H2020-JTI-EuroHPC-2019-1

Project no. 956748

ADAPTIVE MULTI-TIER INTELLIGENT DATA MANAGER FOR EXASCALE

D3.1
Malleability requirements definition.

Version 1.0

Date: September 30, 2021

Type: Deliverable
WP number: WP3

Editor: David Exposito-Singh
Institution: UC3M

Project co-funded by the European Union Horizon 2020 JTI-EuroHPC research and innovation programme and Spain, Germany, France, Italy, Poland, and Sweden		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Change Log

Rev.	Date	Who	Site	What
1	21/04/21	Jesus Carretero	UC3M	Document creation.
2	04/05/21	Hamid Fard	TUDA	Redesign of ADMIRE architecture.
3	11/09/21	Hamid Fard	TUDA	Executive summary.
4	12/09/21	Hamid Fard	TUDA	Malleability requirements and focus of WP3.
5	20/09/21	Ramon Nou	BSC	I/O Malleability.
6	20/09/21	Hamid Fard	TUDA	Contribution in the Introduction and background sections.
7	20/09/21	David E. Singh	UC3M	Intra/Inter node malleability.
8	21/09/21	Massimo Torquati	CINI	Safe reconfiguration state.
9	21/09/21	David E. Singh	UC3M	FlexMPI.
10	22/09/21	Taylan Özden	TUDA	Added Slurm section (Overview and Plugins).
11	22/09/21	Taylan Özden	TUDA	Finalized Slurm section (Malleability extension).
12	22/09/21	David E. Singh	UC3M	API description.
13	23/09/21	Massimo Torquati	CINI	Revised the introduction of ADMIRE architecture.
14	23/09/21	David E. Singh	UC3M	Updated images of ADMIRE architecture workflow and data flow.
15	24/09/21	Hamid Fard	TUDA	Conclusion.
16	24/09/21	Hamid Fard	TUDA	Reorganizing the structure, integrating the material and proofreading.
17	27/09/21	Aasem Ahmad	TUDA	Revised sections WP3 focus and intra/inter node malleability.
18	27/09/21	David E. Singh	UC3M	UML diagram.

Executive Summary

Data-intensive applications represent a growing share of the applications running on HPC systems. To achieve the best possible performance of such systems, we need a balance of compute and storage requirements for the applications running in the systems. By emerging the new I/O technologies, such as NVMe storage, we are still missing an extensive software stack that can harness the capacity of all available technologies in HPC ecosystem.

To this end, ADMIRE project focuses on producing such a framework consisting three main components: malleability manager, ad hoc parallel storage system and I/O scheduler. For malleability management, which is the task of WP3 in this project, we need to clearly explain (a) in which level malleability should be implemented in the project, (b) which requirements should be prepared for each level and (c) which APIs could achieve the role of malleability manager connecting other components of ADMIRE project. This document will summarize the answer to these critical questions.

Contents

Chapter 1

Introduction

Traditionally, increasing the parallel computing performance of HPC systems was the main concern of providers and users. But the ever increasing data processing needs of newer applications such as machine learning are pushing all stakeholders to revise their view of the HPC field.

Balancing the compute and I/O requirements of the applications could lead to a more efficient utilization the current HPC resources and achieve better performance. In line with this approach, we initiated the ADMIRE project to provide a software stack combining computation and data resource malleability.

Consequently, since the beginning of the project proposal, malleability management was considered a vital component of the whole system. The goal is to provide base mechanisms for the combined malleability of compute and I/O resources. Those mechanisms will be guided by new scheduling algorithms and policies, integrated into plugins for the Slurm batch scheduler, able to maximise throughput of the system by balancing computation and I/O. Moreover, I/O malleability will be achieved through guiding of the ad-hoc storage systems developed in WP2, by dynamically estimating the I/O requirements of applications.

First, a malleability protocol will be designed and implemented. It will allow negotiation of expand and shrink operations between jobs and the ad-hoc storage systems on the one hand, and the job scheduler on the other. The protocol will also support the job-initiated evolution of the resource set. Second, we will develop runtime mechanisms by generalising Flex-MPI. This will provide malleability to applications and allow their execution using the previous negotiation protocol. Finally, we will develop an application programming interface (API) to allow application developers to insert scheduling points into their code that indicate when the application can respond to external malleation requests.

Regarding the proposal, UC3M will lead the task and implement the malleable runtime, including the scheduling-point APIs. BSC and JGU will design the malleability interfaces of the ad hoc storage systems developed in WP2. CINI will contribute to the co-design of the malleability manager with the intelligent controller in WP6 and TUDA will design and implement the malleability protocol and contribute to the scheduling-point APIs. The output of the task T3.1 will provide the base components and extensions for the combined malleability of compute and I/O resources.

Chapter 2

Background

In this chapter, we will first explain the concept of malleability in HPC systems. Next, we will present Flex-MPI and an extension to the MPI standard, which are the two MPI implementations to support malleability that are being considered for ADMIRE project. Next, we will discuss the possible approach of extending the Slurm batch scheduler to deal with malleable jobs.

2.1 Malleability in HPC

Regarding elasticity of resource allocation, parallel jobs can be classified in four different categories [6], as shown in the Table 2.1. Rigid jobs are the most common jobs in HPC, They provide no flexibility for changing their resources after job submission. Moldable jobs allow the resource manager to change the assigned resources before starting the jobs. Allocated resources for evolving jobs can be modified by users, even after job submission and during the execution time. The fourth category, malleable jobs are the most flexible class of jobs, which are the focus of this project. Resource manager could modify the assigned/allocated resources for malleable jobs any time before or during execution time. This modification can be shrinking or expanding of resources. Although malleation operations may be costly and add some overhead to the system, efficient usage of these operations can improve performance for both resource providers and users [7].

2.2 Flex-MPI

Message Passing Interface (MPI) is a standardized and portable message-passing interface designed to orchestrate parallel programs on parallel computing architectures. Flex-MPI is an MPI extension that supports malleability and implements performance-aware dynamic reconfiguration for iterative MPI applications. Flex-MPI is implemented as a library on top of the MPICH [1] implementation and automatically reconfigures the application to run on the number of processes that is necessary to increase the performance such that an application completes within a specified time interval. This runtime modifies the application performance by adding or removing processes whenever it detects that the performance target is not achieved. The reconfiguration process also depends on the user-given performance constraint which can be either the parallel efficiency or the operational cost of executing the application. Alternatively, this reconfiguration can be specified by an external controller that acts as a coordinator and resource manager.

Table 2.1: Job Classification in HPC Systems

		when it decided?	
		at submittal	**during execution**
who decides?	**user**	Rigid	Evolving
	system	Moldable	Malleable

Flex-MPI implements a computational prediction model to decide the number of processes and the process-to-processor mapping that can achieve the required performance objective under a performance constraint. The efficiency constraint results in minimizing the number of dynamically spawned processes to maximize parallel efficiency. The cost constraint focuses on minimizing the operational cost by mapping the newly created dynamic processes to those processors with the smallest cost (expressed in $ per CPU time unit) while satisfying the performance constraint. This metric is particularly relevant when we consider heterogeneous systems where each type of processor may have a different operational cost.

Flex-MPI targets iterative single program multiple data (SPMD) applications with both regular and irregular computation and communication patterns. A large proportion of the SPMD parallel applications are iterative, for instance, linear solvers, particle simulation and fluid dynamics simulators. In the SPMD paradigm, each process executes the same code but operates on a different subset of the data. The usual structure of an iterative SPMD application consists of an initializing section in which each process loads its data partition; what follows is an iterative section during which the processes operate in parallel and communicate with each other to reach a global solution. This approach focuses on applications which use one-dimensional and two-dimensional distributed data structures in which each process stores only its own data partition and does not replicate data managed by the other processes.

Flex-MPI was implemented as a library on top of the MPICH, release v.3.0.4. This makes it fully compatible with all the features of the MPI-3 standard. Figure 2.1 shows the execution environment of an application which consists of the Flex-MPI library, the MPI user application, the Performance API (PAPI) [9] and MPI library, the user-given performance objective and performance constraints, and the resource management system. Figure 2.2 shows the workflow diagram of a malleable MPI application. Each box shows in square brackets the components that provide the corresponding functionality. Initially, the MPI application runs on n processes. At every iteration, the MPI program instrumented to use the Flex-MPI API automatically sends different application performance metrics to the monitoring (M) module (label `1.a`). These include hardware performance counters, communication profiling data, and the execution time for each process. Once Flex-MPI has collected these metrics it returns the control to the MPI application (label `1.b`). Additionally, at every *sampling interval*—consisting of a fixed, user-defined number of consecutive iterations—the monitoring module feeds the gathered performance metrics to the reconfiguring policy (RP) module (label `2`). This allows the RP module to track the current performance of the application and decide whether it needs to reconfigure the application to adjust the performance of the program to the objective. A reconfiguring action involves either the addition (label `3.a`) or removal (label `3.b`) of processes. The computational prediction model (CPM) estimates the

