Singh, D. E., Carretero, J. (2019). Combining malleability and I/O control mechanisms to enhance the execution of multiple applications. *The Journal of Systems and Software*, 148, pp. 21-36.

DOI: 10.1016/j.jss.2018.11.006

Combining malleability and I/O control mechanisms to enhance the execution of multiple applications

David E. Singh *, Jesus Carretero

Computer Science and Engineering Department, Carlos III University of Madrid, Leganés, Spain

Abstract

This work presents a common framework that integrates CLARISSE, a cross-layer runtime for the I/O software stack, and FlexMPI, a runtime that provides dynamic load balancing and malleability capabilities for MPI applications. This integration is performed both at application level, as libraries executed within the application, as well as at central-controller level, as external components that manage the execution of different applications. We show that a cooperation between both runtimes provides important benefits for overall system performance: first, by means of monitoring, the CPU, communication, and I/O performances of all executing applications are collected, providing a holistic view of the complete platform utilization. Secondly, we introduce a coordinated way of using CLARISSE and FlexMPI control mechanisms, based on two different optimization strategies, with the aim of improving both the application I/O and overall system performance. Finally, we present a detailed description of this proposal, as well as an empirical evaluation of the framework on a cluster showing significant performance improvements at both application and wide-platform levels. We demonstrate that with this proposal the overall I/O time of an application can be reduced by up to 49% and the aggregated FLOPS of all running applications can be increased by 10% with respect to the baseline case.

1. Introducción

One of the existing challenges in the design of High Performance Computing (HPC) infrastructures is to efficiently balance the computational and storage I/O resources of the platform (Panziera et al., 2017). This balance depends on the interaction between the hardware resources, the system software stack, and the executing applications. In general, this interaction is difficult to tackle because of the complexity of the components and the dynamic nature of a HPC platform that executes different applications with distinct characteristics. Currently, there are no existing solutions that completely address this problem (Habib et al., 2016). For instance, job schedulers are based mostly on static resource allocation, where storage I/O is mainly unscheduled. This can cause poor resource utilization resulting in delayed execution, prolonged waiting time, and low system throughput. Recently, non-volatile RAM have been introduced as burst buffers to reduce the pressure on the underlying storage levels. However, they are not always able to avoid I/O contention when different applications compete for the storage resources. Current open problems include characterizing the dynamics of I/O accesses in HPC platforms and providing inter-application coordination capabilities to a high-level scheduler in order to reduce the I/O contention.

In this work, we explore new strategies for improving the system performance based on a closer cooperation between the I/O software stack, the scheduler, and the running applications. We present a cross-layer solution that includes a unified monitoring of the application's computational and I/O demands, an application scheduler, an I/O scheduler, and several strategies based on the coordinated use of I/O scheduling and application malleability. As far as we know, this is the first work that integrates these functionalities into a single framework. The aim of this work is to develop, and make available, a prototype that can be used as a proof-ofconcept to assess new techniques that contribute to enhancing the design of HPC infrastructures. The goal is to minimize the overall execution time, or makespan, for the executing jobs.

The prototype integrates two dynamic runtimes CLARISSE (Isaila et al., 2016) and FlexMPI (Martín et al., 2015) into a common execution framework. CLARISSE is a middleware for enhancing I/O flow coordination and control in the HPC I/O stack software. CLARISSE decouples the policy, control, and data layers of the I/O stack software in order to simplify the task of globally coordinating the parallel I/O on large-scale HPC platforms. Flex-MPI is implemented as a library on top of MPICH (Amer et al., 2017) and provides dynamic load balancing and malleability capabilities to MPI applications.

As an overview, the proposed framework analyzes the executing applications, takes decisions for enhancing the system performance, and applies these decisions by means of control mechanisms. The applications are analyzed using the monitoring components of CLARISSE and FlexMPI. These modules send the data to an external framework that gathers the information from all

applications executed in the system considering the CPU, communication, and I/O performances. Following this, two complementary approaches are used to improve the system performance.

The first approach provides cross-application I/O coordination to the executing applications. It aims to reduce the number of I/O interferences between executing applications by means a combination of malleability and I/O scheduling techniques. We define an *I/O interference* (or interference) as multiple I/O operations performed by different applications that are competing for the same resources (I/O servers) at the same time. Note that because of the large amount of potential traffic involved in the I/O operations, I/O interferences may produce bottlenecks in the I/O software stack (Yildiz et al., 2016).

The second approach is an elastic scheduler that aims to enhance the overall system performance by selectively assigning the platform's unused computational resources to the executing applications by means of malleability. The criteria for distributing the resources are based on the application characteristics, including the application I/O intensity levels.

Finally, based on the actions required by the external framework, control commands are sent to the applications in order to increase/decrease the number of processes, and schedule the I/O phases.

In this work we provide an evaluation and validation of our framework in an HPC cluster using different experiments. Our proposed techniques are compared against other existing alternatives, including those commonly used in production HPC facilities as well as more advanced solutions. Results show that our approach improves on previous alternatives, enhancing the application performance as well as improving the overall system utilization by means of a reduction in the number of I/O interferences.

The rest of this paper is organized as follows. Section 2 provides a motivation as well as the background of this work. Section 3 provides an overview of the proposal while Section 4 describes in greater detail the design of the unified framework. In Section 5 we present an extensive performance evaluation. Section 6 discusses the existing state of the art, and Section 7 summarizes the conclusions and future work.

2. Motivation and background

The main motivation of this work is to take advantage of the unused computational resources of a cluster infrastructure. These resources (compute nodes) may be available because the system utilization does not reach the 100%, or because the queued applications, that are waiting to be executed, require more resources than those available. We propose assigning these resources to the executing applications by means of malleability in a temporary or permanent way. The novelty of this work is that this assignation is based on a holistic view of the platform, including in the analysis the CPU, communication and I/O application behaviors.

We define *application configuration* , as the state of a given application executed with a given number of processes. In a similar way, *application reconfiguration* involves changing the application state to a new one with a different number of processes (that can be bigger or smaller than the original one) by means of a malleable operation. Note that an application reconfiguration implicitly involves a data redistribution and a load balance operation (both performed transparently by FlexMPI) to ensure the efficient execution of the program.

In this work we focus on SPMD applications with periodic I/O. Many scientific applications fit in this category. These applications usually show bursty I/O behaviors and alternate between computation and I/O phases with a reduced overlap between them. Consequently, the compute nodes and the I/O subsystem may be idle for significant periods of time. Algorithm 1 shows an example of a SPMD legacy application that does not use our framework. Some of the function arguments are omitted for the sake of clarity. In this simplified example, the shared data is a distributed single-dimensional array (denoted as A) in which each process has assigned consecutive blocks of entries ranging from A[displ] to A[displ+count-1] where displ and count are local variables for each function. The iterative code section alternates CPU (line 10), communication (line 12), and I/O phases (lines 13–18). In this example MPI_Allgatherv() is used for the communication although other collective/non-collective operations like MPI_Reduce() or MPI_Send()/MPI_Recv() could be used instead. In this work we assume, without loss of generality, that the I/O operation represents a checkpoint, where the application state and data are stored on disk by means of a MPI write collective operation. We also assume that the duration of the CPU phases is much longer than the I/O phases. The Evaluate_Termination() function (line 19) is responsible for detecting whether the problem's solution has been reached with acceptable precision. If so, the program finalizes the execution.

Algorithm 1 Example of pseudocode of a SPMD MPI legacy code.

```
1: // Initializing section
2: MPI_Init( . . .)
```

```
3: MPI_Comm_rank( . . .)
4: MPI_Comm_size( . . .) 5: Initialize_data(A,displ,count, . . .)
6:
7: // Iterative section
8: for (it=0; it < itmax ; it++) do
9:     for (i=displ; i < displ+count; i++) do
10:        Compute(A[i], . . .)
11:    end for
12:    MPI_Allgatherv( . . .)
13:    if (it % 100==0) then
14:        MPI_File_open( . . .)
15:        MPI_File_set_view( . . .)
16:        MPI_File_write_all( . . .)
17:        MPI_File_close( . . .)
18:    end if
19:    Evaluate_Termination( . . .)
20: end for
21: MPI_Finalize( . . .)
```

Algorithm 2 illustrates how the previous code is instrumented with CLARISSE and FlexMPI. In the legacy code all the MPI specific functions are managed by the MPI library. The new parallel code is instrumented with a set of functions to get the initial partition of the domain assigned to each process (line 5) and register each of the data structures managed by the application (line 6). Registering is necessary to know which data structures should be redistributed every time a reconfiguring action is carried out. Function EMPI_Get_shared_data() is used by the newly spawned processes to receive the corresponding domain partitions that have been assigned. The iterative section of the code is instrumented to monitor each process of the parallel application (line 12). Function EMPI_Monitor_end() , line 23, evaluates whether it is required to create or remove any processes. Then each process checks its execution status (line 24) and in case of being removed, it leaves the iterative section (line 26) and terminates the execution.

