DROM: Enabling Efficient and Effortless Malleability for Resource Managers

Marco D'Amico
Barcelona Supercomputing Center
Barcelona, Spain
marco.damico@bsc.es

Marta Garcia-Gasulla
Barcelona Supercomputing Center
Barcelona, Spain
marta.garcia@bsc.es

Víctor López
Barcelona Supercomputing Center
Barcelona, Spain
victor.lopez@bsc.es

Ana Jokanovic
Barcelona Supercomputing Center
Barcelona, Spain
ana.jokanovic@bsc.es

Raül Sirvent
Barcelona Supercomputing Center
Barcelona, Spain
raul.sirvent@bsc.es

Julita Corbalan
Universitat Politecnica de Catalunya
Barcelona, Spain
juli@ac.upc.edu

ABSTRACT

In the design of future HPC systems, research in resource management is showing an increasing interest in a more dynamic control of the available resources. It has been proven that enabling the jobs to change the number of computing resources at run time, i.e. their malleability, can significantly improve HPC system performance. However, job schedulers and applications typically do not support malleability due to the common belief that it introduces additional programming complexity and performance impact. This paper presents DROM, an interface that provides efficient malleability with no effort for program developers. The running application is enabled to adapt the number of threads to the number of assigned computing resources in a completely transparent way to the user through the integration of DROM with standard programming models, such as OpenMP/OmpSs, and MPI. We designed the APIs to be easily used by any programming model, application and job scheduler or resource manager. Our experimental results from two realistic use cases analysis, based on malleability by reducing the number of cores a job is using per node and jobs co-allocation, show the potential of DROM for improving the performance of HPC systems. In particular, the workload of two MPI+OpenMP neuro-simulators are tested, reporting improvement in system metrics, such as total run time and average response time, up to 8% and 48%, respectively.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**;

1 INTRODUCTION

In High Performance Computing (HPC) systems the software stack consists of different layers, from parallel runtime to the workload manager, each one being responsible for a specific task. Application developers focusing on the individual performance of their applications use different programming models. This approach is a must to hide low-level architectural details from the application developers and users and extract the maximum performance of new systems.

On the other hand, the objective of the workload manager is to maximize the efficient utilization of the computing resources. However, improving the system efficiency is, typically, not well accepted by users and application developers, since their only objective is to speed up their application even if some of the resources are left underutilized. We claim that these two objectives must coexist and that cooperation between the different stack layers is the way to reach this goal.

We propose to provide resource managers with more tools that will give them a dynamic control of resources allocated to the application and a particular feedback about the utilization of these resources. In this paper we will extend the DLB [17] [18] library with a new API designed to be used by the resource managers. This new API will offer a transversal layer in the HPC software stack to coordinate the resource manager and the parallel runtime. We call this API *Dynamic Resource Ownership Management (DROM)*. DROM has been implemented as a part of DLB distribution and integrated with well know programming models, i.e. MPI [31], OpenMP [33] and OmpSs[12] and with the SLURM [8] node manager.

By integrating DROM with above programming models, the API will work transparently to the application, and thus, to developers. By integrating the API with SLURM, we enable efficient co-scheduling and co-allocation of jobs. This means that jobs are scheduled to share compute nodes by dynamically partitioning in an effective way the available resources, improving hardware utilization and job's response time.

This paper presents the following contributions:

- Definition of DROM, an API that allows cooperation between any job manager and any programming model.
- Integration of DROM with SLURM node manager for effective resources distribution in the case of co-allocation.
- Integration of DROM with MPI, OpenMP and OmpSs programming models.
- Evaluation of DROM with real use cases and applications motivated based on needs in the Human Brain Project (HBP) [35].

The rest of the paper is organized as follows: Section 2 presents the related work, Section 3 describes the DROM API, Sections 4 and 5 present the DROM integration with programming models and SLURM, Section 6 shows the experiments done to validate the integration and demonstrate the potential of this proposal, and finally Section 7 presents the conclusions and future work.

2 RELATED WORK

Malleable Parallel Task Scheduling (MPTS) problem has been explored for many years. The theoretical research shows its potential benefits [27] [13] [32]. These works mainly pick the number of resources that best improves the performance of the parallel task based on a model of its performance given at schedule time. Feitelson [16] classify a malleable job as a job that can adapt to changes in the number of processors at run time. Deciding on resizing a job at run time is not an easy task for a scheduler, and it is still not fully supported by any standard programming model. However, job scheduling simulations [21] showed the potential benefits of malleability concerning response time.

Several studies propose malleability based on MPI [31], that allows, in different ways, to spawn new MPI processes at run time or use moldability and folding techniques [36]. These approaches are limited by the inherent program data partition between processes. Data partition and redistribution is application dependent, so it needs to be done by application's developers. Furthermore, data transfer among nodes has a high impact on performance, making malleability very costly, especially if using checkpoint and restart techniques. To limit the amount of extra code, for the users to have malleable applications the structure of MPI application is usually constrained, iterative applications using split/merge of MPI processes are used in [28], master/slave applications are needed in [11]. Martin et al. [29] try to automatize data redistribution, but only for vectors and matrices.

Recent work includes an effort on Charm++ [19] programming model to support malleability. Charm++ allows malleability for applications by implementing fine-grained threads encapsulated into Charm++ objects. This solution is not transparent to developers, i.e., they need to rewrite their applications using this programming model. Adaptive MPI [20] tries to solve this issue by virtualizing MPI processes into Charm++ objects, partially supporting MPI standard. Charm++ lacks a set of API that would allow communicating with the job scheduler, because malleability features were studied for load balancing purpose. There was an effort to implement a Charm++ to Torque [15] communication protocol to enable malleability, but they are not comparable with DROM because DROM gives generalized APIs that can serve to communicate with any job scheduler or programming model.

Castain et al. in [9] presented an extensive set of APIs, part of PMIx project, including job's expanding and shrinking features. It is an interesting attempt to create standardized APIs that can be used by applications to request more resources to the job scheduler. However, the main difference is that they are designed for evolving applications, different from malleable, because changes in resources is demanded by the application itself, not the resource manager.

Despite this tendency in the research, users still do not have simple and efficient tools, neither the support from job schedulers in production HPC machines that would allow them to exploit malleability. We propose DROM, an API that enables malleability of applications inside computing nodes, with a negligible overhead for developers and applications. We integrated DROM APIs with OpenMP [33] and OmpSs [12] programming models and SLURM [8] job scheduler. However, DROM is independent of them, and it can be integrated with any other programming models or job schedulers.