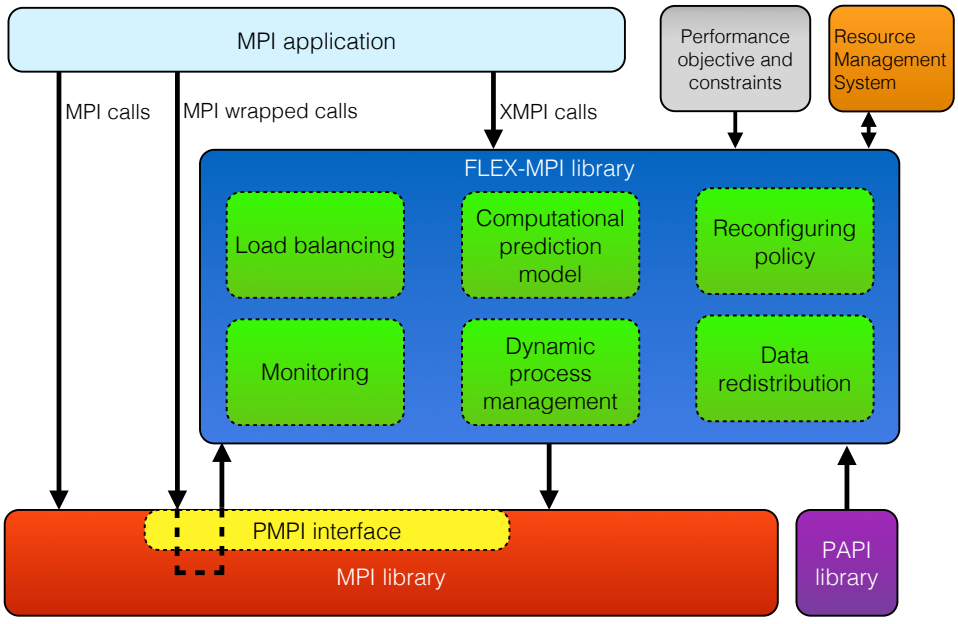

Figure 2.1: Execution environment of a Flex-MPI application.

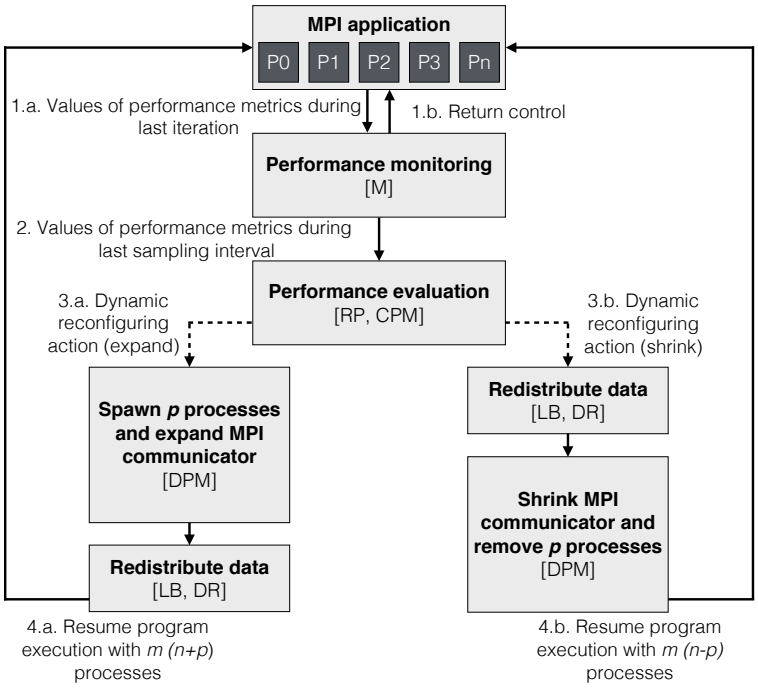

Figure 2.2: Workflow diagram of a malleable MPI application using Flex-MPI.

number of processes and the computing power (in $FLOPs$) required to satisfy the performance objective. Using this prediction, the RP module computes the new process-to-processor mapping based on the number and type of the processors that are available and the performance constraint (efficiency or cost). The number and type of available processors is provided by the resource management system.

The dynamic process management (DPM) module implements the process spawn and remove functionalities and is responsible for rescheduling the processes according to the mapping. A reconfiguring action changes the data distribution between processes, which may lead to load imbalance. Each time a reconfiguring action is carried out, the load balancing (LB) module computes the new workload distribution based on the computing power of the processing elements allocated to the application. The data redistribution (DR) module is responsible for mapping and redistributing the data between processes according to the new workload distribution. We define each of the computing cores of a multi-core processor as a processing element (PE). Once Flex-MPI has reconfigured the application to the new number of processes (m), it resumes its execution (labels `4.a` and `4.b`).

Application developers can access the Flex-MPI library through the API, which consists of a set of high-level interfaces—carrying the `XMPI` prefix—that automatically reconfigure the MPI application. MPI initialize (`MPI_Init`) and finalize (`MPI_Finalize`) functions are wrapped to initialize and finalize the Flex-MPI library functionalities and the MPI environment. MPI point-to-point and collective communication operations are wrapped to collect performance metrics. Wrapped functions are managed using the MPI profiling interface (PMPI) which redirects the function calls to the Flex-MPI library in a user-transparent way.

Figure 2.3 shows a comparison between a simplified legacy code sample and the same code instrumented with Flex-MPI functions. The SPMD application uses a data structure (`vector A`) distributed between the processes (`L4`). In the iterative section of the code (`L5-10`) each process operates in parallel on a different subset of the data structure. At the end of every iteration, the program performs a collective reduce operation (`L9`). In the legacy code all the MPI specific functions (in red) are managed by the MPI library.

The instrumented code consists of: (1) native MPI functions (in red), (2) wrapped functions (in yellow), (3) Flex-MPI functions which allow the parallel program to get and set some library-specific parameters (in blue), and (4) Flex-MPI functions to access the dynamic reconfiguration library functions (in green). Additionally, all the references to the default communicator `MPI_COMM_WORLD` in the legacy code are replaced with `XMPI_COMM_WORLD`, a dynamic communicator provided by Flex-MPI. To simplify the presentation the instrumented code shows the high-level interfaces of the Flex-MPI API without the required function parameters.

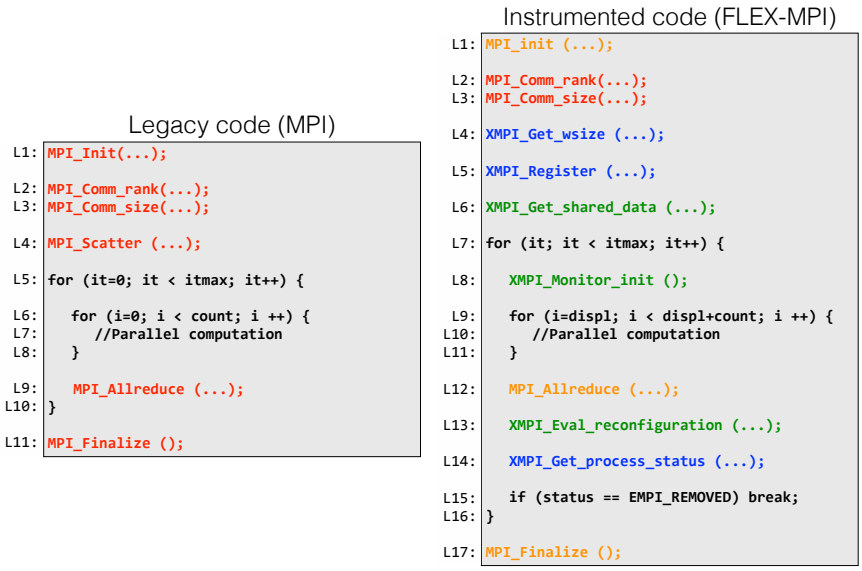

Figure 2.3: Comparison of the legacy code (left) and the instrumented Flex-MPI code (right) of an iterative MPI application.

In Flex-MPI the MPI initialize (`L1`), finalize (`L17`), and communication (`L12`) functions are transparently managed by the Flex-MPI library using the PMPI interface. The rest of the MPI specific functions (`L2-3`) are directly managed by the MPI library. The parallel code is instrumented with a set of functions to get the initial partition of the domain assigned to each process (`L4`) and register each of the data structures managed by the application (`L5`). Registering is necessary to know which data structures should be redistributed every time a reconfiguring action is carried out.

The DR module communicates with the newly spawned processes to pass them the corresponding domain partition before starting the execution of the iterative section (`L6`). The iterative section of the code is instrumented to monitor each process of the parallel application (`L8`) during every iteration. In addition, at every sampling interval, the RP module evaluates whether reconfiguring (`L13`) is required. Then each process checks its execution status (`L14`). In case that the RP module decides to remove a process, this leaves the iterative section (`L15`) and terminates execution. The following sections provide a description of the components.

2.2.1 Monitoring

Flex-MPI uses $FLOP$ to calculate the computing power of each processor as the number of floating point operations per second $FLOPs$. PAPI and PMPI are used to dynamically collect performance metrics from the MPI program and low-level PAPI interfaces to track the number of floating point operations $FLOP$, the real time $Treal$ (i.e. the wall-clock time), and the CPU time $Tcpu$ (i.e. the time during which the processor is running in user mode). PMPI is an interface provided by the MPI library to profile MPI programs and collect performance data without modifying the source code of the application or accessing the underlying implementation. The PMPI interface is used to collect the type of MPI communication operation, the size of the data transferred between processes, and the time spent in communication operations.

2.2.2 Dynamic process management

The dynamic process management module manages the addition and removal of MPI processes, as well as the inter-process communication whenever a reconfiguring action is carried out. This functionality uses the dynamic processes management interface of MPI to spawn dynamic processes at runtime.

