migration for job compaction, with this being the only change over Slurm default scheduling policy (a FIFO queue). The cluster status was being monitored every 5 s.

These workloads have been randomly generated, using the analysis present in [42] to propose realistic cases. This information comes from 20 traces from different systems with millions of jobs. In particular, the execution time and degree of parallelism are both known to follow an exponential distribution. The arrival of individual jobs tends to display daily and weekly cycles, but as it largely depends on the system we have been considered it random for the purpose of this work. Doing so and bearing in mind that random workloads are harder to optimize than sequential or periodic ones, we stress our solution to the worst potential case that it could face in order to test its suitability and correctness.

In order to determine whether this approach is useful with different workloads, we have generated four different ones embracing 25%, 50%, 75% and 90% of the system resources. As shown in Figs. 1 and 2, their distribution is not perfect, as every job has been randomly generated following the aforementioned principles.

Figure 3 shows the result of a given execution. Here, jobs representing 50% of the available CPU time were submitted over 24 h. The same experiment was repeated with and without job migration. The abscissa axis depicts the nodes belonging to ACME that have been used in this test. The left column per node depicts the node occupancy when no job migration is carried out and the right column depicts the same node occupancy when the algorithm designed for scheduling and compacting jobs is ruling the workload. As can be seen, migration has caused there to be a higher percentage of nodes either full occupied or fully empty along a larger fraction of the time. On the other hand, there is a reduced percentage of nodes presenting a mixed state. That is, the desired result is obtained in order to allow new scheduling algorithms, a better usage of the cluster with nodes fully occupied (their percentage has been increased from 18.4% to 26.4%), nodes fully empty (from 28.1 to 31.9%), etc. Thus, in the overall experiment, the percentage of nodes with a mixed behavior (partially occupied) has been reduced from 53.5 to 41.7%.

As can be easily inferred, designing more complex algorithms could drive an even better result. As mentioned before, in this work we present the feasibility of the proposed approach and how a new path for designing more accurate scheduling policies is available from now on.

Degree of parallelism in jobs

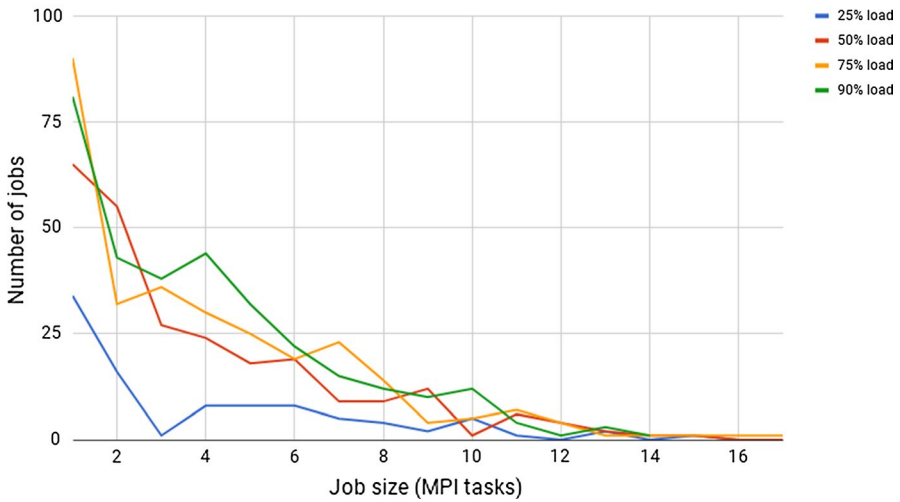


Fig. 1 Randomly generated workloads for this work representing 25%, 50%, 75%, and 90% of the system resources