DROM manages computing resources by using CPUSETs, lightweight structures used at the operating system level, easy and fast to use and manipulate. A similar approach was presented by [14], based on dynamically changing the operating system CPUSETs for MPI processes, but in this case, there was no integration with the programming model. This approach is equivalent to oversubscription of resources, i.e., more than one process running in the same core, which in general has a negative impact on the applications' performance, as demonstrated in [26]. In our integration, we used OpenMP/OmpSs programming models to adapt the number of threads to the change in the number of computing resources. OpenMP and OmpSs use threads instead of processes, easier to create and destroy, more efficient, lighter than MPI processes. At the same time, we support hybrid MPI+OpenMP/OmpSs applications, which allow the expansion of DROM capabilities to multi-node environments.

3 DROM: DYNAMIC RESOURCE OWNERSHIP MANAGEMENT

DROM is a new module included in the DLB library; it offers a new API to change the computing resources assigned to a process at run time. This module provides a communication channel between an administrator process and other processes to adjust the number of threads accordingly.

In this section we will explain the structure of the DLB library briefly to understand how DROM is integrated into it, we will present the proposed DROM API, and we will detail how we have integrated it with SLURM.

3.1 DLB Framework

DLB is a dynamic library that aims at improving the performance of individual applications, and at the same time to maximize the utilization of the computational resources within a node.

The DLB Framework is transversal to the different layers of the HPC software stack, from the job scheduler to the operating system. The interaction with the different layers is always done through standard mechanisms such as PMPI [2], or OMPT [3] explained in more detail in Section 4. Thus, as a general rule, applications do not need to be modified or recompiled to be run with DLB as long as they use a supported programming model (MPI + OpenMP/OmpSs). Simply by pre-loading the library, these standard mechanisms can be used to intercept the calls to the programming models and modify the number of required resources as needed.

DLB was initially designed for the *Lend When Idle (LeWI)* module. This module acts as a dynamic load balancer for a single application that suffers from processes' load imbalance by adjusting the number of threads per process when needed. However, our claim is that in the HPC systems there is also a necessity to dynamically balance the load among multiple jobs' processes that are executed within the same reservation. In this way, the system can benefit from increased utilization, which would not be the case when asking for separate job submissions. For this reason, we propose the DROM API and offer an implementation of it within the DLB library.

In Figure 1 we can see the DROM module within the DLB Framework. DROM provides an API for external entities, such as a job scheduler, a resource manager, or a user, to re-assign the resources used by any application attached to DLB. Then, the DROM module running on each process will react and modify the computing resources allocated for the application. This procedure depends on the programming model, but in essence, it implies two steps. First,

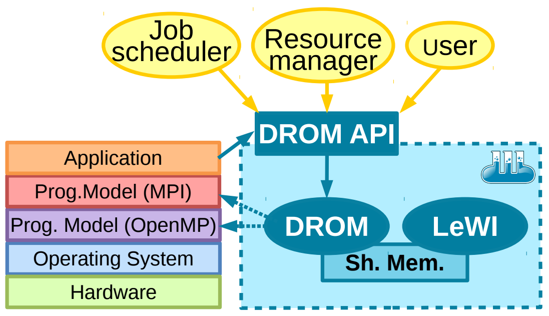

Figure 1: DLB Framework

the application will modify the number of active threads running within the shared memory programming model (OpenMP, OmpSs, etc.). Lastly, each active thread will be pinned to a specific CPU to avoid any oversubscription during the coexistence of the many processes in the node.

DLB uses node's shared memory to communicate the different processes, implemented as a common, lock protected, address space where all processes attached to DLB can read and write. While the communication can be asynchronous for the *sender*, the *receiver*, by default, will use a polling mechanism based on the interception interfaces. This mechanism produces a negligible overhead but relies exclusively on the frequency of the programming model invocation. Alternatively, DLB also implements an asynchronous mode for the receiver using a helper thread and a callback system.

This design of the framework allows user, or developer, to add DLB support to an application with minimal effort. However, there are some considerations to take into account before running an application with DLB support, i.e. how the application reacts to an unintended change of the number of running threads, and whether other hardware resources, apart from CPUs, can perform when other applications are co-allocated. The former only depends on the application implementation, and the latter may depend on several factors such as total memory consumed, I/O bandwidth, etc.:

- *Application's inherent non-malleability.* DLB may change the number of active threads, or *max_threads* in OpenMP nomenclature, at any time during the execution. For this reason, the application should be malleable. We consider an application completely malleable when its design allows changing the number of threads at any time, and its completion is still valid and successful. This condition requires a thread based programming model with some level of malleability, although its effect does not need to be immediate. For example, OpenMP is not able to modify the number of threads until the next parallel construct, but we consider it acceptable.
  An application is not malleable when, at any arbitrary point of the execution, obtains the number of threads and assumes it will not change in the future. For instance, a common practice in OpenMP applications is to allocate some auxiliary memory based on the current number of threads. At a later time, inside a parallel region, this memory will be indexed by the current thread identification number and may cause different errors depending whether the team size is smaller or larger than the assumed by the application. The suggested alternative is to exploit the features that the programming model already provides. For instance, a private array where its scope is limited to the parallel construct or a reduction clause where the programming model manages the auxiliary memory are both two solutions that solve this issue and keep the application malleable.
- *Hardware is finite.* During the job co-allocation, DLB may reduce the number of active threads of other processes and may rearrange the pinning of each thread to a new CPU, but it will not reduce the amount of allocated memory of any application. Therefore, the total memory capacity and bandwidth will be shared among applications.

3.2 DROM API for managing the co-allocation of applications

Processes attached to the DLB system can be managed from another process, referred as administrator process from now on. In this paper, we consider SLURM as the main candidate for the administrator process, but the implementation of the interface presented in this section allows users to program their own administrator process. In this case, the administrator process always runs as the same user doing the submission and the co-allocations are always limited to other applications of the same user.

The administrator process can manage other processes by communicating with the DLB system, which mainly consists of a single shared memory per node. Therefore, if the submission allocates more than one node, one administrator process must be created for each node that requires management, and eventual synchronization need to be implemented within those processes.