MPI provides a default intra-communicator `MPI_COMM_WORLD` which encapsulates the set of all processes initiated by the `mpirun`/`mpiexec` command. From now on we refer to this set of processes as the *initial* set of processes. Those processes which are dynamically spawned and removed at runtime are called *dynamic* pro-

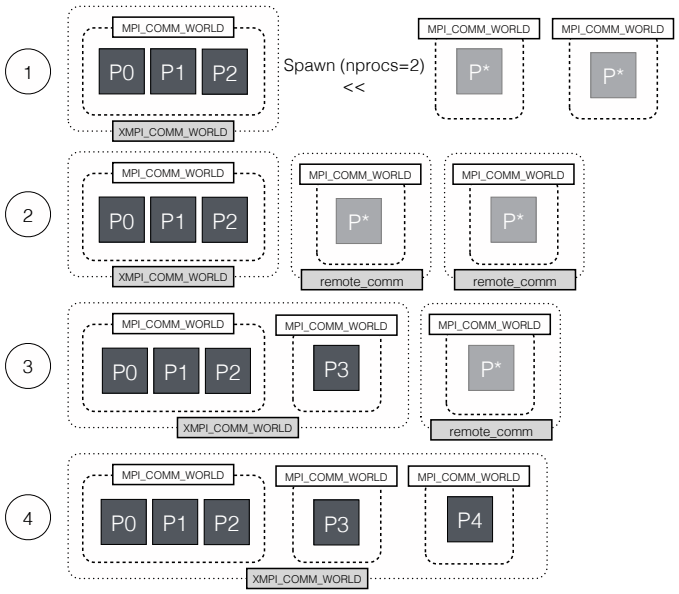

Figure 2.4: Low-level actions of dynamic process management functionality at process creation.

cesses. Due to the restrictions of the current implementation of MPI, only dynamic processes can be removed at runtime. The members of the initial set of processes (default intra-communicator `MPI_COMM_WORLD`) cannot be terminated until the program completes and dynamic processes can not be added to or removed from this default communicator. For this reason, Flex-MPI introduces its own global intra-communicator called `XMPI_COMM_WORLD` to enable communication between initial and dynamic processes.

Figure 2.4 illustrates the behavior of the dynamic process management module when two dynamic processes (`P3,P4`) are added to an MPI program already running on an initial set of processes (`P0-2`) (step 1). Each of the new processes is spawned using an individual call to `MPI_Comm_spawn`. This makes each process have its own (`MPI_COMM_WORLD`) intra-communicator and `remote_comm` remote communicator (step 2). The local and remote communicators are merged by invoking `MPI_Intercomm_merge`, which returns a new `XMPI_COMM_WORLD` intra-communicator encapsulating processes `P0-3` (step 3). The merge function is invoked once more to merge this intra-communicator and the remote communicator of `P4`. The result is a global intra-communicator which encapsulates all of the processes (`P0-4`) (step 4).

The reconfiguring policy dictates both the number of processes and the type of processors on which to spawn them. MPI provides a mechanism to set the *host* key of the `MPI_Info` argument of `MPI_Comm_spawn` to the host name of the compute node where the new process needs to be allocated. The dynamic process management module implements a scheduler which uses the mechanism provided by MPI to map processes to compute nodes with processor types corresponding to those dictated by the reconfiguring policy.

Removing a dynamic MPI process from an application implies disconnecting the process from the communicator `XMPI_COMM_WORLD` to allow the process to leave the iterative section and finish execution by invoking `MPI_Finalize`. This operation is implemented by first deallocating the merged intra-communicator and then allocating a new intra-communicator for the remaining processes. Due to the fact that `MPI_Finalize` is blocking and collective for all the processes in `MPI_COMM_WORLD`, each dynamic process must have been spawned via a separated call to allow individual termination.

Figure 2.5 illustrates the behavior of the dynamic process management module when process `P4` is removed from the previous MPI program (step 1). The current `XMPI_COMM_WORLD` is deallocated. This allows disconnecting `P4` from the rest of the processes (step 2). A new group is then formed via `MPI_Group_incl` to include `P0-P3`, and a new intra-communicator `XMPI_COMM_WORLD` is allocated for this group. `P4` finishes its execution by calling `MPI_Finalize` (step 3).

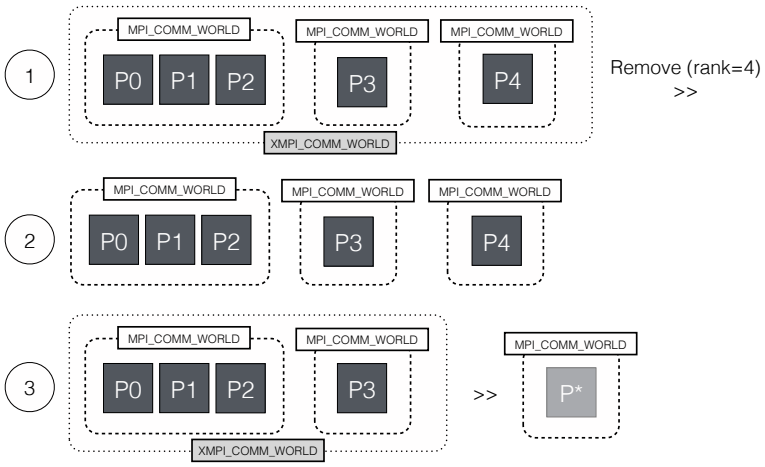

Figure 2.5: Low-level actions of dynamic process management functionality at process removal.

2.2.3 Load balancing

Load balancing is a major issue in parallel applications because it can have a huge impact on the overall performance of the program. Flex-MPI integrates a dynamic load balancing technique for SPMD applications that uses the performance metrics collected by the monitoring functionality to make workload distribution decisions. One of the main advantages of this approach is that it does not require prior knowledge about the underlying architecture. The load balancing module computes the new workload distribution using the values for the computing power of each processor on which the application is running the MPI processes. The idea is to assign to each process a data partition that is proportional to the *relative computing power* which is defined as the computing power of the processor (in $FLOPs$) divided by the sum of the computing power of all of the p processors on which the MPI program is running.

2.2.4 Data redistribution

Flex-MPI provides a user-transparent data redistribution mechanism which is triggered as a result of load balancing when a reconfiguring action is carried out. The data redistribution module uses MPI standard messages to efficiently move data between MPI processes at runtime. The data structures that it can handle must be one-dimensional (e.g., vectors) or two-dimensional (e.g., matrices), and they may be dense or sparse with block-based one-dimensional (row or column) domain decomposition. Once the load balancing module decides the new workload distribution, the data redistribution module maps it to a set of data partitions—one per process—and moves this data from the previous to the new owners.

When the dynamic process management module spawns a new process this will receive a portion of the data which is proportional to the computing power of the processor mapped to the process. When a process is terminated its data is transferred to the remaining processes according to the computing power of the processors mapped to each of these processes.

2.2.5 External controller

Flex-MPI can be also configured as a passive component that receives the reconfiguration commands from an external controller. In this case, the root process of each running application will execute a Flex-MPI proxy running as a separate thread[1] The proxy includes a listener port for receiving the reconfiguration commands from the external controller. These commands include: creating new application processes in certain compute nodes; destroying application processes; performing CPU affinity in which certain application processes are bound to specific CPU cores; activating/deactivating the application monitoring, and specifying the Flex-MPI optimization policy.

[1]Note that there will be a single proxy thread per application, independently of the application number of processes.

When monitoring is activated, the proxy uses a sender port to send the performance metrics to the external controller. These metrics are internally aggregated by Flex-MPI and include the total number or the average number of events (FLOPS, MIPS, I/O operations per second, etc.) among the existing application processes.

2.3 MPI extension to support malleable jobs

The work presented in [3] proposes an extension to the MPI standard to provide malleable capabilities to MPI applications. The main idea behind this proposal is to only introduce few new MPI operations in order to simplify the application programmability and reduce the reconfiguration overheads. Another advantage of this extension is that it is fully integrated with an extension of Slurm that was developed by the same team. In this way, it is possible to use it in real execution environments. In this extension, the reconfiguration is initiated by the resource manager, such that the application acts as a passive entity receiving the resource manager decisions for performing malleable reconfigurations. Algorithm 1 shows a pseudocode of this extension that consists of four new operations. The first one is used for initializing the MPI library considering previous malleable actions. The second one indicates if it is necessary to perform a reconfiguration. The last two operations are used to to define the beginning and end of the reconfiguration window, that is defined as the code section in which a reconfiguration can be carried out. A more detailed description of these operations is shown below.

Algorithm 1: Malleable pseudocode example of the MPI extension to support malleable jobs.

```
1: MPI_Init_adapt(...);
2: for ... do
3:    MPI_Probe_adapt(...);
4:    if (malleable action is performed) then
5:       MPI_Comm_adapt_begin(...);
6:       MPI_Comm_adapt_commit(...);
7:    end if
8: end for
```

- `MPI_INIT_ADAPT`. This operation is equivalent to `MPI_INIT` that serves to initialize the MPI runtime. The difference between them is that with this new operation it is possible to distinguish whether each application process was created at the beginning of the application execution or it is a new process created during a malleable reconfiguration.

- `MPI_PROBE_ADAPT` This operation permits the preexisting processes to query the resource manager if there is a pending reconfiguration and what is the status of each application process. The status can be `STAYING`, if the process remains in the process group after the reconfiguration; `JOINING`, if the process is a newly created instance, and `LEAVING`, if the process is going to be removed from the process group. This information will determine the actions taken by each of the application processes during the reconfiguration.