Algorithm 2 Example of pseudocode of a SPMD MPI code integrated with CLARISSE and FlexMPI runtimes.

```
1: // Initializing section
2: MPI_Init( . . .)
3: MPI_Comm_rank( . . .) 4: MPI_Comm_size( . . .) 5: EMPI_Get_wsize( . . .)
6: EMPI_Register( . . .)
7: EMPI_Get_shared_data( . . .) 8: Initialize_data(A,displ,count, . . .)
9:
10: // Iterative section
11: for (it=0; it < itmax ; it++) do
12:     EMPI_Monitor_init( . . .)
13:     for (i=displ; i < displ + count; i++) do
14:         Compute(A[i], . . .)
15:     end for
16:     MPI_Allgatherv( . . .)
17:     if (it % 100==0) then
18:         MPI_File_open( . . .)
19:         MPI_File_set_view( . . .)
20:         MPI_File_write_all( . . .)
21:         MPI_File_close( . . .)
22:     end if
23:     EMPI_Monitor_end( . . .)
24:     status=EMPI_Get_status(…)
25:     if (status==REMOVED) then
26:         break;
```

```
27:     end if
28:     Evaluate_Termination( . . .)
29: end for
30: MPI_Finalize( . . .)
```

This work is based on the observation that two different applications performing the I/O phases at the same time, may produce a performance degradation in the I/O subsystem because of producing simultaneous accesses to the same resources. In order to analyze the I/O interference we have developed a synthetic benchmark that is synchronized with the CLARISSE central controller by means of a control signal. When the signal is received, the benchmark performs an I/O collective operation. Based on that, we have set up a use case with two identical benchmarks that perform the I/O with different overlap levels.

Let $T_{startIO}^{1}$ and $T_{startIO}^{2}$ be the I/O start time for application 1 and 2, respectively. We define $T_{delay} = | T_{startIO}^{1} - T_{startIO}^{2} |$ as the time interval between the beginning of the two I/O phases. Fig. 1 shows the application I/O time for different T_{delay} values for an execution with 20 processes per application and a file size of 24.4 GB. The tests were carried on the cluster depicted in Section 5 . The I/O time for a stand-alone application is around 8 s. When $T_{delay} = 0$ both I/O phases start at the same time and the I/O time is nearly doubled. These times gradually decrease until reaching the minimum values for T_{delay} . Based on these results we can conclude that the I/O interference has a strong impact on the I/O performance for this architecture. Similar results were obtained for different file sizes and numbers of processors. For example, two simultaneous writes on a file of 11.9 GB ($T_{delay} = 0$) take 9.2 s per application. For $T_{delay} = 10$ the I/O time is 4.7 s per application. Similar results were obtained by Dorier et al. (2014) and Yildiz et al. (2016).

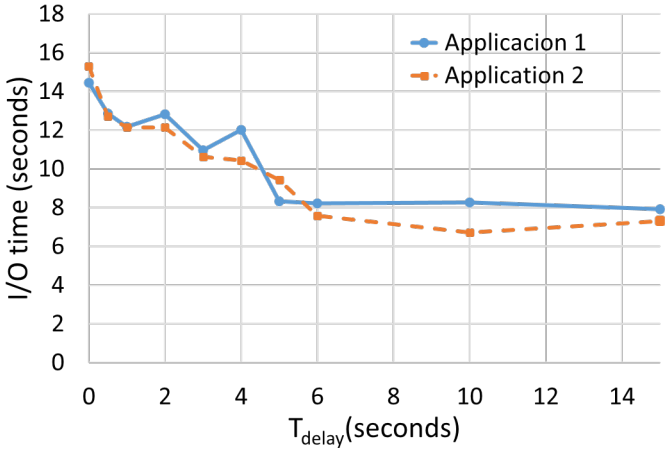


Fig. 1. Execution time of the I/O phase of two interfering applications.

Current HPC platforms address the I/O interference problem by means of I/O scheduling mechanisms that operate at request-level. However, this low-level approach is not able to leverage the application characteristics nor provide I/O access coordination mechanisms to the executing applications. Consequently (as our experiments confirm), despite these efforts, I/O interference still incurs performance degradation. This paper presents a framework that extracts the properties of the executing applications and provides inter-application coordination mechanisms for avoiding I/O interferences. We do not consider queued jobs, restricting our study to executing applications. The two strategies presented in this work are complementary and address this problem from two different perspectives.

3. Framework overview

In the proposed framework, the MPI applications (shown in Fig. 2) transparently use FlexMPI and CLARISSE logic for I/O and global system optimization. This is achieved by intercepting selected MPI calls through the PMPI profiling mechanism of MPI and inserting control points that implement the logic of distributed control algorithms. For CLARISSE, the MPI I/O routines (arrows 1 in Fig. 2) are

wrapped by CLARISSE's Control Point Logic in order to optimize I/O, using the CLARISSE's implementation of the I/O calls (Isaila et al., 2016). Note that CLARISSE's implementation produces the same results as the original MPI call, but in a more efficient way.

Following a similar approach, when an MPI routine is wrapped by FlexMPI (arrows 3 in Fig. 2), some actions are performed by the library (Martín et al., 2015) and subsequently, the corresponding MPI routine is called through the PMPI interface. This scheme permits us to execute the library-related actions in a transparent way, while preserving the original MPI call behavior. For example, the MPI_Init() and MPI_Finalize() routines are wrapped and used to initialize and terminate the internal components of our framework. MPI communication routines are also intercepted by FlexMPI and used to collect different performance metrics. Finally, some application's MPI calls (arrow 2 in Fig. 2) are not intercepted by the libraries and are directly executed by MPI. Examples of these are synchronization and datatype-management routines.

For each parallel application, CLARISSE and FlexMPI employ two application controllers (one per library), that are executed associated to the rank-0 process. These controllers are responsible (arrows 4 and 5 in Fig. 2) for sending the monitoring data to the performance modeler, including the type and timestamps of the I/O and communication operations, and receiving different commands from CLARISSE and FlexMPI central controllers.

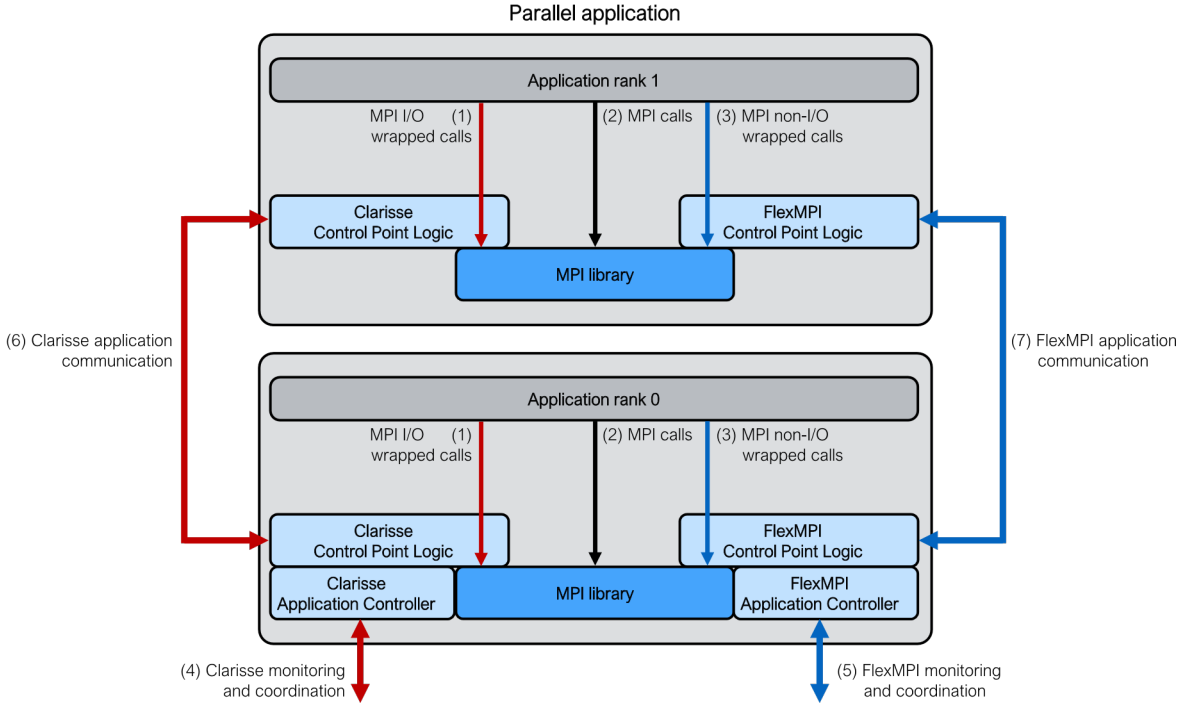


Fig. 2. Diagram with the integration of CLARISSE and FlexMPI runtimes into a MPI application consisting of two processes.

The control point logic of each library is responsible for coordinating all the processes of the application (arrows 6 and 7 in Fig. 2). This coordination includes collecting monitoring data from the whole processes into the application controller, and broadcasting control information received by the application controller.

Fig. 3 shows the proposed unified execution framework for CLARISSE and FlexMPI: on the upper-left part of the figure there are two parallel applications using four and two processes. Here we show a simplified scheme for the applications that only displays the application process and the two application controller threads. For the sake of simplicity, we do not represent the MPI library nor the CLARISSE and FlexMPI control point logics.

The application monitoring data is sent (arrow 1) to the Performance Modeler. The Performance Modeler is responsible for aggregating the data and generating models that predict the performance of each application on different configurations. The I/O Interference Controller and the Global Performance Controller use this information (arrows 3a and 3b) to find the best configuration for the running applications in order to meet two different performance objectives: the first objective (implemented in the I/O Interference Controller) is to reduce the I/O time by mitigating the I/O interference. The second performance objective

(implemented in the Global Performance Controller) is to maximize the global system performance considering the performance metrics, as well as the achieved speedups, of each application that is being executed.

The Resource Manager is responsible for assigning the unused computational resources (compute cores) to the newly executed applications (arrow 8) or to the ones that are already being executed but with a number of processes that has been changed by means of malleability (arrows 5a and 5b). Note that the controllers only specify how many processes have to be created or destroyed, and the Resource Manager is responsible for determining, for each reconfiguration, which specific cores must be allocated or released. This information is sent to CLARISSE and FlexMPI controllers (arrow 6), and they translate them into commands that are sent to each application (arrows 7a y 7b). Examples of these commands are, for CLARISSE, to enable or disable the Parallel I/O scheduling policy, and for FlexMPI to spawn or destroy certain application processes into specific compute nodes.