The proposed DROM interface is presented below, and its code is Open Source and available at [6]:

```
int DROM_Attach(void)
```
Attach current process to the system as DROM administrator. Once attached, the process is able to query or modify the process mask of other processes running with DROM support.

```
int DROM_Detach(void)
```
Detach current process from DROM system. If previously attached, a process must call this function to correctly close file descriptors and clean data.

```
int DROM_GetPidList(int *pidlist, int *nelems,
    int max_len)
```
Obtain the list of running processes registered in the DROM system.

```
int DROM_GetProcessMask(int pid, dlb_cpu_set_t
    mask, dlb_drom_flags_t flags)
int DROM_SetProcessMask(int pid,
    const_dlb_cpu_set_t mask, dlb_drom_flags_t
    flags)
```
Getter and Setter of the process mask for a given PID.

```
int DROM_PreInit(int pid, const_dlb_cpu_set_t
    mask, dlb_drom_flags_t flags, char ***
    next_environ)
```

Preinitialize a starting process into the DROM system, reserving some CPUs or making room in the node by shrinking other running processes according to *mask*. The usual workflow for this function is to register the current PID, then fork and exec into the new process keeping *next_environ* variable that permits the child process to be able to register using the parent's process ID.

```
int DROM_PostFinalize(int pid, dlb_drom_flags_t
    flags)
```

Finalize a previously preinitialized process. This function should be called after a preinitialized child process has finished its execution. The child process may have cleaned the shared memory if runs a supported programming model but this is not known from the job scheduler perspective. Is is always recommended to call this function to clean the data.

Non-standard C types used in this interface are:

- `dlb_cpu_set_t` is actually a *void pointer* provided as an opaque type and it is casted back internally to `cpu_set_t`. This data set is a bitset where each bit represents a CPU. It is defined in the GNU C library [7].
- `dlb_drom_flags_t` is a custom bitset provided by DLB. This argument adds some flexibility to the interface by allowing some options like: whether the function call is synchronous or asynchronous, whether to steal the CPUs from other processes, etc.

4 INTEGRATION OF DROM WITH PROGRAMMING MODELS

As previously explained, DLB applications need a shared memory programming model to modify the number of running threads, thus achieving the resource co-allocation. Currently supported thread based programming models are OpenMP and OmpSs. MPI interception is also supported by DLB to add more synchronization points between the application and DLB, as well as to gather more information about the application structure and improve the resource scheduling policies.

4.1 Integration with OpenMP

Any OpenMP application can use the DLB library without having to be recompiled as long as the OpenMP runtime used supports OMPT. OMPT is a new interface introduced in the OpenMP Technical Report 4 [3] and it will probably be included in the next OpenMP 5.0 specification. The interface allows external tools to monitor the execution of an OpenMP program. Even though the interface is not yet part of the OpenMP standard, several OpenMP runtimes already include it in their latests versions, such as the proprietary branch of Intel (2018.2.046) and their open-source branch based on LLVM's runtime [1].

If the OpenMP runtime implements this interface, DLB can register itself as a monitoring tool when the library is loaded. Then, DLB can set callbacks that will be automatically invoked for each parallel construct and implicit task creation allowing to modify the number of resources accordingly.

4.2 Integration with OmpSs

OmpSs is a task based programming model developed also at BSC, forerunner of many features accepted in the OpenMP specification. The OmpSs runtime includes DLB support, and if enabled, any compiled application can enable the DLB features provided by the runtime by setting the appropriate option.

4.3 Integration with MPI

HPC applications often request several computing nodes, thus, shared memory programming models are not enough. A message passing interface is required for the communication among the different processes of the application and the MPI standard is probably the most used for that purpose. Being aware of its importance, DLB implements an interception mechanism by using the MPI standard profiling interface, PMPI. PMPI allows any profiler, in this case DLB, to intercept any standard MPI call, and run custom code before and after the real MPI call.

DLB supports MPI interception and acts as an application profiler but it does not implement malleability at process level, i.e., MPI processes are never decreased or increased, nor any program data is ever moved between processes. For DROM purposes, MPI interception is only used to poll DLB and check if there are some pending actions to be taken. If the program runs with a new version of OpenMP implementing OMPT or with OmpSs, the MPI layer is completely optional.

4.4 Integration with applications without a supported programming model

DROM has been designed to be easily used even for applications that do not run a supported programming model. The DLB library includes an interface for applications in order to become DROM-responsive and react to the reallocations performed by the manager process. Using the DLB interface in the application implies that it has to be recompiled, but it also offers more flexibility to only call DLB on those *safe points* where the application can change the number of threads if it is not completely malleable.

Listing 1 shows an example of an iterative application manually modified to support DROM. The effort for developers is minimal. First, the application needs to initialize and finalize DLB correctly when appropriate. Then, just before entering the malleable parallel code, it should poll the DROM module to check if the resources need to be readjusted and, if needed, perform the necessary actions. This adjustment needs to be done by the application. In case of an OpenMP application, it may include a call to `omp_set_num_threads` and, optionally, a rebind of threads if the runtime is configured to bind them to CPUs.

```
#include "dlb.h"
int main(int argc, char **argv) {
    /* initialization */
    DLB_Init();
    ...
    /* main loop */
    for(i=0; i<end; i++) {
        if (DLB_PollDROM(&ncpus, &mask)
                == DLB_SUCESS) {
```

```
            modify_num_resources(ncpus, &mask);
        }
        #pragma omp parallel
        ...
    }
    /* Finalization */
    DLB_Finalize();
    ...
    return 0;
}
```

Listing 1: Iterative application manually invoking DROM

5 INTEGRATION OF DROM WITH SLURM

The resources of an HPC machine are managed by a job scheduler and a resource manager. These two pieces of software allow for fast and efficient exploitation of the systems' computing resources. To execute their applications on the part of the machine, users need to submit a job in which they specify the type and the amount of the resources they need, and the period of time during which they need these resources. All users requests are collected as jobs, and they are usually managed into a priority queue. One of the most used job schedulers in research, as well as in production systems is SLURM. It stands out for its efficiency and scalability, and because it is open source software.

DROM APIs were integrated into SLURM, to automatize the placement of jobs' tasks inside computing nodes, whenever one or more malleable jobs are scheduled inside the same nodes.

The following implementation only affects jobs placement inside nodes, i.e. selecting for each node on which CPUs job will run. *Slurmctld*, the cluster controller in charge of scheduling jobs and selecting on which compute nodes they will run, is unchanged, as the purpose is to give a proof of integration of DROM APIs, not to present new scheduling policies.

SLURM structure grants portability of the code by the use of plugins, dynamic libraries that allows system administrators to avoid recompiling the SLURM core. For this reason the implementation is enclosed in the SLURM's *task/affinity* plugin, in charge of distributing the resources assigned by *slurmctld* to the job's tasks.

Task/affinity is dynamically loaded by *slurmd* and *slurmstepd*, dividing the code flow in two parts. The first is done inside *slurmd*, in charge of managing single computing node resources, and thanks to the plugin, calculating and distributing CPU masks to tasks of the scheduled job. The second part is called by *slurmstepd*, a daemon that controls correct task launch and execution. At launch point, the plugin picks the mask assigned by *slurmd* and actually sets it.