- `MPI_COMM_ADAPT_BEGIN`. This operation starts the application reconfiguration. Note that the programmer is responsible for inserting this call in a safe and appropriate place of the source code. After the operation execution, two communicators are generated: one equivalent to the one provided by the standard spawn operations, and another one that represents the final global communicator. Both of them only include the processes that remain after the reconfiguration (the ones flagged as `STAYING` and `JOINING`).

- `MPI_COMM_ADAPT_COMMIT`. This operation is used to commit the reconfiguration. The operation modifies `MPI_COMM_WORLD` communcator in the way that any leaving processes are eliminated from it, and any new joining processes are inserted into it. This operation also notifies the Slurm resource manager that the reconfiguration has been completed.

2.4 Slurm

This section introduces the workload manager Slurm, describes its programming interfaces and its extensibility by using plugins, and evaluates possible modifications in order to incorporate malleability based on recent works.

2.4.1 Overview

Slurm [12] is an open source resource and job management system used by a large number of today's HPC facilities. Regarding the proposal, Slurm is the workload manager of choice in the ADMIRE project. Slurm's key functionalities include allocation of compute resources, provisioning of a framework allowing users and administrators to start and monitor workloads, and management of a queue comprising pending jobs [11].

Slurm's centralized architecture provides an ideal environment to connect ADMIRE components in order to manage the overall system state by orchestrating the execution of user-submitted batch jobs. The `slurmctld` is the central controller of Slurm and is responsible for the communication of work to the `slurmd` instances running on each compute node. An overview of the Slurm architecture with its components and a partial list of supported commands is given in Figure 2.6.

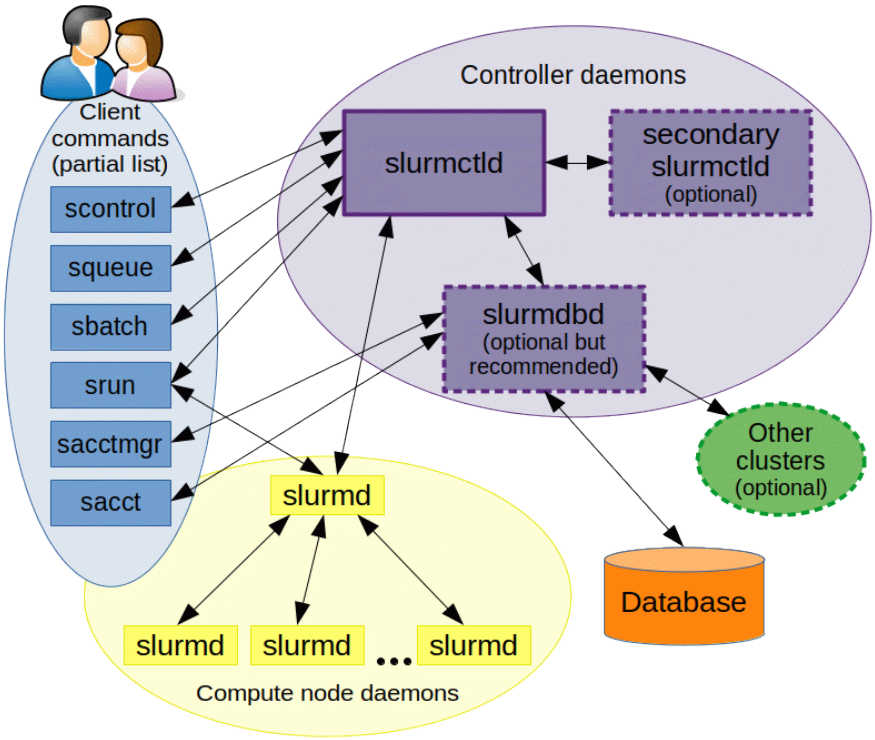

Figure 2.6: Slurm Architecture (taken from [11])

We plan to leverage the collection of APIs provided by Slurm to facilitate the integration into the ADMIRE software stack. Slurm's APIs allow users and applications to directly query data online in order to receive relevant runtime information. The APIs provide methods to query data such as overall Slurm information, job information (regardless of state) and information on compute nodes as well as resource allocation. A complete list of API functions is given in [10]. The ADMIRE APIs exposed to users will make implicit use of provided functions by the Slurm APIs. The actions taken within the ADMIRE API will be forwarded to a plugin connecting ADMIRE components with Slurm. Additionally, we plan to extend and make use of presently available plugin APIs to integrate ADMIRE-related functionalities into well-established HPC use cases (see Section 2.4.2).

2.4.2 Plugins

Slurm provides a plugin interface for general-purpose extensions. We plan to extend Slurm with a plugin to connect it to ADMIRE components (i.e. the Intelligent Controller). The plugin takes the responsibility of communicating job-related actions to Slurm and is the bridging interface between the ADMIRE APIs and the Slurm core components.

In addition to a project-specific plugin, we evaluate the employment of several presently available Slurm plugins for common HPC use cases. These considerations currently comprise the following plugins:

- **Job submit plugin:** Methods provided by this plugin API are initiated when jobs are submitted or modified. A use case for ADMIRE includes setting project-specific parameters in jobs and the interception of jobs after they have been submitted by users. Additionally, our software stack needs to be aware of all modifications of jobs to keep the overall system state up-to-date.
- **Scheduler plugin:** The scheduler plugin determines the schedule of jobs. As the scheduling algorithm is a vital part of the Malleability Manager (see the Task 3.2, in the project proposal), we plan to employ the plugin to apply schedule decisions taken by the Malleability Manager and the Intelligent Controller.
- **Node selection plugin:** This plugin is responsible for the selection of compute resources. One of ADMIRE's key objectives is to create an intelligent I/O software stack. As the placement of jobs on compute resources plays an important role in preventing I/O congestion, we evaluate the employment of this plugin to allow contention-aware resource allocations.

2.4.3 Malleability extension

According to the ADMIRE project proposal, the partners have agreed to extend Slurm to support malleable batch jobs using the plugin interface described in Section 2.4.2. However, this approach only offers limited capabilities when it comes to expanding or shrinking the number of compute nodes assigned to a job at certain scheduling points.

Further investigation showed an extension to Slurm to support malleable jobs [2], based on a previous work where an early Slurm prototype capable of supporting adaptation operations for interactive MPI applications was presented [3]. The authors extended and replaced the `slurmctld` by two components: the Elastic Runtime Scheduler (ERS) and the Adaptive Batch Scheduler (ABS). Whereas the ERS manages *expand* and *shrink* operations, the ABS is a scheduler plugin responsible for scheduling and the reconfiguration of batch jobs. An overview of the architecture is given in Figure 2.7. Within ADMIRE, we plan to extend the ERS to accept *expand* and *shrink* operations from the project-specific ADMIRE plugin (receiving commands from the Intelligent Controller) and forward and apply malleability decisions by employing the proposed ABS.

The development on this extension is part of the EuroHPC project DEEP-SEA [4] that may lead to an inter-project collaboration. The ADMIRE partners agreed that the extension presented in [2] suits our requirements and is the best candidate to extend Slurm with a support for node-level malleability. Furthermore, a cooperation between ADMIRE, DEEP-SEA and the REGALE [5] project is currently investigated. However, this introduces additional considerations. First, the latest version of the extension is based on an outdated Slurm version and will consequently require the adaptation to the latest version. Second, the extension is based on iMPI (see Section 2.3). As ADMIRE bases its adaptive MPI runtime on Flex-MPI (Section 2.2), an adaptation to support Flex-MPI is currently considered.