Job Length

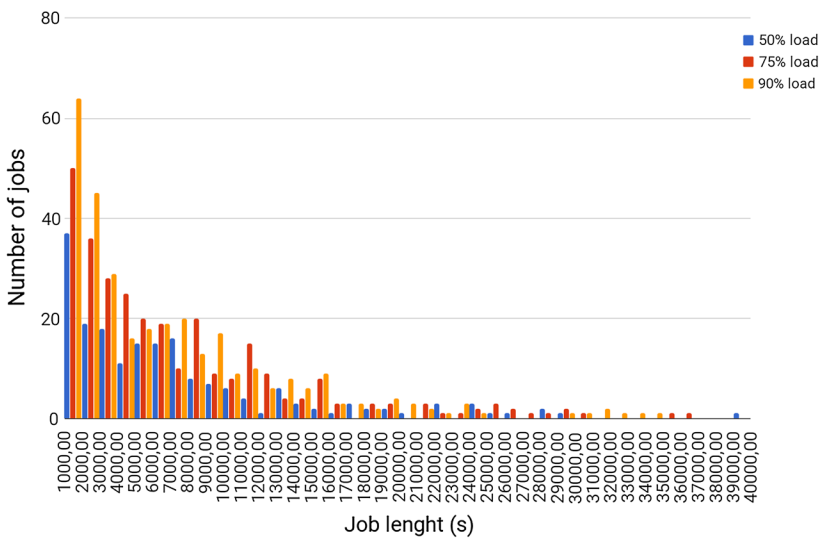


Fig. 2 Distribution of the job length in the workloads depicted in Fig. 1

In a more general way, Fig. 4 presents the values of repeating this experiment with different workloads. As can be seen, in all cases the migration has been useful to compact the running jobs in a smaller number of nodes. The percentage of nodes with a mixed behavior (partially occupied) has been reduced from 33.8 to

Node occupancy rate with 50% load

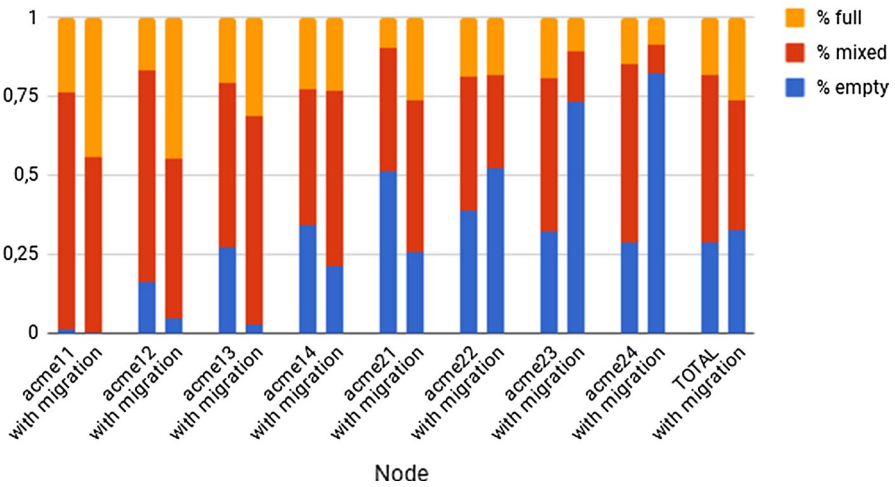


Fig. 3 Node occupancy with and without migration for job compaction with a total workload of 50%. The last couple of columns in the right of the figure depicts the total result of the test, being the rest the results per node