In Figure 2 we give an example that clarifies the actions of DROM within SLURM. It illustrates the steps performed within DROM-enabled *slurmd* and *slurmstepd*. We present a scenario of two jobs starting to share a computing node. The *job 1*, to simplify the figure, is a one-task job already running in the *node 1*, while the *job 2* is a two-task job just submitted and given resources on both *node 1* and *node 2*. Initially, *job 1* uses all the resources of *node 1* until a part of them is taken by DROM and given to *job 2*.

On the left we have *job 1* running *task 1.1* into *node 1*, on the right the start procedure for *job 2*. Vertical axis represents time for

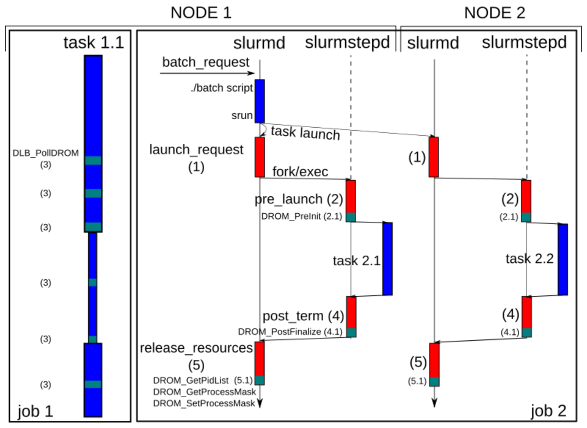

Figure 2: SLURM job launch procedure for DROM malleable applications in two computational nodes.

each involved component, red boxes are modified SLURM parts, blue boxes are unmodified parts, green boxes are DROM calls.

Starting from the top, *node 1*'s *slurmd* executes the submitted *batch script*, that uses *srun* to launch a parallel malleable application, i.e., *job 2*. *Srun* sends requests of launching the tasks to the two *slurmd* involved in *job 2* allocation. Both *slurmds* call *launch_request* (1) function, that calculates the CPU mask for the starting task.

In this part of the code, since job 1 is running in the node, our implementation calculates a new mask for both the new and the running job, where the mask of the running job is a subset of its original mask. In this case, CPUs distribution is done to maintain running and new processes balanced in the number of CPUs for each task, assuming that imbalance in hybrid MPI+OpenMP/OmpSs applications degrade performance. The algorithm also distributes CPUs trying to keep applications in separate sockets in order to improve data locality. In this scenario, for fairness, computational resources are equally partitioned among running jobs.

After calculating masks for both new and running tasks, *slurmd* forks and executes *slurmstepd*. *Slurmstepd* calls a *pre_launch* (2) function, in charge of setting the mask calculated by *slurmd* to the controlled task, and eventually update the other running task 1.1 mask, if necessary. This is done using *DROM_PreInit* (2.1) function. At next malleability point, when task 1.1 runs *DLB_PollDROM* (3), it gets a new CPU mask from shared memory and applies it, reducing the number of assigned CPUs per task. In Figure 2 we see the reduction in CPUs as shrinkage of the blue line. If job 1 runs on node 2, coordination is implicit in *slurmd*'s CPUs distribution, that gives the same placement for both nodes.

When a task ends *post_term* (4) is invoked, that involves a call to *DROM_PostFinalize* (4.1). This function can return CPUs to the job that is initial owner of the CPUs, i.e., *job 1*. Of course, this is only possible in the case this job is still running and keep calling *DLB_PollDROM* (3).

When a job completes, *slurmd* calls *release_resources* (5), that redistributes free CPUs to still running tasks. In the case the job owner of the CPUs, in this case *task 1.1*, completes before the job 2, CPUs will be acquired by the job 2, that will expand its mask to increase node utilization. This is done by using *DROM_GetPidList*, *DROM_GetProcessMask* and *DROM_SetProcessMask* (5.1) APIs.

6 EVALUATION OF DROM-ENABLED SYSTEM'S PERFORMANCE

To evaluate the potential and utility of the DROM API we perform two types of experiments that follow two realistic use case scenarios, supported by HBP:

(1) *In-Situ Analytics.* The workload consists of two jobs: 1) a big and long job that we will refer to as *simulation* and 2) small and short job that we will refer to as *analytics.* This scenario corresponds to a use case of HBP in HPC machines, where a neuro-simulation is running, and a visualizer or a data analytics program can periodically check partial simulation results, instead of waiting simulation to complete. To run an analytics application, within a standard system, the user would launch a second job asking for resources and wait until they are available. Using DROM, the analytics would use part of the resources allocated to the simulation, by temporarily shrinking its the number of used resources. This permits running analytics in the same node, avoiding reading and writing data to disk in case the analytics is able to exchange data with the simulation in-memory, or data transfer in case the analytics runs on a local machine.
(2) *High-priority job.* In the second use case, also part of HBP use cases, we consider the scenario of two jobs: 1) a long-running simulation and 2) a new high-priority long-running job, e.g., an interactive job or urgent simulation, arriving in the queue. In the absence of available resources, the high-priority job needs to wait in the job queue, or the already running job needs to be preempted or oversubscribed, which would degrade the performance, as previously explained in Section 2. With other malleability implementations, the simulation would need to shrink in the number of nodes, creating overhead due to data movement and checkpoint/restart operations. In DROM case, the application can keep executing on the same number of nodes, but on a reduced number of resources per node, while the high-priority job is scheduled to run in the same job allocation.

We use a set of real applications - two neuro-simulation applications and two synthetic benchmarks:

- NEST [24] is a simulator for spiking neural network models. It is parallelized with MPI and OpenMP. We have modified the code of NEST, based on version 2.12.0, to make it malleable[5]. Additionally, we have added calls to `poll_DROM` in the safe points where the number of threads can be changed.
- CoreNeuron is a simulator for modeling neurons and networks [23]. It is parallelized with MPI and OpenMP. We have modified the code to add calls to `poll_DROM` in safe points for malleability[4].
- Pils[18] is a synthetic benchmark, doing computation-intensive operations. It is parallelized with MPI + OmpSs. It can be configured to run with different numbers of MPI processes and OpenMP/OmpSs threads. In our experiments, we use it to simulate a compute bound parallel data analytics.
- STREAM is a benchmark intended to measure sustainable memory bandwidth[30]. The used dataset size can be adjusted, we configured it to run multiple iterations with an 8GB dataset. The application is parallelized with MPI + OpenMP. We used this benchmark to simulate a memory bound analytics software.