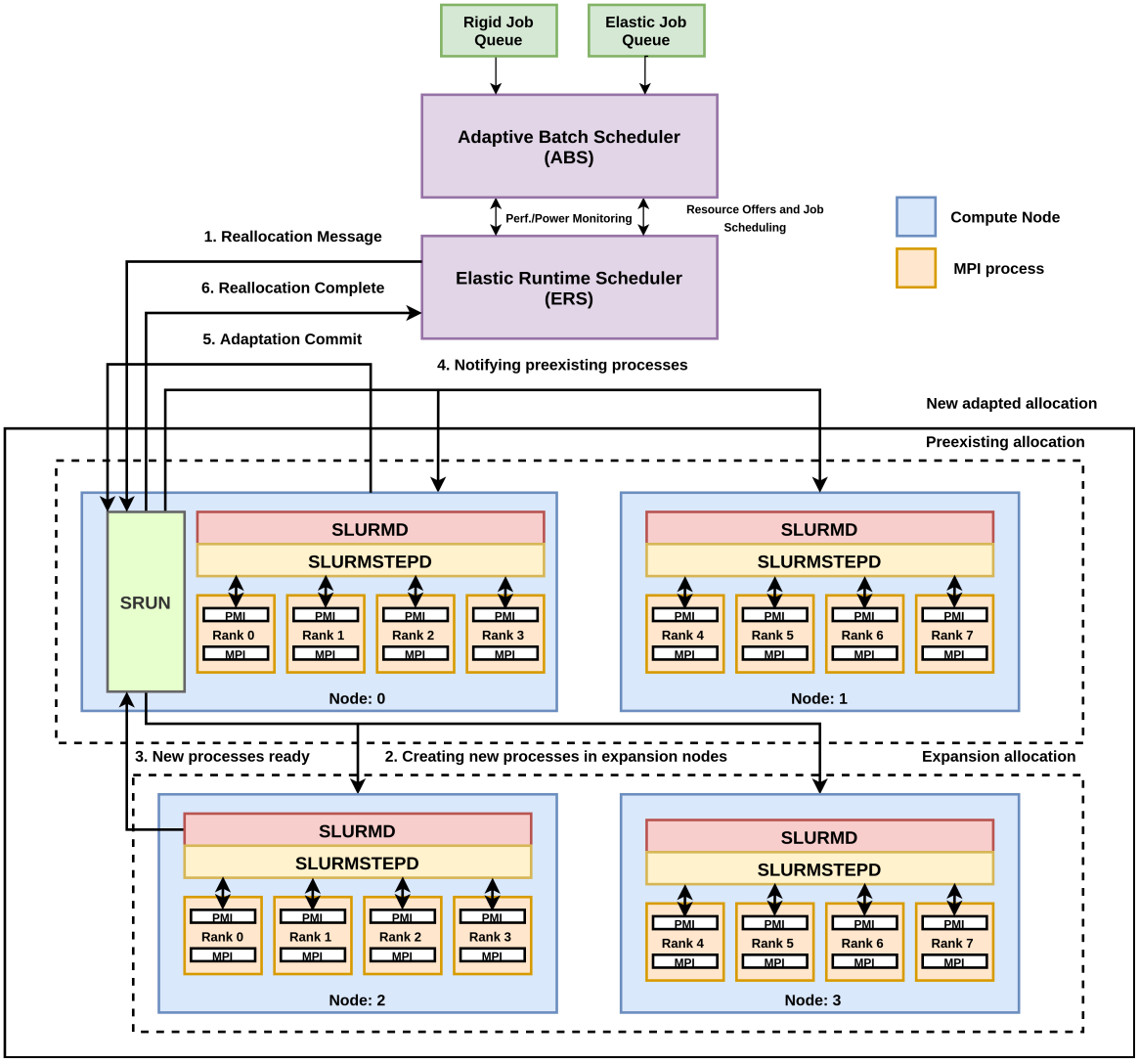

Figure 2.7: Slurm Malleability Extension (taken from [2])

Chapter 3

ADMIRE Architecture

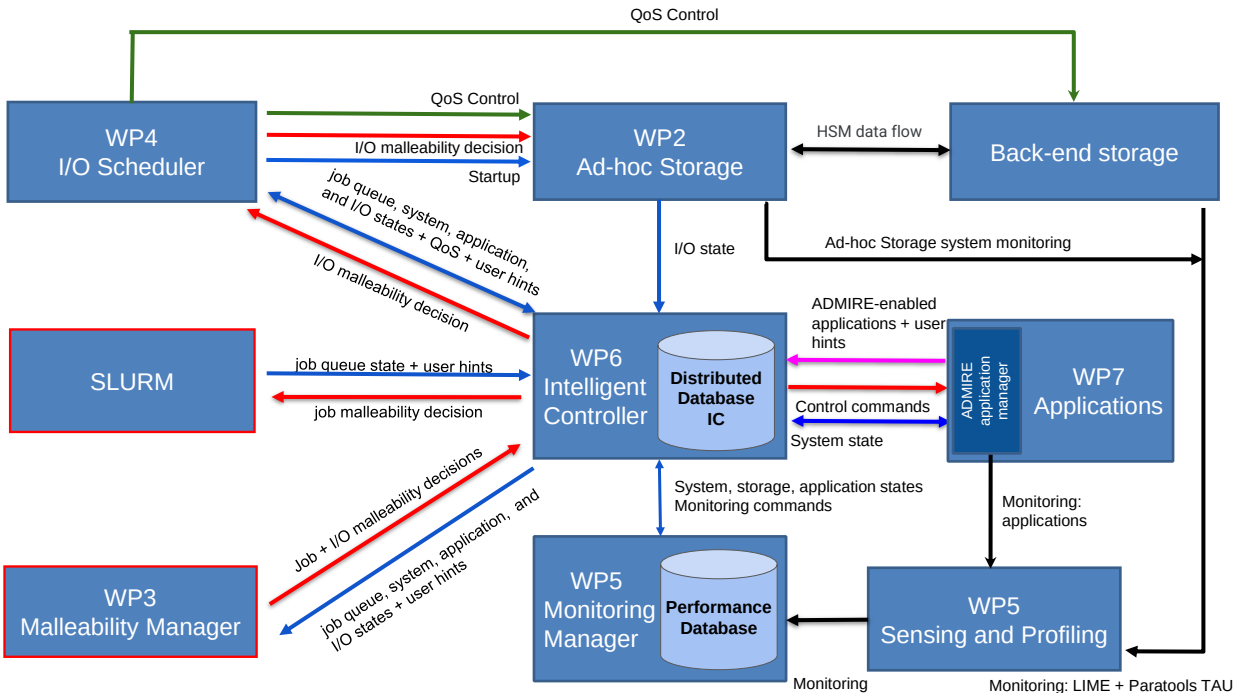

Figure 3.1: ADMIRE architecture overview. Each component developed in the project's scope has included the label of its related Work Package (WP).

Figure 3.1 illustrates an overview of the ADMIRE architecture, its components and the exchanged information (data and control) between these components. The storage subsystem is represented on the upper part of the figure and consists of the ad-hoc and back-end storage systems. The ad-hoc storage system is designed by ADMIRE's WP2 and is responsible for providing to each application an ad-hoc parallel file system tailored to the application's characteristics. The latter one (back-end storage) represents the parallel file system used by the HPC platform (e.g., Lustre). Both storage systems are coordinated by the I/O scheduler (shown in the upper-left corner of the figure), which is responsible for the deployment and configuration of the ad-hoc storage, the specification of Quality-of-Service (QoS) metrics and the implementation of I/O scheduling policies. The applications that are being executed in the platform are shown in the central-right part of Figure 3.1. ADMIRE-enabled applications can provide user-defined application-specific information to the system to aid the identification of I/O patterns and reconfiguration-safe states in which malleable commands can be executed. Both applications and storage systems are monitored by the Sensing and Profiling component (lower-right corner of the figure), which is developed in WP5. This component is responsible for collecting system-wide performance metrics at node-level which will be stored in an internal database. Also, this component will enable the generation of

performance models to aid the malleability manager for a more concrete schedule solution. The Monitoring Manager (lower-central part of the figure) will manage this database and will generate performance metrics and models related to each running application concerning both I/O as well as computational activities. The Malleability Manager (lower-left corner of the figure) is responsible for determining the malleable actions related to each running application and ad-hoc storage system. These actions may produce reconfiguration of processes/threads of a specific application or the deployment/removal of one or more instances of the ad-hoc storage to better balance the computation and I/O requirements. The Intelligent Controller (central part of the figure) has different roles. The first one is to collect the current system status using the information collected from Slurm, the Monitoring Manager, the storage systems, and the applications. This will include combined information about the hardware status, the existing running applications, and the storage. This information will be kept in an internal distributed database. The second role of the Intelligent Controller is to generate performance models of these components and use them to predict potential performance bottlenecks in the system. These models will also be provided to the Malleability Manager and I/O scheduler to support the decision-making of both of them. The third main role of the Intelligent Controller is to coordinate the actions taken by other ADMIRE's components. Examples of these actions are (1) to activate/deactivate each application monitoring and (2) to send the malleable decisions taken by the Malleability Manager to the I/O scheduler or the applications, in case of being I/O-related or application-related decisions, respectively.

3.1 WP3 Focus

To apply malleability, we need to support malleability at application level (programming model, e.g. [8]), node level (OS and runtime system), and resource management and job scheduling level. We will provide malleability at application level by preparing a programming model to define a set of so-called scheduling points that address the possible points for applying malleable operations. At node level, we need to extend the core of Slurm to supporting malleability since Slurm assumes all jobs as rigid jobs. Finally, we need a malleability management to suggest schedule solutions based on a set of malleability mechanisms to complete the cycle of malleability in the project.

In state-of-the-art malleability management systems, only computational resources are assumed to be malleable. Although this type of malleability has been efficiently applied on several real-world problems, it could not be efficient enough for the execution of huge data-intensive applications in HPC environments, as the malleability of computational resources also affects the I/O requirements of such applications.

In the ADMIRE project, we will work toward a combination of computation and I/O malleability, such that we could achieve a balance for computation and I/O requirements of the jobs. The Malleability Manager is one of the three main components in ADMIRE, alongside the I/O scheduler and the ad-hoc file system. Malleability management balances I/O and compute performance via dynamic scaling of application resources. The ADMIRE malleability management module, which is the task of WP3, will offer several configurable policies for the elasticity of computational and I/O resources.

In the current architecture of ADMIRE, the Malleability Manager (MM) obtains the system's current state from the Intelligent Controller and provides back the schedule solutions at each scheduling point of the application. The Intelligent Controller analyzes the provided schedule solutions to assess their compliance with the system requirements and limitations. In affinity cases, the intelligent controller forwards the schedule solutions to the Job and I/O schedulers to reconfigure the job resources. Otherwise, the Intelligent Controller overrides the provided schedule solutions.

3.2 Malleability Manager Workflow

This section describes the actions taken by the Malleability Manager for each execution scenario. An execution scenario defines the existing system conditions when a malleable action is taken. In the following sections, we consider four different execution scenarios: intra-node malleability, inter-node malleability using a pool of resources, inter-node malleability using Slurm, and I/O malleability.