Node usage

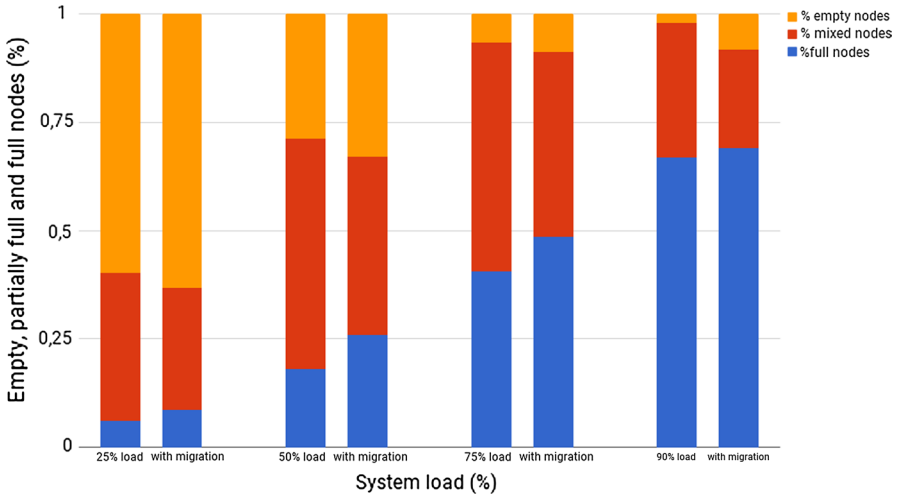


Fig. 4 Cluster state with different workloads. Execution time is one day

28.2% when the workload represents 25% of the cluster, from 53.5 to 41.7% for the 50% case, from 52.8 to 42% for the 75% case, and from 30.5 to 22.3% for the 90% case. In other words, the difference in the percentage of partially used nodes without and with migration has two behaviors: When the cluster is either empty or almost fully load (25% and 90% of workload, respectively), the algorithm for the migration

of tasks for compaction improves the cluster occupation in 5.6% (25%) and 8.2% (90%). On their side, when the workload is in a mid-range, this improvement is better: 11.8% for the 50% workload case and 10.8% for the 75% one.

#### 4.5 New scheduling algorithms: scheduling with priorities

The same methodology has been followed when testing the algorithm “scheduling with priorities.” In this case, the objective was to have the nodes in which the “high-Priority” queue was configured to be as full as possible, moving the jobs there from the “lowPriority.” To do so, we divided the cluster into two partitions with different priorities and created random jobs that could run on any of them. Slurm would then try to run every incoming job on the high-priority queue if there were free resources and on the low-priority one if not.

This scenario could be of great interest in heterogeneous clusters where there are nodes containing different processors, i.e., there are nodes which present a higher computing power than others. By defining an algorithm that will take into account the peak performance of these different zones of the whole cluster, for example, tasks could be migrated to/from the “highPriority” queue in order to obtain a better usage of the supercomputer. This concept could be extended to the clusters composed of islands.

The first thing to take into account is the pre-fixed conditions for this test. In this case, we already defined these two boundary cases:

- The percentage of nodes fully empty was reduced to a minimum in the nodes where the “highPriority” queue was configured, just around 1%. Thus, there were not many free cores to which migrate jobs.
- Only two out of the eight nodes were configured with the “highPriority” queue.
- The test was carried out with the highest workloads.

Doing so, we set up experimental conditions against the migration of tasks, i.e., drawbacks that were limiting the possibility of migrating tasks. The aim of this test definition is to demonstrate that if we are able to improve the cluster efficiency in such an unfavorable scenario, it can be derived that the possibility of migrating tasks will be useful in general conditions.

Table 2 shows a comparison of the Slurm default scheduling algorithm with a migration-based one. As can be seen, migration can be a helpful tool for this kind of environments.

Thus, the percentage of free nodes remain constant in those nodes where the “highPriority” queue was configured (1.05% and 1.25%), which indicates that the test has been properly carried out according to the first boundary previously mentioned. Second, with the 75% load, there is an increase in the time in which the nodes are fully used in the “highPriority”-related nodes along the test, moving from 25.58 to 31.64%. As the percentage of nodes with a mixed usage is decreased, the cluster is more efficiently used and complies with the purpose of the test. Third, when the cluster is almost fully used, the percentages are roughly the same (when

**Table 2** Results of the test in which two queues with different priorities were configured

	Without migration		With migration	
	Low priority	High priority	Low priority	High priority
75% load				
%empty	20.33	1.06	19.24	1.05
%mixed	43.49	73.36	48.33	67.31
%full	36.18	25.58	32.43	31.64
90% load				
%empty	7.72	1.25	7.72	1.25
%mixed	31.65	57.56	36.85	58.02
%full	60.63	41.19	55.43	40.73

The percentage of the nodes of the cluster without and with the migration of tasks via checkpointing/restart is shown

even a small decrease in the test with migration of tasks), a result that is consistent with the fact that there is almost nothing to migrate to.