Pils and STREAM benchmarks are used to reproduce the behavior of in-situ visualizers and analytics used in HBP project, at this point still at an early stage.

All the experiments are real-machine workload runs. For that purpose, we used MareNostrum III (MN3) supercomputer [10], based on Intel SandyBridge processors, with each node containing two sockets with eight cores per socket and 128 GB of DDR3 memory. The operating system is a SLES distribution, with Platform LSF [22] resource manager. NEST and CoreNeuron were compiled using Intel 2017.1 compilers and OpenMPI libraries version 1.10, Pils and STREAM were compiled with Mercurium 2.0.0 and Nanos 0.13a. We run the experiments using the original SLURM based on version 15.08.11 and the modified version that uses DROM to exploit malleability as described in Section 5. To run the modified SLURM version, we created an environment where we can launch the real SLURM as an LSF job on a portion of 3 nodes of the production-machine, one for controller, two for computing nodes. This environment allows us to run SLURM as a regular job scheduler for real-application jobs submitted to it and being freely configurable by us.

All the reported results are an average of at least 3 runs performed in two MN3 nodes, we observed a maximum coefficient of variation of 3.4% in run time measurements. We analyzed the use cases from a system and application perspective, by measuring:

- Total run time: time to complete the workload, calculated as last job end time minus first job submission time.
- Response time: calculated as a sum of job's wait time in scheduler's queue and job's execution time.
- Average response time: arithmetic mean of response times of all the jobs in the workload.
- IPC: number of instructions completed per processor cycle by a specific thread.
- Cycles per microsecond: number of processor's cycles per microsecond dedicated to the specific thread.

We obtained system metrics from SLURM logs and application's metrics by tracing the use cases using Extrae [25] and visualizing traces with Paraver [34]. We compared the baseline and DROM enabled implementation by measuring run time on two exclusive MN3 nodes. We didn't find any visible overhead between them, so we can compare the two versions in our experiments.

Each of the use cases is evaluated for several different configurations regarding the number of MPI processes and OpenMP threads per MPI process, as summarized in Table 1. All applications ask for 2 nodes and distribute MPI processes among them. We run NEST and CoreNeuron with different configurations and we observed increasing IPC switching from *Conf. 1* to *Conf. 2.* This is due to a different data access pattern and better data locality. We kept both configurations to check how the use cases perform. Regarding Pils, in *Conf. 2* and *Conf. 3* it does request and run only on a part of node resources, even if the node is free. Even though it is supposed to be

Application	Conf. 1: MPI x OpenMP	Conf. 2: MPI x OpenMP	Conf. 3: MPI x OpenMP
NEST	2 x 16	4 x 8	-
CoreNeuron	2 x 16	4 x 8	-
Pils	2 x 16	2 x 1	2 x 4
STREAM	2 x 2	-	-

Table 1: Use cases applications configurations.

a small application, we run Pils in *Conf. 1* to have a reference case in which nodes are fully utilized, as a further case for comparison. Concerning STREAM, we don't need to change configuration for it as the application is memory bound and over two CPUs per node performance keeps constant. We will refer to the different configurations as *App-name Conf. x*, e.g. *NEST Conf. 1* means NEST with 2 MPI processes and 16 OpenMP threads per process.

6.1 Use Case 1: In Situ Analytics

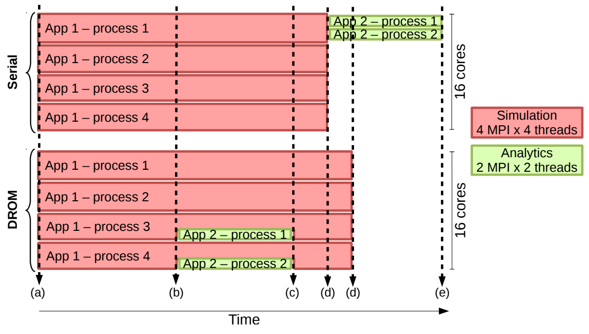

Figure 3: In situ analytics example.

In Figure 3 we can see a graphical representation of this use case. The horizontal axis represents time while the vertical axis computational resources, i.e., number of cores. At time *(a)* the simulation is launched and started in the available resources. At time *(b)* the analytics is submitted.

We compare two scenarios, the *Serial* one considers that the analytics must wait for the simulation to finish before it can start at point *(d)* because no resources are available. Thus the simulation runs using all the available cores and only when it finishes the analytic is executed. The second scenario is using *DROM* to start analytics immediately at time *(b)*, reducing the number of resources assigned to the simulation. Once the analytics finishes at point *(c)* the simulation gets its resources back.

We evaluated the following two-applications workloads. We will use the notation *simulation application+analytics application* when naming the workloads, i.e. NEST + Pils, NEST + STREAM, CoreNeuron + Pils, CoreNeuron + STREAM. For this use case, we evaluate and analyze the total run time, average response time, individual application's response time, followed by a discussion on the differences between Serial and DROM scenarios and the various configurations.

Figure 4 shows the difference in the workload's total run time when running the two versions of NEST in one node with Pils. Y-axis represents total run time in seconds, while on X-axis we have the different configurations of the applications involved. For both

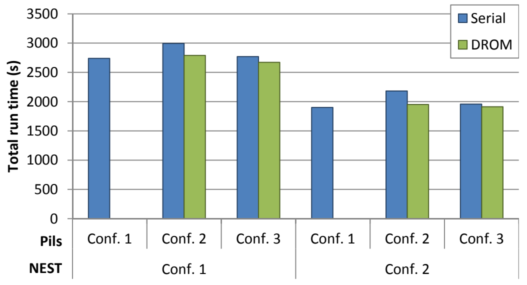

Figure 4: Run time of NEST + Pils workload. Y-axis represents total run time in seconds, X-axis shows the different configurations of the applications.

NEST configurations, run time for DROM case is in average 5.9% better than Serial case for *Pils Conf 2* and *Conf 3*, and comparable to the reference case *Conf. 1*. Average overhead of DROM scenario over *Pils Conf. 1* is 0.6%, varying with the analyzer's configuration. We observed that this is because of NEST implementation. Since its data is statically partitioned according to the maximum number of computational resources during initialization, as explained in Section 3, when applying malleability to shrink NEST, the tasks not computed by the removed thread are computed by some of the remaining resources, creating imbalance, as shown in Figure 5. This is a limitation of the application and not an overhead introduced by DROM. In fact, increasing the number of stolen computing resources, like in the case of *Pils Conf. 3*, the number of excess tasks increases, and they are better distributed among the remaining resources. In this situation, we improve total run time up to 2.5% with respect to *Pils Conf. 1*, while for *Conf. 2* it can reach -2.6%. A fully malleable NEST version that doesn't partition data according to initial number of threads would improve this result.