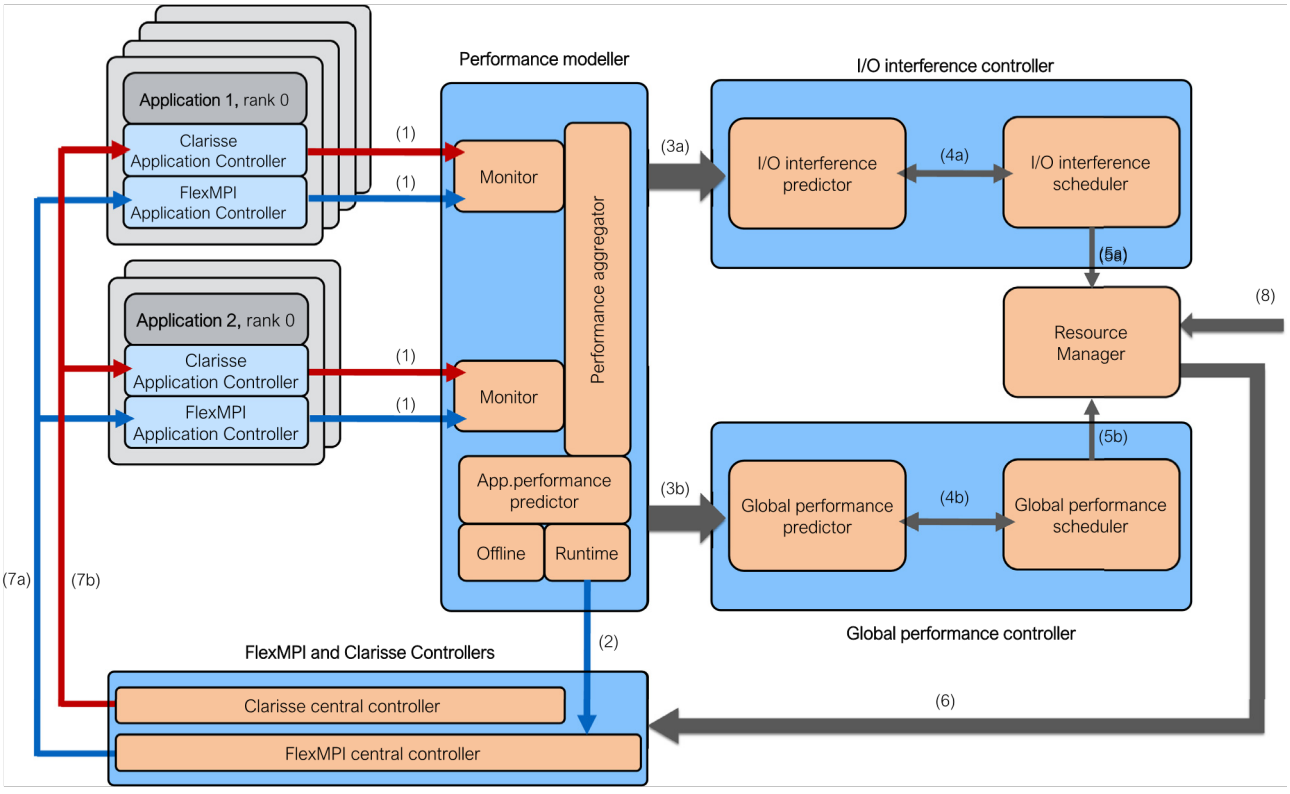


Fig. 3. Unified execution framework for CLARISSE and FlexMPI.

4. Framework structure

This section presents a more detailed description of the framework components and describes how they operate.

4.1. *Performance modeler*

The performance modeler is responsible for aggregating monitoring metrics from both CLARISSE and FlexMPI, providing a unified view of the application performance. For a given application, each MPI I/O operation (like, for instance, MPI_File_write_all()) is wrapped by CLARISSE library, measuring its duration. When the I/O operation is completed, the CLARISSE Application Controller sends this data (arrows 1 in Fig. 3) to the associated Monitor, that records them together with its timestamp using the local timer. A similar approach can be used for the FlexMPI wrapped calls. In this way, the *Performance aggregator* keeps internal records (one per application) of all MPI communication and I/O operations, with their related durations and timestamps. These records are dynamically updated during the application execution.

For each application i, we define the *I/O period* ($P_{I/O}^{i}$), as the overall time between two consecutive I/O operations. In our experiments we have considered applications that periodically perform collective MPI writes, assuming a behavior similar to checkpointing operations. For simplicity, we assume that there is a single I/O collective operation in each I/O period. This period is obtained as Eq. (1) shows, by means of the addition of all intermediate CPU, communication, and I/O times between two I/O phases of application i, named T_{cpu}^{i}, T_{comm}^{i}, and $T_{I/O}^{i}$, respectively.

$$P_{Ii/O} = T_{cpui} + T_{commi} + T_{Ii/O} \quad (1)$$

The Application Performance Predictor predicts the performance of each application under different configurations. Our approach uses both off-line and on-line prediction. The off-line prediction is done by FlexMPI based on a performance-analyzer that evaluates the application as a benchmark, testing different configurations and measuring the related CPU, communication, and I/O times. The performance metrics are stored in a database and subsequently read when the application is executed.

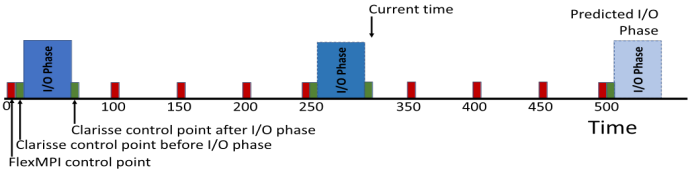
The on-line predictor uses the Runtime component of the Application Performance Predictor. At the beginning of the application execution, this component sends FlexMPI commands (arrow 2 in the Fig. 3) for reconfiguring the application for a different range of processes. For each configuration, several performance measurements are done. After that, the initial application configuration is restored. Note that the only effect of these actions on the application are changes in its execution time during the evaluation period. In both alternatives (off-line and on-line), only some representative application configurations are evaluated (for instance, using a number of processes multiple of 10), and subsequently, an interpolation model is used to estimate the metrics for the intermediate cases. Note that the number of samples depends on the application characteristics. Some applications (like the ones used in our experiments) have a slight variation of performance, thus only few samples are required to precisely model them. In other cases, with more complex behaviors, a greater number of samples should be required.

Both the Performance Aggregator and Application Performance Predictor are executed within a single thread. In contrast, each monitor is executed in a separated thread. Consequently, the performance modeler employs as many threads as the number of running applications plus one.

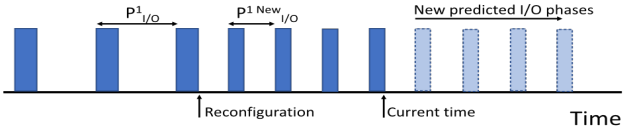
4.2. *I/O interference controller*

This controller is responsible for detecting and mitigating I/O interference with the aim of reducing their effect on the system performance. The controller consists of two components: the I/O interference predictor and the I/O interference scheduler.

The I/O interference predictor analyses the performance modeler records (arrow 3a in the Fig. 3) and predicts, both in time and duration, the forthcoming I/O requests. We assume that the I/O operations occur periodically, thus the predicted values can be obtained by extrapolating the average values of the period and duration of the previous I/O operations. Fig. 4 (a) shows an example in which each bar represents an I/O phase with a given starting time (x-axis value) and duration (bar width). The space between bars corresponds to program phases not related to I/O (CPU, communication, etc.).



(a) Example of I/O phase prediction



(b) Example of reconfiguration

Fig. 4. Graphic example of the prediction and reconfiguration processes.

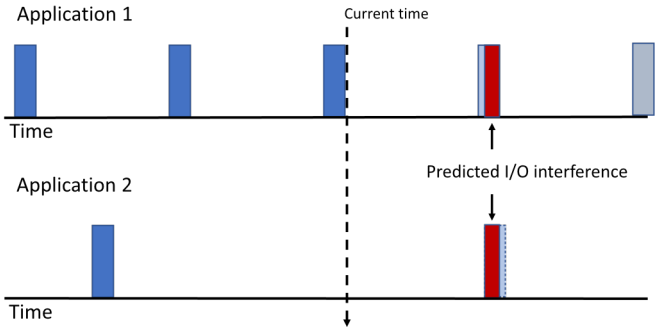


Fig. 5. Graphic example of I/O interference detection.

Small red bars correspond to FlexMPI control points, that are executed every certain number of iterations. In this example, there is one I/O phase every 400 iterations and a FlexMPI control point every 100 iterations (iteration values are not shown in the figure). Small green bars are CLARISSE control points, that are only reached before and after the I/O phase, that is, every 400 iterations. Note that the execution interval of the FlexMPI control point that contains the I/O phase is longer (100-time units) than the others (50-time units). This is due to the extra time related to the I/O operation. Note that during the control points, the application is monitored and when the application is reconfigured, the predicted values are recomputed considering the new configuration performance. Fig. 4 (b) shows an example for a reconfiguration that increases the number of processes, reducing both the period (from PI^1/O to $PI^1/^{New}O$) and duration of the I/O operations. For simplicity, the I/O phases are only shown as blue rectangles. In this case, the Application Performance Predictor is able to update the prediction of the new incoming I/O phases after monitoring the new reconfigured state.