Again, it is worth mentioning that the scheduling algorithms used for the experiments comprise two different scenarios, to demonstrate the feasibility of the live task migration, i.e., counting on more sophisticated algorithms in which artificial intelligence and/or stochastic processes would be applied would be conducive to even better results.

## 5 Conclusion and future work

There is a clear need for checkpoint/restart in current and future HPC exascale systems. Platforms with millions of cores and thousands of nodes are expected to suffer of more errors that should be overcome in an efficient way. Mean time between failures will be reduced and codes will make use of a great number of resources. So checkpoint/restart is a must.

In this work, we have presented the design, implementation, and further integration between a user-level checkpoint/restart library and a resource manager. By making this integration transparent and automatically available for all jobs, a whole new set of possibilities have been enabled. In this way, not only is fault tolerance enhanced, but also the scheduling mechanisms, dynamic migration of tasks, job preemption, easier cluster administration, and so on, can be seamlessly performed from now on. Results on a real cluster are provided to demonstrate this.

This demonstrates possibilities for a more efficient and flexible use of supercomputers in which these new algorithms can be defined to improve the efficiency of the platform, reduce their energy consumption, and determine a trade-off between efficiency and reduced energy.

Related to the checkpoint library, DMTCP is nowadays only able to save the state of standard CPUs, though there is a working version for Nvidia CUDA that has provided their first successful results [43]. Such a fact will deeply enhance the impact

of this work. With respect to Xeon Phi, it seems that it is going to be decommissioned, but from the C/R operational point of view, it is like a standard CPU as KNL places the accelerator directly on the motherboard.

Regarding the migration of software containers inside HPC clusters, checkpointing of Docker is not supported by DMTCP at this time. Until this support is added, jobs employing such technologies are just marked as not-checkpointable and not considered by the migration policies and tools. The integration of these accelerators and/or containers as resources supported by DMTCP will, in the future, enhance the impact of the work presented here. Finally, it should be reiterated that the solution showed in this work is not tied to DMTCP and another checkpoint library with the right properties could be integrated into the tool described here, while continuing to support the same functionality.

Most importantly, future scheduling algorithms can benefit from job migration. But there is still scarce literature on sophisticated scheduling algorithms that have been tested at scale and in practice. In part, this is because of the previous lack of a robust mechanism for checkpoint-based job migration in common usage. The scheduling algorithms presented in this work serve as a test case and a demonstration of our approach, but are still limited in their functionality and performance. This opens up the future possibility of the use of artificial intelligence both for pure scheduling design and for new resilience strategies in which actual workload traces will be analyzed instead of random generated workloads in order to provide tailored and customized analysis to specific supercomputers. Approaches using simulation [44] and forecasting methodologies can also be considered here [45].

**Acknowledgements** This work was partially funded by the Spanish State Research Agency projects CODEC2 (TIN2015-63562-R) and CODEC-OSE (RTI2018-096006-B-I00) with FEDER funds and the EU H2020 Project Enerxico (Grant Agreement No 828947) and supported by the RICAP Network (517RT0529) with CYTED funds.