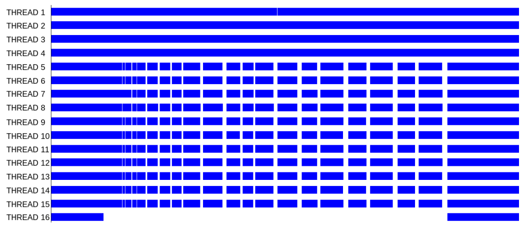

Figure 5: Trace showing simulator's threads on Y-axis. When thread 16 is removed, its data is computed by first 4 threads, while the others report lower utilization (white idle spaces).

Figure 6 shows single application's response time. Pils's response time, painted in lines pattern, decreases up to 96% due to waiting time reduced to zero, while its run time is approximately the same. The cost for this reduction is a small increase in NEST's response time, varying from 0% to 4.2%. A 0% increase is due to increasing IPC when running on a reduced number of threads.

In Figure 7 we can see workload's total run time and each application's response time for NEST and STREAM use case. In this case we remove 2 CPUs from the simulation to run a memory intensive application, gaining in average 1.84% (up to 3.5%) in terms of total run time. STREAM's response time decreases up to 92% while

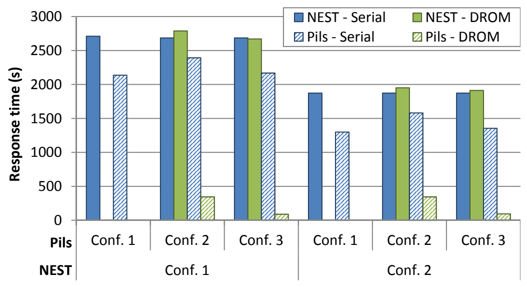

Figure 6: Individual response time of NEST and Pils in the NEST + Pils workload.

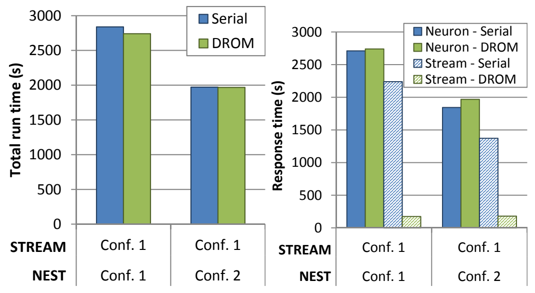

Figure 7: Run time (Left) and Response time (Right) of NEST + STREAM workload varying NEST configuration.

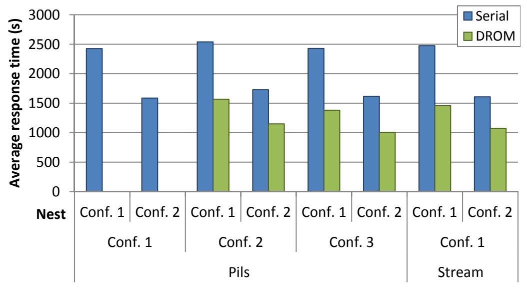

Figure 8: Average response time of NEST workloads.

NEST's increases up to 6.7% in the worst case. Total run time is always better because of benefits of memory bound and a compute bound applications sharing the nodes.

Figure 8 shows average response time. Gain in DROM case is up to 48% and never less than 37% with respect to the Serial case.

Figures 9, 10, 11 and 12 show the same set of experiments but with CoreNeuron neuro-simulator. Results are very similar to NEST workloads, as also CoreNeuron presents the same data partition problem. Figure 9 shows improved run time when comparing with *Pils Conf. 2* and *Conf. 3*, and a maximum overhead of 5% compared to *Pils Conf. 1*. Compared to NEST, CoreNeuron shows slightly worse results when sharing with compute intensive analytics like Pils, even if less affected by the number of requested resources, showing 2% of variation versus 5% of NEST. In Figure 11 total run time is always better then the Serial cases for STREAM workloads (up to 8%), response time decreases up to 91% while CoreNeuron's increase is 4% in the worst case. Compared to NEST, it slightly performs better when sharing the node with memory intensive applications like STREAM, with an average run time gain of 5.3% vs 1.84% for NEST. Average response time in Figure 12 shows an average gain of 46.5% for DROM scenario with respect to the Serial.

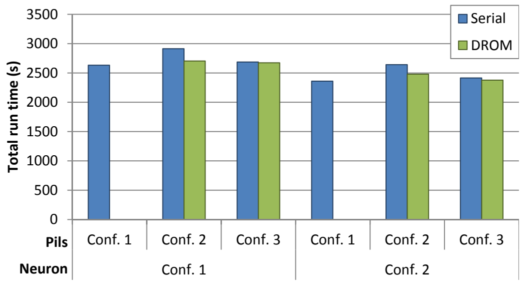

Figure 9: Execution time of CoreNeuron + Pils workload.

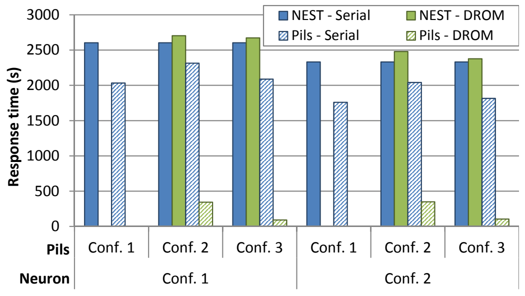

Figure 10: Individual response time of CoreNeuron and Pils in the CoreNeuron + Pils workload.

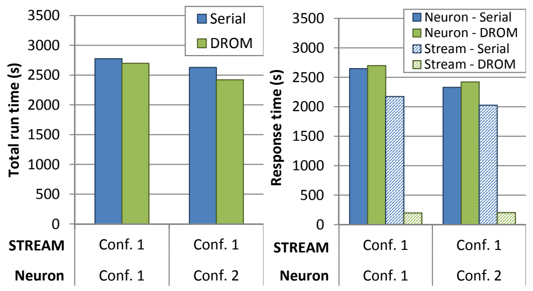

Figure 11: Execution time (Left) and Response time (Right) of CoreNeuron + STREAM workload varying CoreNeuron configuration.