The I/O Interference Predictor detects future I/O interference by computing the intersection between the I/O predictions of all applications that are being executed. Fig. 5 shows an example of this detection procedure for two applications. We can observe that application 1 has a smaller I/O period than application 2 and that the first predicted values overlap in time, producing a potential I/O interference (red bars). Note that the overlap can be total or partial (like in this example). The I/O interference scheduler module uses this information (arrow 4a in the Fig. 3) to avoid I/O conflicts by means of malleability. Two different strategies, called *Phase Shifting* and *Period coupling* , have been implemented in this module and are described next.

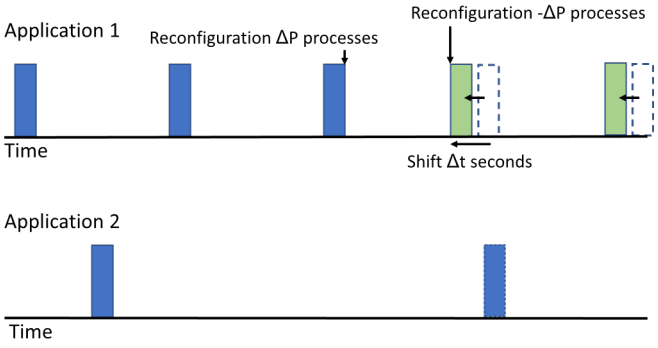


Fig. 6. Graphic example of Phase Shifting algorithm for avoiding an I/O interference.

4.2.1. *Phase shifting strategy*

This strategy introduces a shift in the I/O phase of one of the applications. This is done by means of a reconfiguration that temporarily increases the number of processes. Fig. 6 shows a graphic example of this operation for the previous example for Fig. 5 . Now, the I/O interference is avoided by increasing Application 1 by P processes. Green bars display the I/O phases for the new configuration. The idea behind this strategy is to use resources (compute cores) during a short period of time, releasing them after the interference is avoided. Consequently, after avoiding the I/O conflict, the application's original configuration is restored by changing the number of processes by $-P$.

For this strategy, in case of I/O conflict, the I/O interference scheduler receives as input the predicted overlap time $T_{Overlap}$ of the next I/O phase and the identification of the applications (i and j) involved in this interference. The scheduler pseudocode is shown in Algorithm 3 . The first step of the algorithm is to decide, by means of function *select_application()* , the application k that has to be reconfigured. Based on this, T_{shift} is obtained by means of *obtain_shift()* function. This is the amount of time that the I/O phase of the reconfigured application k has to be shifted for avoiding the I/O interference. Note that k can be either i or j . The criterion for selecting the k value depends on PI^{i}/O and PI^{j}/O, the I/O periods of applications i and j . If the periods are similar but not equal, then the application with the shortest period will be the reconfigured one. In this way, for the following I/O phases, the reconfigured application will execute the I/O phase earlier than the other, and consequently, the distance between the application I/O phases will increase over time.

Algorithm 3 I/O interference controller pseudocode for Phase shifting strategy.

```
INPUTS: i and j, interfering application ids
k = select_application(i, j)
T^k_shift = obtain_shift(i, j, k)
cond = evaluate_interference(k, T^k_shift)
if !cond then
    ΔT^k_reconf = T^k_shift + 2 * Overhead^k + uncertainty
    S_rel = obtain_speedup(T^k_cpu, T^k_reconf)
    ΔProc = obtain_processes(S_rel)
    request_resources(ΔProc, Iter_reconf1)
    request_resources(−ΔProc, Iter_reconf2)
end if
```

Fig. 7 shows a graphic example of two applications with I/O periods (not shown in the figure) of PI^{1}/O and PI^{2}/O that are similar and an overlap of $T_{Overlap}$. In this example applications 1 and 2 start the I/O at times 100 and 80, respectively.

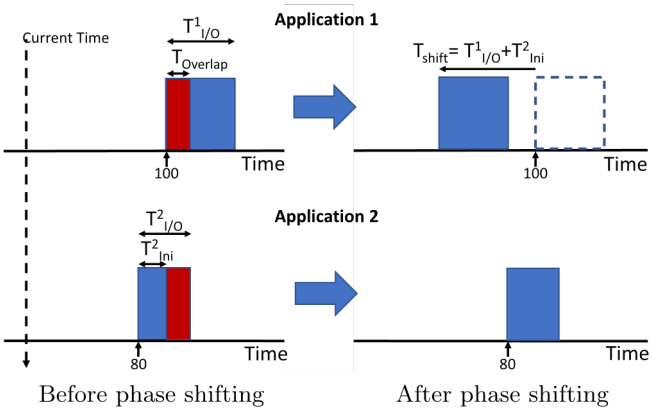


Fig. 7. Graphic example phase shifting strategy when PI^{1}/O is smaller than PI^{2}/O . Application 1 is reconfigured.

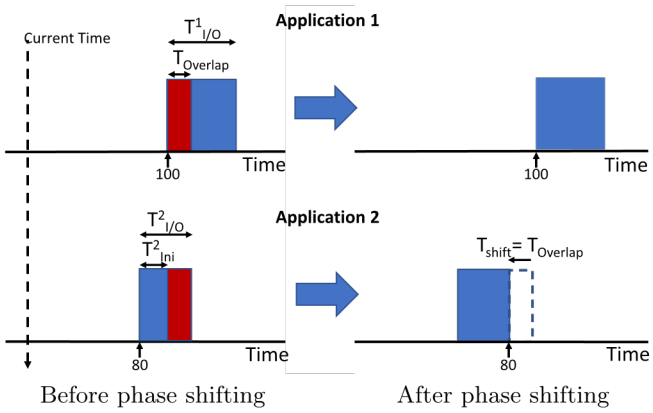


Fig. 8. Graphic example of phase shifting strategy when $PI^2_{I/O}$ is smaller than $PI^1_{I/O}$. Application 2 is reconfigured.

We assume that Application 1 has the shortest period $(P^1_{I/O} < P^2_{I/O})$, and consequently, is the one to be reconfigured. Given that its I/O phase happens after Application-2 I/O phase, $T_{shift} = TI^1_{I/O} + T_{ini}{}^2$, where $T_{ini}{}^2$ is the interval from the beginning of Application-2 I/O phase until the overlap. If Application 2 had the shortest period ($PI^1_{I/O} > PI^2_{I/O}$), then as Fig. 8 shows, T_{shift} will be the same as the amount of overlap. If the I/O periods are equal or very different, the one with the I/O phase that occurs first will be reconfigured, minimizing the amount of shift that has to be applied.

Function *evaluate_interference()* queries to the I/O interference predictor whether the phase shifting of $T_{shift}{}^k$ for application k may produce any interference with the rest of the applications. If the condition is false (no interference), then the optimization continues. It first obtains (line 4) the overall time that the application must be shortened T_{reconf} that includes the shift time plus two times the reconfiguration overhead, plus a predefined uncertainty. The overhead term compensates the overhead of the reconfiguration operations (note that this strategy involves two reconfigurations, one for increasing and another one for decreasing the number of processes), so it does not change the application's I/O period. The uncertainty term considers possible errors in the prediction, so the shift time is increased by an extra amount, that in our experiments is 2 seconds.

For instance, given a $T_{shift}{}^k$ of 10 s, and a reconfiguration overhead of 1 s, then T_{reconf} will be 14 s, that corresponds to the CPU reduction time. However, the actual phase shift will be a reduction of 12 s given that the period includes the overhead of two reconfiguration operations of 2 s in total. Note that if the final value of T_{reconf} is greater than the needed one (because of the uncertainty), the I/O conflict will also be avoided.

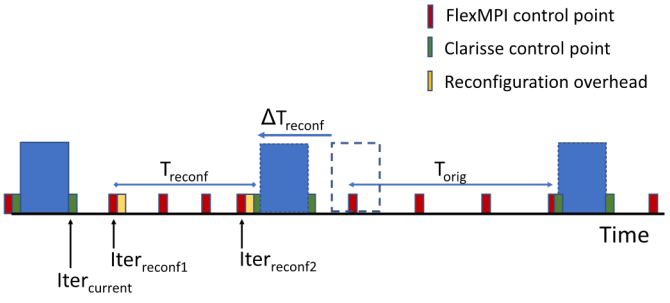


Fig. 9. Graphic example of the existing control points for the Phase shifting algorithm.

Fig. 9 shows a graphic example of the reconfiguration process. X-axis represents the time and includes three milestones. When the first I/O phase completes, CLARISSE control point notifies the I/O operation at application iteration $Iter_{current}$. Based on that, the reconfiguration decision is taken by the I/O interference controller and the application is reconfigured at the following FlexMPI control point, at iteration $Iter_{reconf1}$. The application keeps this new configuration until the last FlexMPI control point before the next I/O phase, at iteration $Iter_{reconf2}$, when the original configuration is restored. Note that in this interval of time, the application execution time is T_{reconf}, a smaller value than the original (not reconfigured) time of T_{orig}. Note that $T_{reconf} = T_{orig} - T_{reconf}$ and that T_{orig} is provided by the Performance Modeler.