## References

1. Flich J et al. (2017) MANGO: exploring manycore architectures for next-generation HPC systems. In: Kubatova H, Novotny M, Skavhaug A (eds) Euromicro Conferences on Digital System Design (DSD), pp 478–485
2. Wyngaard J, Inggs M, Collins J, Farrimond B (2013) Towards a many-core architecture for HPC. In: Cardoso JMP, Morrow K, Diniz PC (eds) 23rd International Conference on Field Programmable Logic and Applications (FPL2013)
3. European technology platform for high performance computing (2017). [www.etp4hpc.eu](http://www.etp4hpc.eu), Strategic Research Agenda
4. Bailey C, Parry J (2017) Co-design, modelling and simulation challenges: from components to systems. In: Proceedings 23rd International Workshop on Thermal Investigations of ICs and Systems (THERMINIC), pp 1–4
5. Hill MD, Marty MR (2017) Retrospective on Amdahl's law in the multicore Era. *Computer* 50(6):12–14
6. Martineau M, McIntosh-Smith S (2017) The arch project: physics mini-apps for algorithmic exploration and evaluating programming environments on HPC architectures. In: Proceedings IEEE International Conference on Cluster Computing (CLUSTER2017), pp 850–857
7. Aupy G et al (2016) Co-scheduling algorithms for high-throughput workload execution. *J Sched* 19(6):627–640

8. Rajan M, Doerfler D (2010) HPC application performance and scaling: understanding trends and future challenges with application benchmarks on past, present and future tri-lab computing systems. In: Psihoyios G, Tsitouras C (eds) *Numerical Analysis and Applied Mathematics*, vol I–III (AIP Conference Proceedings 1281), pp 1777–1780
9. Yoo AB, Jette MA, Grondona M (2003) SLURM: simple linux utility for resource management. In: Feitelson D, Rudolph L, Schwiegelshohn U (eds) *Job scheduling strategies for parallel processing (JSSPP 2003)*, vol 2862. Lecture Notes in Computer Science. Springer, Berlin
10. Ansel J, Arya K, Cooperman G (2009) DMTCP: transparent checkpointing for cluster computations and the desktop. In: *IEEE International Symposium on Parallel & Distributed Processing, Rome*, pp 1–12
11. Tao J, Kolodziej J, Ranjan R, Jayaraman PP, Buyya R (2015) A note on new trends in data-aware scheduling and resource provisioning in modern HPC system. *Future Gener Comput Syst* 51:45–46
12. Ge X, Jin H, Leung VCM (2018) Joint opportunistic user scheduling and power allocation: throughput optimisation and fair resource sharing. *IET Commun* 12(5):634–640
13. Padua D (2011) *Encyclopedia of parallel computing*. Springer, New York
14. Hargrove PH, Duell JC (2006) Berkeley lab checkpoint/restart (BLCR) for linux clusters. *J Phys Conf Ser* 46:494–499
15. Sankaran S et al (2005) The Lam/Mpi checkpoint/restart framework: system-initiated checkpointing. *Int J High Perform Comput Appl* 19(4):479–493
16. Cao J, Arya K, Garg R, Matott S, Panda DK, Subramoni H, Vienne J, Cooperman G (2016) System-level scalable checkpoint-restart for petascale computing. In: *Proceedings IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*, pp 932–941
17. [https://criu.org/Main\\_Page](https://criu.org/Main_Page)
18. Li W, Kanso A, Gherbi A (2015) Leveraging linux containers to achieve high availability for cloud services. In: *Proceedings IEEE International Conference on Cloud Engineering, Tempe, AZ*, pp 76–83
19. Vogt D, Giuffrida C, Bos H, Tanenbaum AS (2015) Lightweight memory checkpointing. In: *Proceedings 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Rio de Janeiro*, pp 474–484
20. Takizawa H, Amrizal MA, Komatsu K, Egawa R (2017) An application-level incremental checkpointing mechanism with automatic parameter tuning. In: *5th International Symposium on Computing and Networking (CANDAR), Aomori*, pp 389–394
21. Ferreira KB, Riesen R, Bridges P, Arnold D, Brightwell R (2014) Accelerating incremental checkpointing for extreme-scale computing. *Future Gener Comput Syst* 30:66–77
22. Moody A, Bronevetsky G, Mohror K, Supinski BR (2010) Design, modeling, and evaluation of a scalable multi-level checkpointing system. In: *Proceedings ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pp 1–11
23. Bautista-Gomez L, Tsuboi S, Komatitsch D, Cappello F, Maruyama N, Matsuoka S (2011) FTI: high performance fault tolerance interface for hybrid systems. In: *Proceedings International Conference for High Performance Computing, Networking, Storage and Analysis, Seattle, WA*, pp 1–12
24. Tiemeyer MP, Wong JSK (1998) A task migration algorithm for heterogeneous distributed computing systems. *J Syst Softw* 41(3):175–188
25. Tsakalozos K, Verroios V, Roussopoulos M, Delis A (2017) Live VM migration under time-constraints in share-nothing IaaS-clouds. *IEEE Trans Parallel Distrib Syst* 28(8):2285–2298
26. Jaswal T, Kaur K (2016) An enhanced hybrid approach for reducing downtime, cost and power consumption of live VM migration. In: *Proceedings International Conference on Advances in Information Communication Technology & Computing*, vol 72
27. Bargi A, Sarbazi-Azad H (2011) Task migration in three-dimensional meshes. *J Supercomput* 56(3):328–352
28. Kale LV, Krishnan S (1993) CHARM++: a portable concurrent object oriented system based on C++. In: *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp 91–108
29. Iserte S, Mayo R, Quintana-Ortí SE, Beltran V, Peña JA (2017) Efficient scalable computing through flexible applications and adaptive workloads. In: *46th International Conference on Parallel Processing Workshops (ICPPW)*, pp 180–189
30. Losada N, Martín MJ, González P (2017) *J Supercomput* 73:316–329
31. Losada N, Cores I, Martín MJ et al (2017) *J Supercomput* 73:100
32. <http://research.cs.wisc.edu/htcondor/>
33. Afsharpour S, Patology A, Fazeli M (2016) Performance/energy aware task migration algorithm for many-core chips. *Comput Digit Tech* 10:165–173