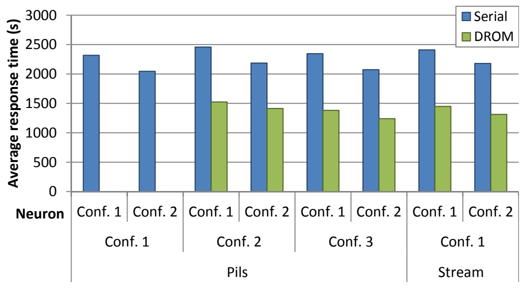

Figure 12: Average response time of CoreNeuron workloads.

6.2 Use Case 2: High-priority job

In this use case we analyze a single workload made up of two jobs, a long NEST and a long CoreNeuron simulation running on 2 MN3 nodes. Both jobs request *Conf. 1* presented in Table 1.

Again, we compare a *Serial* scenario in which the high-priority job can only start after the running job ends, and *DROM* scenario where the same job starts immediately by freeing some resources using DROM interface.

Figure 13 presents traces for both scenarios. X-axes represent time, with same scale to compare total run time, while Y-axes show application's threads. At time *a)* NEST is submitted and runs on the entire two nodes allocation. At time *b)* CoreNeuron is submitted. In the top trace, representing the Serial scenario, CoreNeuron needs to wait for all the resources to be freed to start, starting at time *c)*. The bottom trace represents the DROM scenario, in which CoreNeuron starts at submission time, sharing nodes with NEST. At time *d)* NEST ends, freeing half of the available resources, and CoreNeuron expands its allocation to keep maximum nodes utilization. In the DROM scenario, as both applications ask for two entire nodes, SLURM will apply the implemented automatic resource partition by reducing both new and running jobs used resources. Equipartition is applied, giving 16 CPUs per application on a total of 32.

We present total run time and response time to discuss about system benefits of malleability for this use case, and application related performance counters, like IPC and cycles per µs, to demonstrate applications are not interfering each other when DROM is used for malleability.

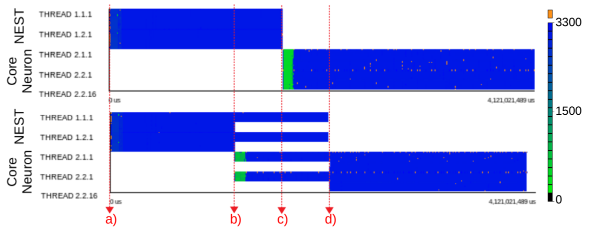

Figure 13: Traces showing cycles per µs of use case 2. Serial scenario is presented on the top, DROM on the bottom.

Looking at the total workload duration in Figure 13, in the case of DROM, better resource utilization leads to a total run time improvement of the 2.5%. The same figure shows the cycles per µs using colors. Showing the same color for both scenarios means there is no difference between Serial and DROM scenarios and constant color during run time shows that there is no variation in this metric when applying malleability to expand and shrink applications. Green color at beginning of CoreNeuron simulator shows lower cycles in memory intensive initialization phase. In the Serial scenario, during initialization, all computational resources are underutilized, while in DROM case, NEST keep running, increasing utilization and contributing to reduce total workload run time.

Figure 14 shows the number of instruction per cycle for both configuration of the use case 2. Figures are grouped by application to be easily comparable. X-axes represent IPC in increasing order, Y-axes application's threads, blue dots show more frequent IPC and

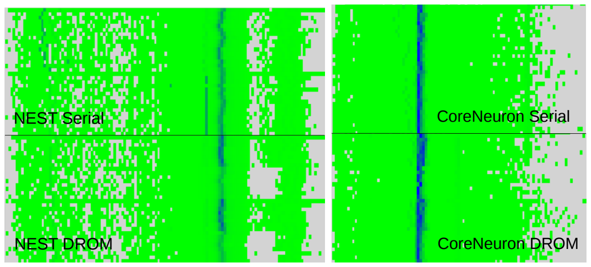

Figure 14: Histogram of instruction per cycle for CoreNeuron and NEST runing in serial and with DROM.

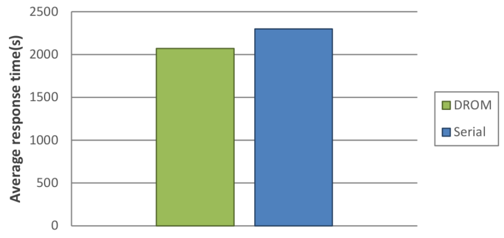

Figure 15: Average response time for use case 2 workload. DROM scenario improves response time by 10% with respect to the Serial scenario.

represent the main information of the histograms. They demonstrate that Serial and DROM scenario are comparable in terms of IPC. Regarding NEST, we distinguish some noise in the Serial scenario, and two color variants for the most frequent IPC for DROM. This is due to the fact that threads corresponding to the lighter color are removed to accommodate CoreNeuron at time (b) of Figure 13, distributing more computation on the darker part of the graph. For CoreNeuron, IPC in Serial scenario is constant but for DROM, we can distinguish two blue zones, in correspondence to the threads in which application starts, reporting slightly higher IPC. This is due to higher parallel efficiency when running on less number of OpenMP threads per MPI rank, improving total run time.

Finally, Figure 15 presents average response time for this use case. Response time improves by 10% with respect to the Serial scenario due to gain in run time and because the high-priority job can start earlier, improving at the same time user experience when the job is interactive or giving earlier partial results when a simulation is able to start earlier.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we presented an interface, named DROM, that allows exploiting malleability by creating communication between two, at the moment, unconnected parts of HPC software stack that should work in a more connected and coordinated way. The presented API permits to change the computational resources allocated to a running application efficiently, without any overhead.

We implemented the proposed APIs within the DLB framework, designing the interface to be easily integrated into any programming model or directly into the application. We integrated them with different and widely used programming models, such as MPI and OpenMP. Additionally, we presented an integration of the API with SLURM node manager, to achieve automatic distribution and placement of co-scheduled jobs inside nodes. We presented two use cases as a proof of concept, and we analyzed results from the workload point of view and application point of view. Our results show up to 48% improvement in average response time, and up to 8% in total run time, by comparing to the serial case.

With this study, we open future work in two directions. On one side we want to expand the potential of DROM, with new functionalities, like the collection of useful data from applications at run time. The collected information can be consulted by an external to get info about applications performance and send them to the job scheduler to be taken into account for further scheduling decisions. On the other side, we want to tight the communication between the different layers of the HPC software stack, i.e., by developing DROM-aware scheduling and resource management policies. The simplicity of DROM APIs gives more freedom to the scheduler, that can implement malleable scheduling techniques, for instance by choosing one or multiple specific jobs to share computational nodes, or at resource management level, by choosing as "victim" nodes the ones with lower utilization. Combined with a job scheduler/resource manager, DROM can be used in many different ways, including implementing new scheduling policies based on malleability, e.g. policies based on co-scheduling, or as alternative to jobs preemption.