The next step obtains the application relative speedup necessary to produce the required I/O phase shift by means of *obtain_speedup* () function. This function uses Eq. (2) to compute the speedup. Note that the more distant the interference is, the smaller speedup is required, and less computational resources are involved, given that there is more time to introduce the required shift. For instance, let's assume that T_{orig}^{k} is 100 s and T_{reconf}^{k} 50 s. Then, the required speedup will be 2. However, if T_{orig}^{k} 200 s the same shift will require a speedup of 1.33. Also note that this speedup is relative to the current configuration. For example, if we are executing the application with a certain number of processes, that has an associated speedup (relative to the sequential version) of 6, and the required relative speedup given by Eq. (2) is 2, we will need an absolute speedup of 12 (respect to the sequential version) to fulfill this requirement.

$$S_{rel} = \frac{T_{orig}^{k}}{T_{reconf}^{k}} = \frac{T_{orig}^{k}}{T_{orig}^{k} - \Delta T_{reconf}^{k}} \qquad (2)$$

Once the relative speedup is computed, the I/O interference scheduler uses the application's performance model provided by the performance predictor (*Obtain_processes()* function) to obtain the number of new processes (P) necessary to reach it. By means of the *request_resources()* function, a request is sent to the Resource Manager, to allocate P cores for the application (arrow 5a in the Fig. 3). The Resource Manager evaluates if the resources are available. In the event of it being impossible to perform the reconfiguration, given that, for instance, the application does not reach the required speedup or the required resources are not available, reconfiguration is cancelled. Otherwise, the Resource Manager allocates the required resources and sends to FlexMPI central controller (arrow 6 in Fig. 3) a command including the list of processes per compute node that the application has to spawn, and the iteration where the reconfiguration has to be performed ($Iter_{reconf1}$).

This command is received by FlexMPI central controller, that converts it into an internal FlexMPI command that is sent to the reconfigured application k (arrow 7a in Fig. 3). The application's FlexMPI controller executes this command in the next control point that the application executes, triggering the application reconfiguration.

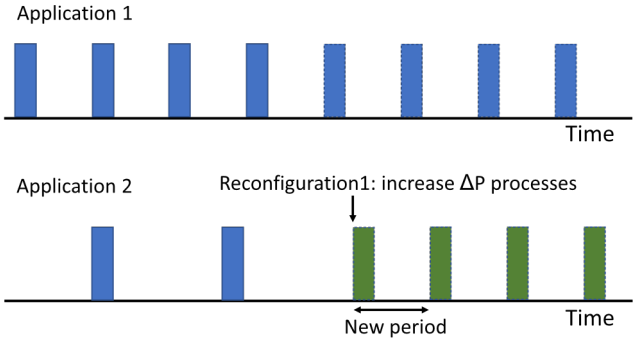


Fig. 10. Graphic example of Period Coupling algorithm for avoiding I/O interference.

Note that it is necessary to keep this new configuration for a certain time until the initial one is restored. Consequently, a new command is generated from the Resource Manager to reconfigure $-P$ processes at the iteration ($Iter_{reconf2}$). The effect of this command is to produce a delayed reconfiguration that is held by the FlexMPI application controller and applied when the specified iteration is reached. With this approach (instead of waiting a certain amount of time) it is possible to precisely determine the reconfiguration time, given that the iteration of the I/O interference can be exactly calculated. After the second reconfiguration, an acknowledgment command (not shown in the Fig. 3) is sent back to the Resource Manager, to permit it to release the resources (cores) temporarily assigned to the application.

4.2.2. *Period coupling strategy*

The idea behind this strategy is to reconfigure one of the applications in order to obtain a similar I/O period between both of them. Fig. 10 shows a graphic example of how this operation works. We can observe that Application 2 is reconfigured to obtain an I/O period similar to Application 1 (green bars). Note that in this strategy the reconfigured application uses the additional CPU resources in a permanent way, and a second reconfiguration that restores the original number of processes is not necessary. It is worth

mentioning a particular case where two applications have similar periods and both I/O phases occur at the same time. This leads to the worst scenario, with a potential interference for each I/O phase. However, this situation is easily avoided combining this strategy with the Phase Shifting strategy, which introduces a shift that avoids conflicts.

The scheduler pseudocode is shown in Algorithm 4 . Note that, like in the previous strategy, it is only executed when two applications are predicted to generate I/O conflicts. For two conflicting applications and j , function *select_largest_period()* obtains the one, k , with the larger I/O period. Note that k can be either i or j . Function *obtain_period()* returns the desired period for application k . This period is the nearest multiple value of the other one. For instance, if application 1 has a period of $P = 220$ s and application 2 has a period of $P_{I/O} = 440$ s, then application 2 is the reconfigured one, so $k = 2$ and P s. Once the new period is obtained, the required speedup is computed obtaining the ratio between the old and new periods. As with the previous strategy, functions *obtain_processes()* and *request_resources()* obtain the number of required processes and generate the request to the Resource Manager, respectively. If the resources are available, then the reconfiguration is performed for the current iteration ($Iter_{current}$). This strategy works in an interactive way, thus if the obtained period is larger or smaller than the required one, subsequent reconfigurations adjust its value to meet the requirements. Finally, once the application period is adjusted, the Phase Shifting Strategy is activated to avoid further I/O interference.

Algorithm 4 Performance model pseudocode for Period coupling strategy.

1: $k = select\_largest\_period(i, j)$
2: $P^k_{I/O-new} = obtain\_period(k)$
3: $Relative\_Speedup = P^k_{I/O}/P^k_{I/O-new}$
4: $\Delta Proc = obtain\_processes(Relative\_Speedup)$
5: $request\_resources(\Delta Proc, Iter_{current})$
6: $Activate\_Phase\_Shifting\_strategy()$

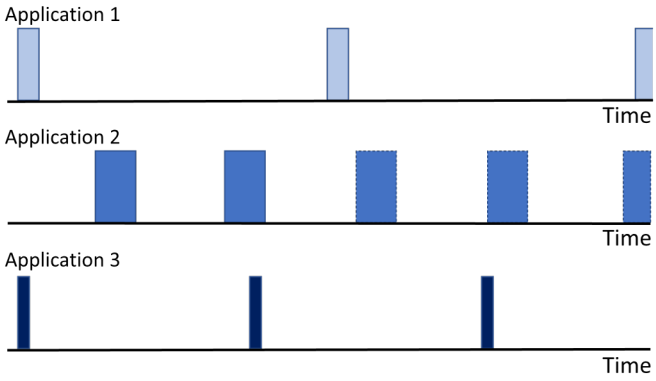
4.3. Global performance controller

The idea behind this approach is to assign as many computational resources as possible to maximize overall system performance. The optimization criteria that we have used is to maximize the aggregated FLOPS between all running applications and minimize the number of I/O delays. An I/O delay is introduced by CLARISSE's Parallel I/O scheduling policy (Isaila et al., 2016) when an I/O interference is produced. In this case, only one interfering application is allowed to perform its I/O phase and the remaining ones are blocked until the I/O subsystem becomes available. This permits the applications to perform exclusive I/O access, improving the system performance at the expense of introducing I/O waiting times (called delays). In this context, we assume that the Global performance controller is used in combination with CLARISSE's Parallel I/O scheduling. This controller consists of two components: the Global Performance Predictor and the Global Performance Scheduler.

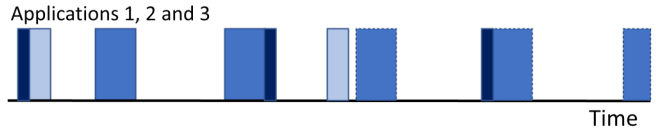
4.3.1. Global performance predictor

We define { A 1 , A 2 , ... A n } as the set of n applications that are initially running with { P 1 , P 2 , . . . , P n } processes. This compo- nent is responsible for evaluating the system performance for a new configuration where each application is reconfigured adding { ΔP 1 , ΔP 2 , . . . , ΔP n } processes to the original configuration. Note that ΔP i can be either a positive, zero, or negative integer value, that means that the application has spawned new processes, it has not been reconfigured or it has reduced the number of processes, respectively. An example of negative ΔP i values occurs when an application that was previously reconfigured, receiving additional computational resources by the Resource manager, has to release them, because they are required by a new executing application. Unlike the I/O interference predictor, the goal here is to predict the I/O ratio of application i (IOR i) defined as the ratio between the I/O time and the I/O period (Eq. (3)), for a given number of processes P i + ΔP i . The Global performance predictor uses metrics generated by the performance modeler (arrow 3(b) in Fig. (3)) for generating the I/O ratio prediction.

$$IOR^i = \frac{T^i_{I/O}}{P^i_{I/O}} = \frac{T^i_{I/O}}{T^i_{cpu} + T^i_{comm} + T^i_{I/O}} \quad (3)$$

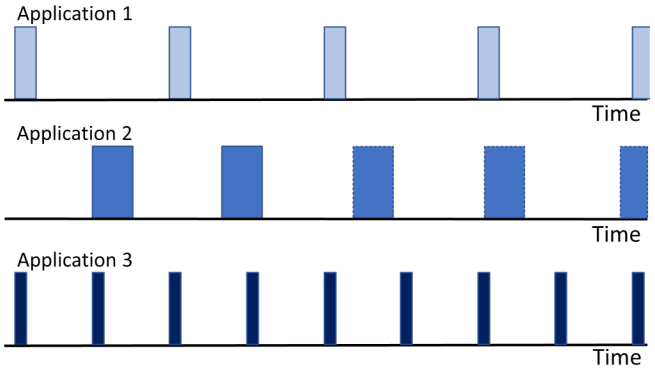


(a) Independent application execution

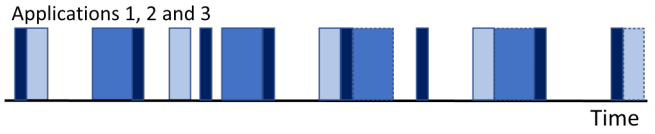


(b) Combined application execution

Fig. 11. Graphic example of the effect of combining the execution of applications.



(a) Independent application execution



(b) Combined application execution

Fig. 12. Graphic example of the effect of combining the execution of reconfigured applications. Application 1 and 3 have reduced the I/O perid to 50% and 20% respectively.