34. Holmbacka S et al (2014) A task migration mechanism for distributed many-core operating systems. *J Supercomput* 68(3):1141–1162
35. Sotomayor B, Montero RS, Llorente IM, Foster I (2009) Virtual infrastructure management in private and hybrid clouds. *IEEE Internet Comput* 13(5):14–22
36. Sefraoui O, Aissaoui M, Eleuldj M (2012) OpenStack: toward an open-source solution for cloud computing. *Int J Comput Appl* 55(3):38–42
37. Boucher R (2016) Cloning running services with docker and CRIU. In: *Docker Conference*
38. Villamayor J, Rexachs D, Luque E (2017) A fault tolerance manager with distributed coordinated checkpoints for automatic recovery. In: *International Conference on High Performance Computing & Simulation (HPCS)*, Genoa, pp 452–459
39. Cabello U, Rodriguez J, Meneses A, Mendoza S, Decouchant D (2014) Fault tolerance in heterogeneous multi-cluster systems through a task migration mechanism. In: *Proceedings 11th International Conference on Electrical Engineering, Computing Science and Automatic Control*
40. Pascual JA, Navaridas J, Miguel-Alonso J (2009) Effects of topology-aware allocation policies on scheduling performance. *Lect Notes Comput Sci* 5798:138–144
41. <http://rdgroups.ciemat.es/web/sci-track/intranet>
42. Feitelson DG (2015) *Workload modeling for computer systems performance evaluation*. Cambridge University Press, Cambridge
43. Garg R, Mohan A, Sullivan M, Cooperman G (2018) CRUM: checkpoint-restart support for CUDA's unified memory. In: *IEEE International Conference on Cluster Computing (CLUSTER)*, pp 302–313
44. Levy S, Topp B, Ferreira KB, Widener P, Arnold D, Hoefler T (2014) Using simulation to evaluate the performance of resilience strategies and process failures, SANDIA report, SAND2014-0688
45. Fernández-Anta A et al (2018) Competitive analysis of fundamental scheduling algorithms on a fault-prone machine and the impact of resource augmentation. *Future Gener Comput Syst* 78:245–256

**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.