Acknowledgments. This work is partially supported by the Spanish Government through Programa Severo Ochoa (SEV-2015-0493), by the Spanish Ministry of Science and Technology through TIN2015-65316-P project, by the Generalitat de Catalunya (contract 2017-SGR-1414) and from the European Union's Horizon 2020 under grant agreement No 785907 (HBP SGA2).

REFERENCES

[1] 2016. Intel©OpenMP Runtime Library. (2016). https://www.openmprtl.org/
[2] 2016. PMPI profiling interface. (2016). https://www.open-mpi.org/faq/?category=perftools
[3] 2016. TR4: OpenMP Version 5.0 Preview 1. (2016). http://www.openmp.org/wp-content/uploads/openmp-tr4.pdf
[4] 2017. Malleable CoreNeuron source code. (2017). https://github.com/BlueBrain/CoreNeuron/tree/hbp_dlb
[5] 2017. Malleable NEST source code. (2017). https://github.com/mggasulla/nest-simulator/tree/malleability
[6] 2018. DLB-DROM source code. (2018). https://github.com/bsc-pm/dlb/
[7] 2018. The GNU C library: CPU Affinity. (2018). https://www.gnu.org/software/libc/manual/html_node/CPU-Affinity.html
[8] Yoo A. B., Jette M. A., and Grondona M. 2003. SLURM: Simple Linux Utility for Resource Management. In *Job Scheduling Strategies for Parallel Processing*. 44–60.
[9] R. H. Castain, D. Solt, J. Hursey, and A. Bouteiller. 2017. PMIx: Process Management for Exascale Environments. In *Proceedings of the 24th European MPI Users' Group Meeting (EuroMPI '17)*. ACM, New York, NY, USA.
[10] Barcelona Supercomputing Center. 2014. Marenostrum 3. (2014). https://www.bsc.es/marenostrum/marenostrum/mn3
[11] I. Comprés, A. Mo-Hellenbrand, M. Gerndt, and H. Bungartz. 2016. Infrastructure and API Extensions for Elastic Execution of MPI Applications. In *Proceedings of the 23rd European MPI Users' Group Meeting (EuroMPI 2016)*. 82–97.
[12] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. 2011. Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters* 21 (2011).
[13] J. Turek et al. 1994. Scheduling Parallelizable Tasks to Minimize Average Response Time. In *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*. 200–209.
[14] M. Cera et al. 2010. Supporting Malleability in Parallel Architectures with Dynamic CPUSETs Mapping and Dynamic MPI. In *Proceedings of the 11th International Conference on Distributed Computing and Networking*. 242–257.
[15] S. Prabhakaran et al. 2015. A Batch System with Efficient Adaptive Scheduling for Malleable and Evolving Applications. In *2015 IEEE International Parallel and Distributed Processing Symposium*. 429–438.
[16] Dror G. Feitelson and Larry Rudolph. 1996. Toward convergence in job schedulers for parallel supercomputers. In *Job Scheduling Strategies for Parallel Processing*. Berlin, Heidelberg, 1–26.
[17] M. Garcia, J. Corbalan, and J. Labarta. 2009. LeWI: A Runtime Balancing Algorithm for Nested Parallelism. In *International Conference on Parallel Processing*.
[18] M. Garcia, J. Labarta, and J. Corbalan. 2014. Hints to improve automatic load balancing with LeWI for hybrid applications. *J. Parallel and Distrib. Comput.* (2014).
[19] Abhishek Gupta, Bilge Acun, Osman Sarood, and Laxmikant V. Kale. 2014. Towards Realizing the Potential of Malleable Parallel Jobs. In *Proceedings of the IEEE International Conference on High Performance Computing (HiPC '14)*. Goa, India.
[20] C. Huang, O. Lawlor, and L. V. Kalé. 2003. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing*.
[21] J. Hungershofer. 2004. On the combined scheduling of malleable and rigid jobs. In *16th Symposium on Computer Architecture and High Performance Computing*.
[22] IBM. 2014. Platform LSF. (2014). www.ibm.com/support/knowledgecenter/en/SSETD4_9.1.2/lsf_welcome.html
[23] P. Kumbhar and M. Hines. 2016. CoreNeuron Neuronal Network Simulator Optimization Opportunities and Early Experience. In *GPU Technology Conference*.
[24] Kunkel, S. et. al. 2017. NEST 2.12.0. (2017). https://doi.org/10.5281/zenodo.259534
[25] G. Llort, H. Servat, J. González, J. Giménez, and J. Labarta. 2013. On the usefulness of object tracking techniques in performance analysis. In *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
[26] V. Lopez, A. Jokanovic, M. D'Amico, M. Garcia, R. Sirvent, and J. Corbalan. 2017. DJSB: Dynamic Job Scheduling Benchmark. In *Job Scheduling Strategies for Parallel Processing: 21st International Workshop, JSSPP 2017, Orlando, FL, USA, June 2, 2017, Revised Selected Papers*.
[27] Walter Ludwig and Prasoon Tiwari. 1994. Scheduling Malleable and Nonmalleable Parallel Tasks.. In *SODA*, Vol. 94. 167–176.
[28] K. El Maghraoui, T. J. Desell, B. K. Szymanski, and C. A. Varela. 2007. Dynamic Malleability in Iterative MPI Applications. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*. 591–598.
[29] Gonzalo Martín, David E. Singh, Maria-Cristina Marinescu, and Jesús Carretero. 2015. Enhancing the Performance of Malleable MPI Applications by Using Performance-aware Dynamic Reconfiguration. *Parallel Comput.* 46 (July 2015).
[30] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *Technical Committee on Computer Architecture* (1995).
[31] Message Passing Interface Forum. 2015. MPI Specifications 3.1, http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf. (2015).
[32] G. Mounie, C. Rapine, and D. Trystram. 1999. Efficient Approximation Algorithms for Scheduling Malleable Tasks. In *Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '99)*. 23–32.
[33] OpenMP. 27/4/2018. OpenMP 4.5 Specifications, http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf. (27/4/2018).
[34] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. 1995. Paraver: A tool to visualize and analyze parallel code. In *Proceedings of WoTUG-18: transputer and occam developments*, Vol. 44. IOS Press, 17–31.
[35] Human Brain Project. 2017. https://www.humanbrainproject.eu/en/. (2017).
[36] G. Utrera, J. Corbalan, and J. Labarta. 2004. Implementing Malleability on MPI Jobs. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*.