4.3.2. *Global performance scheduler*

The Global performance scheduler implements three different policies. The first one is the Fair scheduling that evenly distributes the available cores between the running applications. This policy represents the baseline case. The second policy, named *Speedup-based scheduling*, aims to assign the available cores to the applications with greatest scalability.

Algorithm 5 describes how it works. Variable $avail\_cores$ is the existing number of available cores. Firstly, all elements P^i are set to zero. The policy first computes for each application i, the speedup improvement SP^i, defined as the speedup increment between the current application configuration and the one resulting from spawning one additional process. The speedup information is provided by the application performance predictor. The application j with the biggest SP is selected for being reconfigured ($select\_max()$ function), and consequently, P^j is incremented by one. This procedure is repeated in an iterative fashion until all available cores have been assigned. Note that for each application i, the speedup evaluation considers the current number of processes $(P^i + \Delta P^i)$ that are being evaluated, so a given application may not receive additional resources if the new speedups are not as good as the ones achieved by the rest of the applications. When all the resources have been assigned, the $request\_resources()$ sends a request to the Resource Manager (arrow 5b in the Fig. 3) to reconfigure each application i that has $P^i > 0$.

Algorithm 5 Speedup-based scheduling pseudocode.

```
1: ΔP^i = 0, ∀i = 1, n
2: while avail_cores > 0 do
3:     ΔSP^i = SP^i_{P^i+ΔP^i+1} − SP^i_{P^i+ΔP^i}, ∀i = 1, n
4:     j = select_max(ΔSP)
5:     ΔP_j = ΔP^j + 1
6:     avail_cores = avail_cores − 1
7: end while
8: request_resources(ΔP^1, ΔP^2, ..., ΔP^n)
```

The last policy, named *IO-based scheduling*, is an improvement of the previous one that considers both the speedup and I/O ratios. Fig. 11 (a) shows an example of three applications with different I/O frequencies and intensities. When they are executed at

the same time (Fig. 11 (b)), the I/O conflicts are now dealt with by CLARISSE's Parallel I/O scheduling policy, that ensures that the I/O accesses of each application are exclusively performed at the expense of introducing delays in some of them. Fig. 12 (a) shows the result of reconfiguring applications 1 and 3, reducing the I/O periods to 50% and 25%, respectively. When they are simultaneously executed (Fig. 12 (b)), the number of conflicts increases as well as the amount of I/O delays introduced by CLARISSE. The idea behind this proposal is to use the available resources to selectively reconfigure the applications that have better speedups and less I/O intensity, in order to reduce the pressure on the I/O subsystem.

Algorithm 6 describes the IO-based scheduler structure that is similar to the previous case. The scheduler computes the speedup increment and the I/O ratio (IOR^i) for each executing application i . Then, by means of *select_balance()* function, the application j balances a small I/O ratio and a large speedup improvement is selected. This means that now the application with the best speedup improvement may not be selected if there is another application with a similar speedup improvement and a lower I/O ratio. Finally, the number of process of the selected application is increased by one and the procedure is repeated until there are no more available cores.

Algorithm 6 IO-based scheduling pseudocode.

$\Delta P = \{0, 0, \ldots, 0\}$
while $(avail\_cores > 0)$ **do**
  $\Delta SP^i = SP^i_{P_i+\Delta P_i+1} - SP^i_{P_i+\Delta P_i}, \forall i = 1, n$
  $IOR^i = compute\_IO\_ratio(i, P_i + \Delta P_i + 1), \forall i = 1, n$
  $j = select\_balance(IOR^i, \Delta SP^i, \forall i = 1, n)$
  $\Delta P_j = \Delta P_j + 1$
  $avail\_cores = avail\_cores - 1$
end while
$request\_resources(\Delta P^1, \Delta P^2, \ldots, \Delta P^n)$

4.4. *Resource manager and library controllers*

The Resource manager administrates the platform computational resources. It receives requests from the schedulers (arrows 5a and 5b in Fig. 3) for allocating or releasing a certain number of cores for a given application. It also receives requests from the new incoming applications (arrow 8) that need to be executed with a certain number of processes. In the latter case, part of the computational resources used by the I/O interference scheduler and the global performance scheduler may be released and reassigned to these new applications. This means that our proposal permits a flexible use of the unused resources, that can be reallocated when needed. During the allocation process, the Resource Manager maximizes the core locality, that is, it tries to allocate the new processes to the same compute nodes that the application is currently using (thus minimizing inter-node communications).

The CLARISSE Central Controller also receives commands from the Resource Manager (arrow 6 in Fig. 3). In the current version, the commands permit the activation or disabling of CLARISSE's Parallel I/O scheduling policy. After receiving this command, the CLARISSE Central controller sends the specific orders to each application (arrow 7b in Fig. 3). It is worth mentioning that this technique can be used in combination with the two strategies developed for the I/O interference controller and for the three scheduling algorithms of the global performance controller.

The FlexMPI Central Controller is responsible for sending the reconfiguration commands to the applications (arrow 7a in Fig. 3). In the case of increasing the number of processes, the command includes the list of processes (provided by the Resource manager) that each application has to create in each compute node. Otherwise, the command includes the specific processes (associated to certain compute nodes) that have to be removed.

5. Experimental evaluation

This section summarizes the main results obtained in the practical evaluation of our framework. We have tested our proposal in a Bebop cluster at Argonne National Laboratory. This platform is based on Intel Xeon Broadwell E5–2695v. Each compute node

consists of 36 cores (two processors per node) and 128 GB of memory. The compute network is 100 Gb/s Omni-Path Fabric Interconnect and the filesystem GPFS.

An implementation of the Jacobi algorithm was used as benchmark. Jacobi is an application which implements the iterative Jacobi method for solving systems of linear equations modeled by means of dense matrices. In our implementation, we have included an I/O phase that performs a collective I/O operation MPI_File_write_all() every certain number of iterations. In this collective operation each process stores local data in consecutive locations in a single file following a block distribution.

Jacobi was compiled with MPICH 3.2 distribution and gcc 4.8.5. In order to complete our experiments in a reasonable time, the duration of the CPU and I/O phases have been scaled down. Instead of considering I/O periods of many minutes or hours, and I/O durations of several minutes, we have used intervals of several seconds for both the period and duration of the I/O phases. Note that although these values are scaled, the ratio between the I/O period and duration is similar to real applications.

We have divided the experiments into two sections: the first one evaluates and analyzes the effectiveness of the I/O interference controller to reduce these interferences, and the second one evaluates the ability of the global performance controller to maximize the application performance. Note that both strategies are complementary and can be used independently or in a combined way.

5.1. *I/O interference controller evaluation*

Table 1 describes the different use cases that we have considered. In the experiments we have used four different execution scenarios. The baseline code corresponds to the original unoptimized (without CLARISSE and FlexMPI) execution scenario. This scenario is the existing one in current HPC platforms, where the application are executed without I/O coordination. The I/O blocking scenario includes CLARISSE's Parallel I/O scheduling policy (Isaila et al., 2016). This corresponds to a more advanced existing solution with I/O coordination but no malleability support, where each application acquires exclusive access to the storage, improving the overall I/O performance. The Phase shifting and Period coupling scenarios are one of the contributions of this work. They combine CLARISSE's Parallel I/O scheduling policy with Phase Shifting, and Period Coupling strategies, respectively.

Table 1
Benchmark configuration for each use case. The benchmark code is Jacobi with a matrix of 40,0 0 0 × 40,0 0 0 entries. The output file size is 11.9 GB.

Use case	N_{iter}	$App.1\ N_{procs}$	$App.2\ N_{procs}$
A	20 0 0	128	150
B	20 0 0	128	150
C	40 0 0	64	64
D	40 0 0	64	50

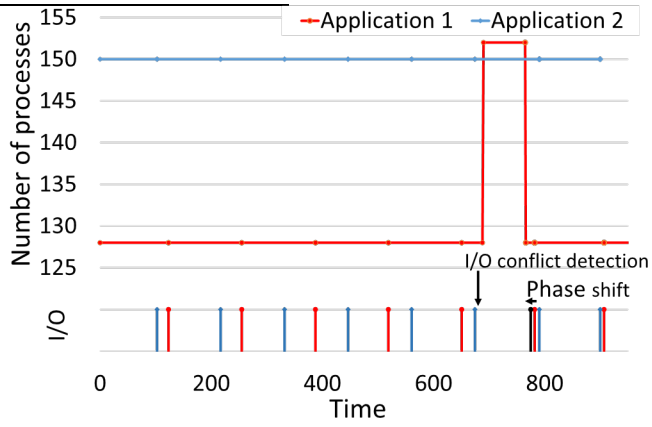


Fig. 13. Phase shifting strategy for use case B. The upper part of Y axis shows the application number of processes. The lower part shows the time stamps of the I/O phases.

Firstly we have evaluated the capabilities of the I/O interference predictor to detect future interferences. We executed the use case A of Table 1 (two applications of 128 and 150 processes) with the I/O interference controller disabled to let the interferences happen. At time 810 s, the I/O interference predictor detected an I/O interference at time-stamps of 903.2 and 911.0 s for each

application. The actual I/O time stamps were 903.0 and 911.9 s. Note that this prediction is made during CLARISSE's control point after the I/O phases (see Fig. 4). In general, the predictions are accurate because between the time when the prediction is made and the next I/O phase (when the interference arises) there are only CPU and communication phases, which have low variability when the application is executed on exclusive compute nodes. Note also that the uncertainty factor (2 s, in our experiments) takes into account small deviations in this prediction. For instance, in this example the considered I/O phase duration for application 1 was 10.49 s (instead of 8.49 s), and an I/O conflict was predicted even if it probably would not have happened. The use of uncertainty permits avoiding potential conflicts in a conservative manner, with no impact in the application performance. Note that the phase shift strategy always reduces the application execution time, thus it does not produce a performance penalty.