Algorithm 2: Malleability Manager workflow for intra-node malleability.

```
1:  The Malleability Manager is executed
2:  for each running application do
3:      Intelligent Controller: system status notification including application performance
4:      Malleability Manager: application performance analysis
5:      Malleability Manager: malleable action
6:      Intelligent Controller: malleable action analysis
7:      if malleable action is performed then
8:          Intelligent Controller: wait for application
9:          Intelligent Controller: malleable action forwarding
10:         Intelligent Controller: wait for completion
11:         Intelligent Controller: application status update
12:     end if
13: end for
```

3.2.1 Intra-node malleability

This strategy can be used when the application scheduler assigns exclusive compute nodes to each executed application. In this context, the main focus of this approach is twofold: first, to leverage the malleability for using the unassigned cores. For instance, assuming that each compute node has 32 cores and an application is originally executed with 48 processes. Then, when the application is executed, it will have two compute nodes exclusively assigned to it. In this way, there will be 16 extra cores that could be used by increasing the application size by 16. This approach will potentially allow to increase the application CPU performance if it has a good scalability, consequently, compute-intensive applications will be benefited by the strategy. However, the application I/O throughput is not likely to be improved given that the number of compute nodes will not change. The second objective of this strategy is to increase the application load-balance by changing the number of allocated resources in each compute node. Note that, this enhancement can be applied at CPU-level or at I/O-level (by adjusting, by means of malleability, the I/O throughput related to each compute node). Algorithm 2 shows the Malleability Manager workflow for the intra-node malleability.

- Line 1: the Malleability Manager is executed when ADMIRE framework is deployed.
- Line 3: the Intelligent Controller creates and updates the application performance model of each running application. In addition, the Intelligent Controller produces a system-wide performance forecast considering the models of all running applications and combining this information with other system-wide performance metrics collected by the Sensing and Profiling module. This forecast will include potential contention situations that may occur in the future as well as the prediction of future system performance indicators. All this information is sent to the Malleability Manager.
- Line 4: the Malleability Manager uses the previous information provided by the Intelligent Controller to analyze the application performance.
- Line 5: based on this analysis, a decision of performing malleable actions at node-level is made. A malleable decision, that is related to a specific application, is sent to the Intelligent Controller. This command will include the number of threads that the application has to create or destroy in certain compute nodes involved in this reconfiguration.
- Line 6: the Intelligent Controller will analyze the reconfiguration command from a system holistic perspective. This command could be overridden in case of being counterproductive according to the results of this analysis.
- In case that the reconfiguration command is feasible (Line 7), the Intelligent Controller will wait until the application is in a safe reconfiguration state (Line 8).

- Line 9: after that, the Intelligent Controller sends the reconfiguration command to the application. This command will involve the creation/destruction of threads and/or processes in the compute nodes that are currently used by the application.
- Line 10: the Intelligent Controller waits for the completion of the application reconfiguration. When this operation is performed, then the application sends a notification to the Intelligent Controller.
- Line 11: after the completion of the application reconfiguration, the Intelligent Controller updates the System Status Table that contains the list of all the executing applications and resources (compute nodes, I/O nodes) associated to each running application.

Algorithm 3: Malleability Manager workflow for inter-node malleability using a pool of resources.

```
1:  The Malleability Manager is executed
2:  for each running application do
3:      Slurm: job execution
4:      Intelligent Controller: system status notification including application performance
5:      Malleability Manager: application performance analysis
6:      Malleability Manager: malleable action
7:      Intelligent Controller: malleable action analysis
8:      if malleable action is performed then
9:          Intelligent Controller: wait for application
10:         Intelligent Controller: malleable action forwarding
11:         Intelligent Controller: wait for completion
12:         Intelligent Controller: application status update
13:     end if
14: end for
```

3.2.2 Inter-node Malleability Using a Pool of Resources

The current implementation of Slurm does not support the dynamic allocation of resources for applications that are already being executed. In this way, if an application increases its size by means of a malleable reconfiguration, we need to modify the Slurm or its plugins for allocating on-demand the new compute nodes. Note that given the complexity of Slurm, the modification of these plugins will take considerable effort and time. The strategy shown in this section is designed to provide a quick prototype for evaluating the ADMIRE architecture without the intervention of Slurm. This will permit us to evaluate the efficiency of the techniques implemented by the Malleability Manager in early stages of the project.

Inter-node malleability enables increasing or decreasing the number of compute nodes allocated to a given application. Therefore, in contrast to intra-node malleability, such an approach might benefit both the CPU and I/O performance of the application. Consequently, this approach suits both computation and I/O intensive applications.

In this strategy, each submitted application will have an extra pool of requested compute nodes that will be originally unused. For instance, assuming that each compute node has 32 cores and an application is originally executed with 64 processes, then, when the job is submitted to Slurm, the user will be requested to specify the requested to specify the required and the maximum number of compute nodes. The maximum number will indicate the pool size related to the job. Note that both the Intelligent Controller and Malleability Manager are aware of the actual number of resources assigned to each running application. In this way, if a malleable reconfiguration is carried out, it will be possible to scale the application up to 96 processes without involving Slurm. Algorithm 3 shows the Malleability Manager workflow for the inter-node malleability using a pool of resources.

- Line 1: the Malleability Manager is executed when ADMIRE framework is deployed.
- Line 3: Slurm executes a new job allocating more resources than the ones originally needed by the application. This information is sent to the Intelligent Controller.

- Line 4: the Intelligent Controller creates and updates the application performance model of each running application considering, for each one of them, both the allocated resources and the currently used ones. In addition, the Intelligent Controller produce a system-wide performance forecast considering the models of all running applications and combining this information with other system-wide performance metrics collected by the Sensing and Profiling module. This forecast will include potential contention situations that may occur in the future as well as the prediction of future system performance indicators. All this information is sent to the Malleability Manager.

- Line 5: the Malleability Manager uses the previous information provided by the Intelligent Controller to analyze the application performance.

- Line 6: based on this analysis, a decision of performing malleable actions at node-level is made. A reconfiguration command, that is related to a specific application, is sent to the Intelligent Controller. This command will include the number of processes that the application has to create or destroy in certain compute nodes among the originally allocated ones.

- Line 7: the Intelligent Controller will analyze the reconfiguration command from a system holistic perspective. This command could be overridden in case of being counterproductive according to the results of this analysis.

- In case that the reconfiguration command is feasible (Line 8), the Intelligent Controller will wait until the application is in a safe reconfiguration state (Line 9).

- Line 10: after that, the Intelligent Controller sends the reconfiguration command to the application. This command will involve the creation/destruction of processes in the compute nodes that are currently used by the application. Note that Slurm is not involved during the reconfiguration process.

- Line 11: the Intelligent Controller waits for the completion of the application reconfiguration. When this operation is performed, then the application sends a notification to the Intelligent Controller.

- Line 12: after the completion of the application reconfiguration, the Intelligent Controller updates the System Status Table that contains the list of all the executing applications and resources (compute nodes assigned and used, I/O nodes, etc.) associated to each running application.

Algorithm 4: Malleability Manager workflow for inter-node malleability using Slurm.

```
1: The Malleability Manager is executed
2: for each running application do
3:     Slurm: job execution
4:     Intelligent Controller: system status notification including application performance
5:     Malleability Manager: application performance analysis
6:     Malleability Manager: malleable action
7:     Intelligent Controller: malleable action analysis
8:     if malleable action is performed then
9:         Intelligent Controller: for malleable expansion, allocate new resources via Slurm
10:        Intelligent Controller: wait for application
11:        Intelligent Controller: malleable action forwarding
12:        Intelligent Controller: wait for completion
13:        Intelligent Controller: for malleable shrinking, release existing resources via Slurm
14:        Intelligent Controller: application status update
15:    end if
16: end for
```

3.2.3 Inter-node Malleability Using Slurm

With this approach, a certain application might be executed on an increasing/decreasing number of compute nodes. Like in the previous approach, by means of malleability, it is possible to change both the application CPU performance and I/O bandwidth. The main difference with the inter-node malleability using a pool of resources is that now, Slurm is actively involved in the application reconfiguration. Algorithm 4 shows the Malleability Manager workflow for inter-node malleability using Slurm.

- Line 1: the Malleability Manager is executed when ADMIRE framework is deployed.
- Line 3: Slurm executes a new job allocating the resources needed by the application. This information is sent to the Intelligent Controller.
- Line 4: the Intelligent Controller creates and updates the application performance model of each running application. In addition, the Intelligent Controller produce a system-wide performance forecast considering the models of all running applications and combining this information with other system-wide performance metrics collected by the Sensing and Profiling module. This forecast will include potential contention situations that may occur in the future as well as the prediction of future system performance indicators. All this information is sent to the Malleability Manager.
- Line 5: the Malleability Manager uses the previous information provided by the Intelligent Controller to analyze the application performance.
- Line 6: based on this analysis, a decision of performing malleable actions at node-level is made. A reconfiguration command, that is related to a specific application, is sent to the Intelligent Controller. This command will include the number of processes that the application has to create or destroy in certain compute nodes among the originally allocated ones.
- Line 7: the Intelligent Controller will analyze the reconfiguration command from a system holistic perspective. This command could be overridden in case of being counterproductive according to the results of this analysis.
- In case that the reconfiguration command is feasible and the application has to be expanded (Line 8), the Intelligent Controller ask Slurm to allocate the new resources (compute nodes) involved in the operation (Line 9).
- Line 10: the Intelligent Controller will wait until the application is in a safe reconfiguration state.
- Line 11: after that, the Intelligent Controller sends the reconfiguration command to the application. This command will involve the creation/destruction of processes in the compute nodes that are currently used by the application. Note that Slurm is not involved during the reconfiguration process.
- Line 12: the Intelligent Controller waits for the completion of the application reconfiguration. When this operation is performed, then the application sends a notification to the Intelligent Controller.
- Line 13: if the application has reduced its size, then the Intelligent Controller notifies Slurm the compute nodes that are not longer used by the application. After that, Slurm deallocates these resources.
- Line 14: after the completion of the application reconfiguration, the Intelligent Controller updates the System Status Table that contains the list of all the executing applications and resources (compute nodes assigned and used, I/O nodes, etc.) associated to each running application.