The second experiment (use case B in Table 1) evaluates the ability of the I/O interference controller to avoid conflicts. Fig. 13 shows the results of the execution of this use case. In the lower part of the figure, the times of the application I/O phases are highlighted whilst the upper part shows the number of processes used by each application. For the sake of clarity, the lower part only displays the starting time of each I/O phase.

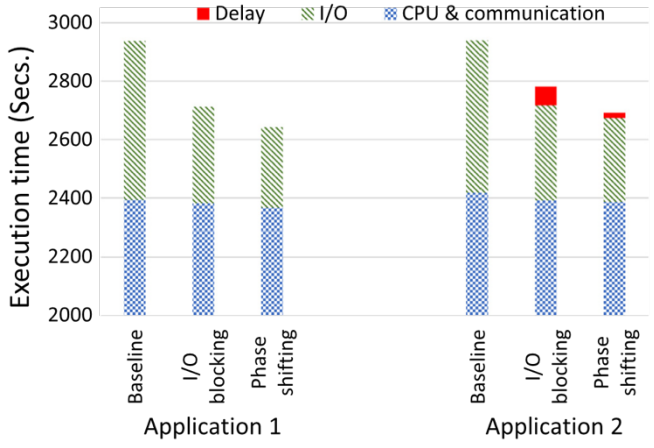


Fig. 14. Total execution time for use case C.

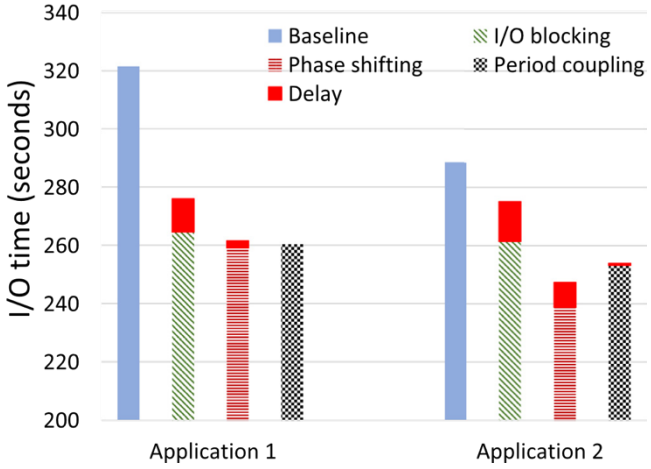


Fig. 15. I/O time for use case D.

We can observe that the I/O period of Application 2 is smaller than the period of Application 1, so the relative distance between the I/O phases changes over time. An I/O conflict is detected at T = 690 (after completing the I/O phase of Application 2) and Application 1 temporarily increases the number of processes by 24 to perform a phase shift. The I/O phase is advanced from 782.4 (displayed in red) to 779.0 s (displayed in black), thus avoiding the conflict. Following this, the original configuration of Application 1 is restored, and the next I/O phases are executed without producing further conflicts.

The next experiment (use case C in Table 1) shows the effect of the Phase Shifting strategy for two identical applications that execute the I/O phase at approximately the same time, producing a large number of I/O conflicts. Although unrealistic, this scenario is an example of close-to-maximum performance gains obtained with our proposal. We compare it against the baseline and block I/O scenarios. Fig. 14 shows the execution times for each program. We can observe that the I/O blocking implementation is able to reduce the I/O time but introduces some delays for Application 2. In total, this version introduces 64.7 s of delay. With the combined implementation, Application 1 is reconfigured creating 21 processes for 156.1 s (note that Application 1 CPU time is slightly smaller for the combined version). This introduces a phase shift that completely removes the I/O conflicts. The delay (18 s for application 2) is due to the existing I/O interferences before the reconfiguration. However, there is no delay after the reconfiguration operation. Note that the Period Coupling strategy is not applicable because both applications have the same period.

If we consider the I/O time exclusively (not shown in the figure) and the introduced delays, Application 1 I/O time is reduced by 39% and 49% for the I/O blocking and combined implementations, respectively. For application 2, this reduction is 25% for the I/O blocking and 41% for the combined implementation. These values consider both the actual I/O time and the waiting time (delays) introduced by the I/O scheduling. If we do not consider these delays, this reduction increases to 45%.

Use case D (see Table 1) consists of two versions of the benchmark executed with a different number of processes. We have repeated the experiments three times. Average values are displayed. Fig. 15 shows the I/O time for each application. The I/O

blocking strategy reduces the I/O time but introduces significant delays. The aggregated delay time is 25.1 s. Note that during this time one of the applications is blocked, wasting the existing computational resources. With the Phase shifting strategy most of the I/O conflicts are avoided, and the I/O time can be reduced as well as the amount of delays (now 11.0 s in total). On average, Phase shifting performs four reconfigurations per execution. Each reconfiguration involves creating and subsequently removing a certain number of processors that ranges between 14 and 104. The disparity in values depends on the amount of overlap and the time interval necessary to reach the predicted conflict. The bigger the overlap or smaller the time interval, the bigger the number of resources that are required to avoid the conflict. Finally, the period coupling strategy also reduces the I/O time with a smaller number of delays (0.7 s in total). In the three repetitions of the test, the period adjustment changed the number of processes of Application 2 to 57, 59 and 59 (again) processes (a similar number to the number of processes in Application 1). This permitted us to achieve similar I/O periods for both applications. For example, in the first test, after the period adjustment, the CPU times of the I/O periods were 67.40 and 67.74 s for each application. The effect of this strategy was to reduce the number of phase-shifting reconfigurations, given that I/O conflicts are less frequent. For example, in the three tests it was only necessary to perform one phase shift in the execution in order to avoid any subsequent conflicts.

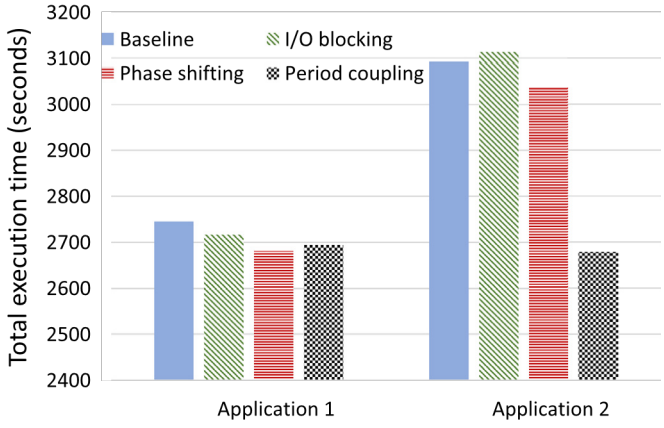


Fig. 16. Total execution time for use case D.

.

Fig. 16 shows the total execution times. Application 2 uses less processes than Application 1, thus the execution times are larger. We can observe that the I/O blocking strategy reduces the execution time of application 1 and slightly increases it for application 2. The Phase shifting strategy combines the reduction of I/O time and CPU time (during the temporary reconfigurations). Both factors contribute to reducing the total execution time. Finally, the Period coupling strategy adjusts the number of processes for Application 2 to approximately the same number as Application 1. Consequently, the execution times of both applications are now similar.

Table 2

Use case configuration with four applications that alternate computation and I/O phases. Np is the configuration with the number of processes that each application initially uses. For each configuration, the computation time corresponds to the CPU and communication times between two I/O phases. The I/O time corresponds to the duration of each I/O phase.

App	Np	Size	CPU time	I/O time	I/O ratio
A1	6	20,0 0 0	77.7	2.9	3.5%
A2	8	17,0 0 0	42.6	5.2	10.8%
A3	8	70 0 0	24.7	1.6	6.0%
A4	6	18,0 0 0	63.3	7.8	11.0%

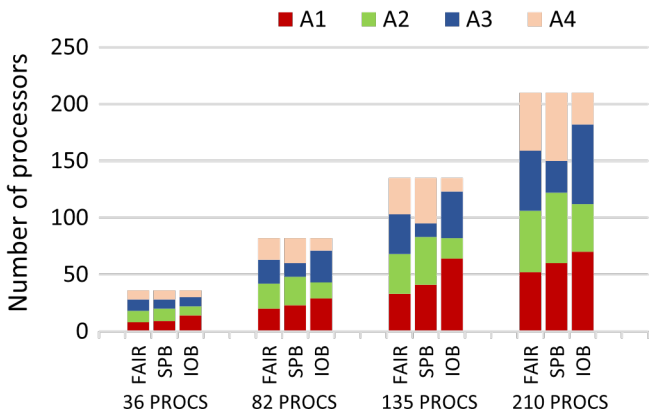


Fig. 17. Number of processes used for different policies of the global performance controller. FR stands for fair scheduler, SP stands for speedup-based scheduler and IO stands for IO-based scheduler.

5.2. *Global performance controller evaluation*

In this section we compare three different execution scenarios. The Fair scheduling performs an agnostic expansion of the applications. The Speedup-based scheduling favors the applications with greatest performance improvement. This scheduling strategy is similar to the one used in Scenario S_3 evaluated by Sudarsan and Ribbens (2016) and represents a more advanced scheduling technique. Finally, the I/O-based scheduling considers both the speedups and I/O intensity for reconfiguring the applications.