3.2.4 I/O Malleability

The I/O malleability follows the Algorithm 5, that is similar to the Intra-node Malleability one but interacts with the I/O Scheduler and does not need any application interaction.

Algorithm 5: Malleability Manager workflow for I/O malleability.

```
1:  The Malleability Manager is executed
2:  for each running application do
3:      Intelligent Controller: system status notification
4:      Malleability Manager: application performance analysis
5:      Malleability Manager: malleable action (add or remove ad-hoc storage nodes)
6:      Intelligent Controller: malleable action analysis
7:      if malleable action is performed then
8:          Intelligent Controller: malleable action forwarding
9:          I/O Scheduler applies action
10:         Intelligent Controller: wait for completion
11:         Intelligent Controller: application status update
12:     end if
13: end for
```

- Line 1: the Malleability Manager is executed when ADMIRE framework is deployed.
- Line 3: the Intelligent Controller creates and updates an application performance model for each running application. In addition, the Intelligent Controller produce a system-wide performance forecast considering the models of all running applications and combining this information with other system-wide performance metrics collected by the Sensing and Profiling module. This forecast will include potential contention situations that may occur in the future, as well as the prediction of future system performance indicators. All this information is sent to the Malleability Manager.
- Line 4: the Malleability Manager uses the previous information provided by the Intelligent Controller to analyze the application performance.
- Line 5: based on this analysis, in line 5 a decision of performing malleable actions at job-level is made. An I/O reconfiguration command, that is related to a specific application, is sent to the Intelligent Controller. This command provides the addition (to increase I/O performance) or the removal (if I/O nodes should be used for another application, or we detect that the I/O phase does not need parallelism, among others) of I/O nodes in the ad-hoc storage system.
- Line 6: the Intelligent Controller will analyze the reconfiguration command from a system holistic perspective. This command could be overridden in case of being counterproductive, according to the results of this analysis.
- In case that the reconfiguration command is feasible (Line 7), the Intelligent Controller can issue the command to the I/O Scheduler (Line 8).
- Line 9: the I/O Scheduler issues the command to the ad-hoc storage and redistribution of data starts considering the system status and other metrics.
- Line 10: the Intelligent Controller waits for the completion of the ad-hoc storage reconfiguration. When this operation is completed, the ad-hoc storage system sends a notification to the Intelligent Controller.
- Line 11: after the completion of the storage reconfiguration, the Intelligent Controller updates the System Status Table that contains a list of all the executing applications and resources (compute nodes, I/O nodes, etc.) associated to each running application.

3.2.5 Safe Reconfiguration State

Changing the number of resources an application uses while running is a potentially disruptive operation. If not correctly planned and handled, it may produce application failures or wrong results. A dynamic reconfiguration plan can only be initiated at particular points in the application execution in which a consistent state can be

reached, and the introduced modifications to the data decomposition/distributions among the computing entities do not result in information loss or semantics changes.

If such points or regions exist, the application can be dynamically reconfigured either to improve overall system performance or decrease the number of system resources used. We define such regions of code in the execution of an application as *safe reconfiguration points* and the application state reached in those points *safe reconfiguration state*.

Typical safe reconfiguration points in parallel applications are global checkpoints, which induce global barriers among all computing entities. However, in general, identifying reconfiguration-safe regions of code depends on the parallel programming model used, and they are challenging to discover without active aid from the application developers. In addition, if the programming model used does not provide transparent dynamic reconfiguration, the correctness of the reconfiguration plan has to be granted and validated directly by the application developers.

To partially overcome such difficulties, ADMIRE-enabled applications may directly interact with the Intelligent Controller (IC) through an API (described in the deliverable D6.1) to provide, among others, information about reconfiguration-safe points in which reconfiguration commands, coming from the IC, may be safely executed by the application.

Chapter 4

Application programming interface

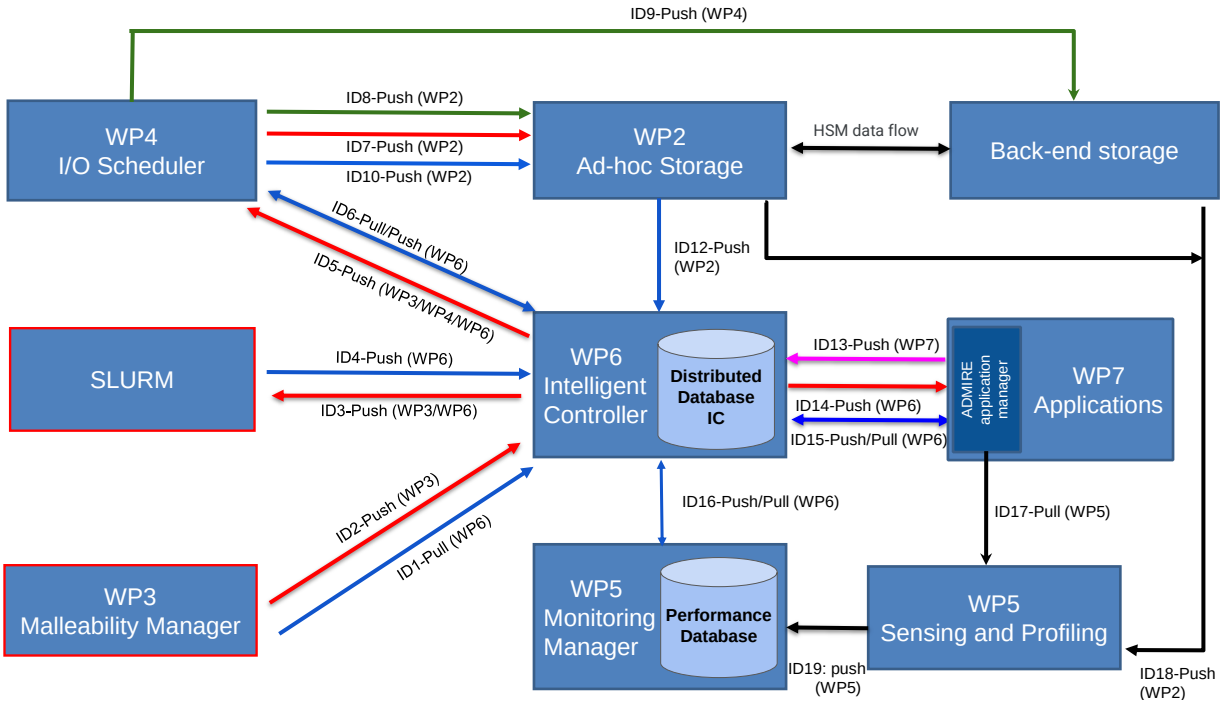

Figure 4.1: ADMIRE controlflow between the components.

Figure 4.1 shows the control flow diagram of the overall ADMIRE architecture. We can observe that the Malleability Manager only communicates with the Intelligent Controller, although, the information provided by the Intelligent Controller will be forwarded to the Slurm, the I/O scheduler, and the applications (via the ADMIRE application manager). Since there are a tight collaboration between the Intelligent Controller and the Malleability Manager, considering the readability of the whole ADMIRE architecture, we preferred to discuss some of the interfaces in the deliverable D6.1. Consequently, in this section, we would explain a single method that is less coupled with the Intelligent controller and include the remaining methods as part of APIs in D6.1.

4.1 Interface of malleability manager

As shown in the Figure 4.1), there are two communication channels between the Malleability Manager and the Intelligent Controller. The first channel, the function `ADM_getSystemStatus` (ID1 in the Figure 4.1) provides the Malleability Manager with information collected and processed by the Intelligent Controller. This information includes platform and application status information, performance models, and user hints, which offer a global view of the current state of the system, as well as a forecast of future platform states. This function

is related to the Intelligent Controller and is depicted in the deliverable D6.1. The second channel, function `ADM_suggestScheduleSolution` (ID2 in the Figure 4.1) includes the job and I/O malleable decisions made by the Malleability Manager for certain running applications and ad hoc storage systems, respectively. A description of this interface is shown below. We are considering two possible implementations for the Malleability Manager: as an independent thread/process or as a library linked with the Intelligent Controller code. Note that the API described in this section is valid for both alternatives.

Name: *ADM_suggestScheduleSolution*
input : struct scheduleSolution. Structure with the combined job and ad hoc storage scheduler solution. It contains the following fields:

- `applicationID`. Application subjected to be reconfigured.
- `adhocStorageID`. Ad hoc storage subjected to be reconfigured.
- `assignedPartition`. Preferred partition of Slurm to run the job.
- `assignedNodes`. List of compute nodes involved in the job reconfiguration.
- `assignedProcessesPerNode`. Number of processes that have to be created (if positive) or removed (if negative) for each one of the previous nodes.
- `assignedThreadsPerNode`. Number of threads that have to be created (positive values) or removes (negative values) for each one of the previous nodes.
- `assignedBandwidth`. Assigned I/O bandwidth for the considered application.
- `jobSchedule`. suggested I/O scheduling policy for the application.
- `adhocStorageNodes`. List of compute nodes involved in the ad hoc storage reconfiguration.
- `adhocStorageMalleabilityRequest`. Number of I/O storage instances that have to be created (positive values) or removed (negative values) for each one of the previous nodes.

output: int exitValue: 0 success, -1 failure
Description:
This function provides the schedule decision including job and I/O malleability to the Intelligent Controller

---

4.2 UML diagram

Figure 4.2 shows the UML diagram of the functions provided by the Malleability Manager. The diagram illustrates the API definitions related to the Intelligent Controller.

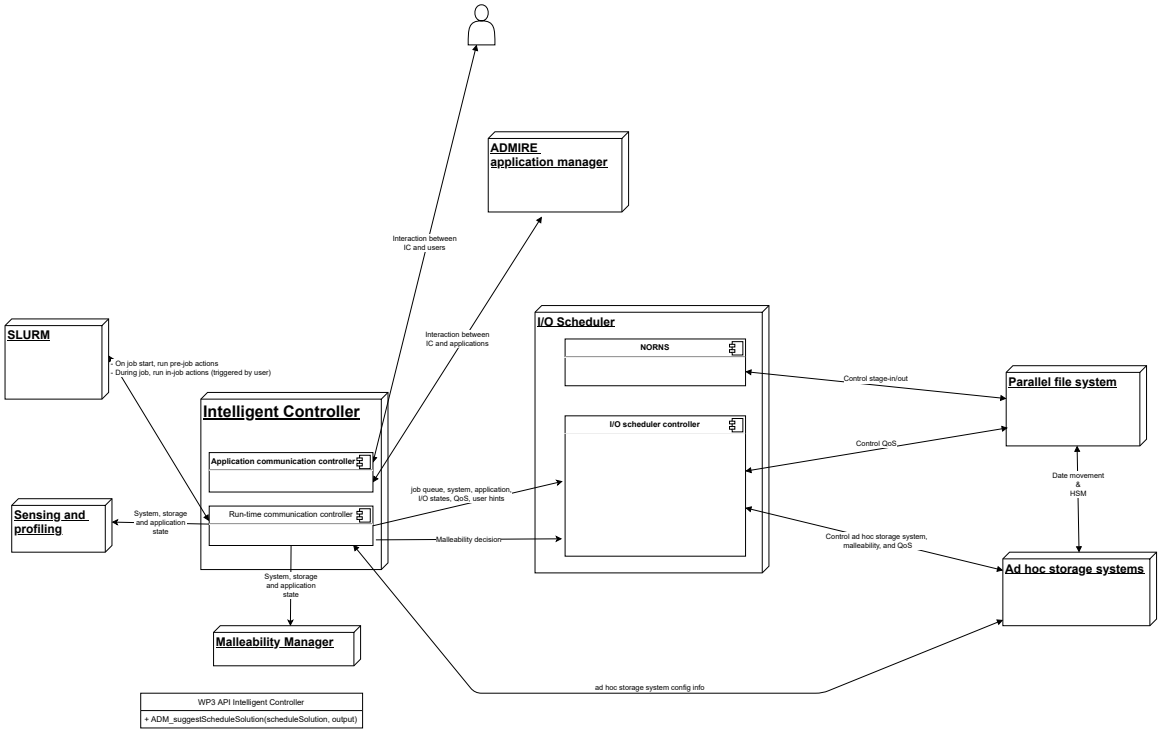

Figure 4.2: UML diagram related to WP3 API.

Chapter 5

Conclusion

To efficiently benefit from the capacity provided by all HPC elements, such as multi-tiered storage, the ADMIRE project aims at providing a software stack for execution of data-intensive applications in exascale HPC environments. Our proposed solution would be a complete framework including application model, monitoring, performance modeling, resource management and job scheduling. Malleability management is an important part of the framework that focuses on balancing computation and storage requirements of data-intensive HPC applications.

To attain an extensive malleability management process, developing a runtime mechanism to provide malleability, at the resource management level, is the first step. Then we need to design and implement malleability protocol(s) for collaborating ad hoc parallel storage systems and I/O scheduler. By providing a set of APIs to facilitate malleability in the application level we will complete the task.

In this document, we first presented the background and technologies needed to better understand the problem. Then considering the role of malleability in ADMIRE framework, we provided the malleability requirements. According to position of malleability manager and its communication with the other components in ADMIRE architecture, we proposed a set of APIs.

Appendix A

Terminology

- Ad hoc Storage System, ephemeral storage system that only exists in a determined period, i.e. during a job's execution.
- CLI, command line interface.
- DRAM, dynamic random-access memory.
- EBNF, Extended Backus–Naur Form is a family of metasyntax notations, any of which can be used to express a context-free grammar. EBNF is used to make a formal description of a formal language such as a computer programming language. They are extensions of the basic Backus–Naur form (BNF) metasyntax notation.
- In situ data, processing the data where it is originated.
- In transit data, processing the data when it is moved.
- NORNS, data transfer service for HPC developed at BSC.
- NVM, non-volatile memory.
- PFS, parallel file system.
- POSIX, Portable Operating System Interface, family of standardized functions.
- QoS, Quality of Service.
- RDMA, remote direct memory access.
- RPC, remote procedure call.
- Slurm, job submission system widely used.
- SSD, solid state drive.
- Object store, persistent storage system where data are stored not as file but as objects. In its canonical implementation Object are immutable and the API is limited to PUT, GET and DELETE. More sophistical object store have been developed on the ground of these concepts such as ADMIRE Data Clay.
- Disaggregated Storage, storage systems where all the storage capabilities are centralized in dedicated network attached storage servers. This approach allows connected compute nodes to access a storage capacity without constraints related to the capacity of a single storage device.
- PFS, Parallel File System, type of distributed file system supporting a global namespace and spread across multiple storage servers.

- Node Local Storage, ability for a compute server to store persistent data on physically local storage devices.
- Ephemeral Storage, file systems which are making persistent (surviving across system reboot) but which are designed to be deployed and destroyed over a limited period of time, from few hours up to few months.
- API, Application Programming Interface, a mechanism that enables an application or service to access a resource within another application or service. The application or service doing the accessing is called the client, and the application or service containing the resource is called the server.
- Rest API, such APIs can be developed without constraint and the programming language and support a variety of data formats. The only requirement is that they align to the following six REST design principles - Uniform interface, Client-server decoupling, Statelessness, Cacheability, Code on demand (optional).
- OSS, an Object Store Server in the Lustre terminology is a computing server in charge of managing the ingest of data, including generation of the data protection, and ship these data to the correct Object Store Target.
- OST, Object Store Target in the Lustre terminology is a storage server accommodating potentially a large number of hard drives and/or NMVes. The OST write the data received from the OSS and make them persistent.
- MDS, MetaData Server.
- MDT, MetaData Target.
- Stripe, an elementary chunk of data according to the Lustre terminology. A large file is split in multiple stripes and each stripe is sent to an individual OST. The higher is the number of stride, the higher is the parallelism.
- Monitoring Manager,
- Intelligent Controller,
- Monitoring Daemon,
- TBON, Tree Based Overlay Network,
- PromQL, the query language supported by the Prometheus database. Syntax, documentation and examples are available here: https://prometheus.io/docs/prometheus/latest/querying.

Bibliography

[1] Abdelhalim Amer, Pavan Balaji, Wesley Bland, William Gropp, Yanfei Guo, Rob Latham, Huiwei Lu, Lena Oden, Antonio J. Pe~na, Ken Raffenetti, Sangmin Seo, Min Si, Rajeev Thakur, Junchao Zhang, and Xin Zhao. *MPICH User's Guide*. Mathematics and Computer Science Division, Argonne National Laboratory, November 10, 2017.

[2] Mohak Chadha, Jophin John, and Michael Gerndt. Extending slurm for dynamic resource-aware adaptive batch scheduling. In *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 223–232, 2020. doi:10.1109/HiPC50609.2020.00036.

[3] Isaías Comprés, Ao Mo-Hellenbrand, Michael Gerndt, and Hans-Joachim Bungartz. Infrastructure and API extensions for elastic execution of MPI applications. In *Proceedings of the 23rd European MPI Users' Group Meeting*, EuroMPI 2016, page 82–97, New York, NY, USA, 2016. Association for Computing Machinery. URL: https://doi.org/10.1145/2966884.2966917, doi:10.1145/2966884.2966917.

[4] EuroHPC. Programming environment for european exascale systems, 2021. URL: https://deep-projects.eu/.

[5] EuroHPC. Regale - open architecture for exascale supercomputers, 2021. URL: https://regale-project.eu/.

[6] Dror G. Feitelson and Larry Rudolph. Towards convergence in job schedulers for parallel supercomputers. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–26, 1996.

[7] Abhishek Gupta, Bilge Acun, Osman Sarood, and Laxmikant V. Kalé. Towards realizing the potential of malleable jobs. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10, 2014. doi:10.1109/HiPC.2014.7116905.

[8] Chao Huang, Orion Lawlor, and L. V. Kalé. Adaptive mpi. *Springer Lecture Notes in Computer Science*, 2958(3):1–43, 2004. URL: https://doi.org/10.1007/978-3-540-24644-2_20.

[9] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. PAPI: A Portable Interface to Hardware Performance Counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[10] SchedMD. Slurm APIs, 11 2019. URL: https://slurm.schedmd.com/api.html.

[11] SchedMD. Slurm workload manager, 8 2021. URL: https://slurm.schedmd.com/.

[12] Andy B. Yoo, Morris A. Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 44–60, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.