Table 2 shows the use case configuration that we use in the evaluation. It consists of four applications. Each one of them executes the Jacobi benchmark using different configurations and input data that produce different performance characteristics: application A1 has a relatively large computation time between the I/O phases and a small I/O time. The CPU time is much larger than communication time. Due to its large dataset size, the application exhibits high speedups when the number of processes is increased. Applications A2 and A4 also use large data sets and good speedups as well, but they have larger I/O times than the previous application. Finally, application A3 dataset is the smallest one, leading to shorter execution and I/O times, as well as smaller speedups.

We have evaluated the previous configuration in four scenarios assuming an increasing number of available resources. In our experiments, we started running the application with the number of processes shown in Table 2 . Then, we increased the number of processes of each application based on the available resources and the criteria related to each global performance controller.

Fig. 17 shows, for each scenario, the number of processes assigned to each application by each global performance controller that we abbreviate as scheduler. In the Figure, FR stands for Fair scheduler, SP stands for Speedup-based scheduler and IO stands for IO-based scheduler. We can observe that the Fair scheduler evenly distributes the existing resources between the existing applications. The speedup-based scheduler assigns more resources to the applications with better speedups (A1, A2 and A4).

Finally, the IO-based scheduler, considers both the application scalability and the I/O ratio, trying to favor the applications with small ratios. When the number of resources is reduced (36 processes in total), it assigns all the resources to application A1 because it is the one that meets both requirements (greatest scalabilities and smallest I/O ratios). However, A1 does not have linear speedups, so, when the number of processes is increased, at a given point, further scalability gains are similar to the achieved ones in A3. This is because for 82 and 135 processes both A1 and A3 benefit from the I/O-based scheduler policy. For larger amounts of available resources (210 processes), the I/O ratios of A1 and A3 become similar to A2 and A4, so all applications benefit (A1 and A3 to a greater extent).

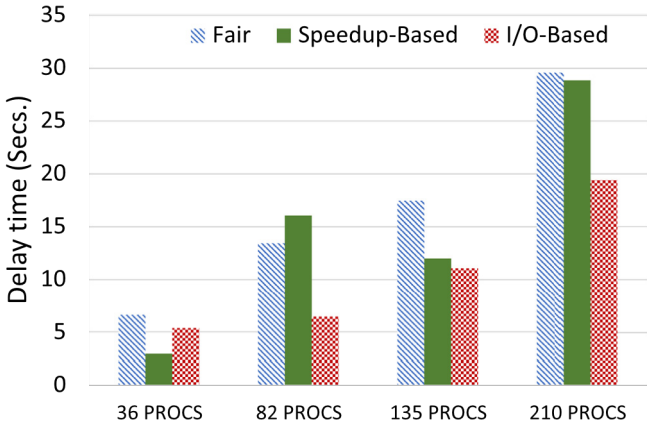


Fig. 18. Aggregated I/O delay times introduced by CLARISSE for different policies of the global performance controller.

Fig. 18 shows the aggregated delay time for each scheduling policy. Each value is the sum of the average delay times of each application after the reconfiguration. Note that the delays are introduced by CLARISSE when two different applications are performing I/O at the same time. The fair scheduler, reduces the I/O period of all applications, increasing the combined pressure on the I/O subsystem mainly by reducing the I/O period of applications A2 and A4. In contrast, the I/O-based scheduling reduces resource provisioning to the I/O intensive applications (A2 and A4), diminishing overall I/O traffic. This produces in general a smaller number of I/O delays and a better overall system utilization. The Speedup-based scheduling obtains intermediate results.

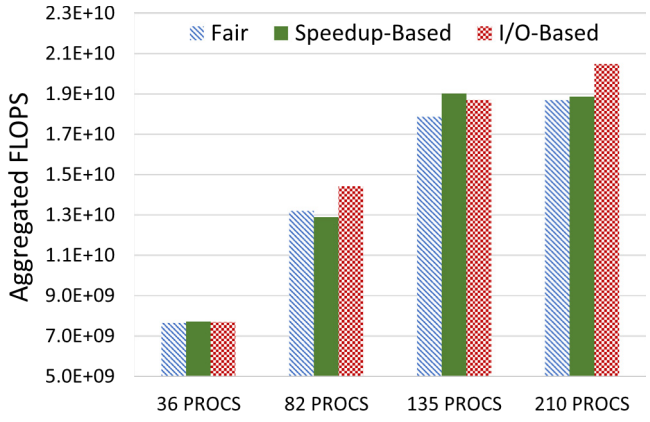


Fig. 19. Aggregated FLOPS for the global performance controller evaluation.

Fig. 19 shows the aggregated FLOPS of all applications after the reconfiguration stage. We can observe that the speedup-based and I/O based scheduling strategies increase overall system performance. The reason is twofold: on one hand, the available resources are assigned to the applications that exhibit good scalability, so more FLOPS are obtained from assigned cores. On the other hand, the I/O phase (that counts both the actual I/O and the I/O delay time) is also reduced using these two strategies, so a greater percentage of the application execution time is now used for CPU time. Note that the I/O based scheduling achieves the best improvements.

6. Related work

6.1. *Providing malleability to applications*

There are two main alternatives for providing malleability for MPI applications: by means of offline reconfiguration or by means of dynamic reconfiguration. Offline reconfiguration, consists of stopping the execution of the application, checkpointing the state

in the persistent memory or disk, and then restarting the application with a different number of processes. We can find several works (Vadhiyar and Dongarra, 2003; Mayes et al., 2005; Cores et al., 2017) that provide malleability to applications using this mechanism. E-HPC (Fox et al., 2017) uses offline reconfiguration in a more general way, implementing a workflow manager that is able to schedule elastic jobs. Process Checkpointing and Migration (PCM) (Maghraoui et al., 2007) is a runtime system built in the context of the Internet Operating System (IOS) (Maghraoui et al., 2006), which uses process checkpointing to provide malleability to MPI applications.

Offline reconfiguration has several important drawbacks, one of the major ones being the overhead introduced by the I/O operations carried out every time the application is reconfigured. Dynamic reconfiguration, on the other hand, provides malleability support during the program execution without the use of checkpointing. This results in lower overheads than its counterpart. We can find several frameworks (Buisson et al., 2005; Kalé et al., 2002) for writing malleable applications based on dynamic reconfiguration. FlexMPI also belongs to this category. Adaptive MPI (Huang et al., 2003; Diener et al., 2017), is an implementation of the MPI standard written on top of Charm++. Adaptive MPI programs receive information about the availability of processors from an adaptive job scheduler. Based on this information, the runtime system uses object migration to adapt the application to a different number of processes.

The work closest to FlexMPI is ReSHAPE (Sudarsan and Ribbens, 2007). ReSHAPE is a framework for malleable MPI applications that performs reconfiguring actions based on application performance. ReSHAPE assumes that all iterations of a parallel application are identical in terms of computation and communication times, and that they have regular communication patterns. FlexMPI's approach targets parallel applications with varying iteration computation times, considering both regular and irregular communication patterns. In this work we extended FlexMPI to consider applications that include parallel I/O operations. In this way, the number of processes involved in the application I/O phases can be dynamically adjusted.

6.2. *I/O-aware coordination and modelling*

On current large-scale HPC platforms one of the main hurdles of performance optimization is that storage I/O performance is difficult to predict because of I/O interferences (Dorier et al., 2014; Yildiz et al., 2016). One of the main roots of performance instability is the fact that the I/O accesses to the storage hierarchy are typically not scheduled. For instance, two applications writing at the same time can interfere and have both their performance and global throughput greatly affected.

There are several solutions that address this problem on different levels of the HPC platform architecture. Shared-burst buffers represent an intermediate layer between the executing applications and the file system. Although designed to efficiently absorb bursty I/O traffic, shared-burst buffers are also susceptible to suffering performance degradation under I/O interference. Several solutions (Thapaliya et al., 2016; Kougkas et al., 2016; Tang et al., 2017) have been introduced to mitigate the effect of I/O interference based on coordinating the access to shared-burst buffers and back-end parallel file systems. Aequilibro (Neuwirth et al., 2016) mitigates the I/O interferences by providing I/O load balancing at file-system level. Son et al. (2017) present an approach in which the I/O requests are analyzed at runtime in order to identify and exclude the I/O nodes that have low performance. These solutions are complementary to our work.

A different solution consists of predicting the application interferences (Alves and de Assumpo Drummond, 2017) or predicting the application I/O patterns (Snyder et al., 2015; Dorier et al., 2016b) for resource provisioning. These solutions could be used to improve the accuracy of the I/O interference predictor used in our work.

Elasticity has been applied at file-system level. One solution is provided in AHPIOS (Isaila et al., 2010), where a light-weight adhoc parallel I/O system is presented. This system includes elastic partitions that can scale up and down with the number of storage resources. Another solution is IKAROS (Filippidis et al., 2016), that permits the dynamic creation of clusters of storage nodes per job including both local and remote storage resources, based on application characteristics. In the context of cloud computing, some solutions for elasticity in I/O have been proposed, such as SpringFS (Xu et al., 2014) as well as solutions based on the Hadoop Distributed File System (HDFS) (Lim et al., 2010; Cheng et al., 2012). Nonetheless, by and large, existing malleability solutions (Casanova et al., 2014; Kalé et al., 2002; Klein and Pérez, 2011; Hungershofer, 2004; Cirne and Berman, 2002; Prabhakaran et al., 2015) are mostly confined to the elasticity of compute and memory allocations.

Existing solutions based on the I/O traffic coordination include TWINS (Bez et al., 2017). TWINS is a new I/O scheduling technique that represents an alternative to collective I/O. By means of TWINS, the access to the I/O nodes is coordinated at the I/O forwarding layer, reducing the contention on the I/O nodes. Damaris (Dorier et al., 2016a) uses dedicated cores (or nodes) to perform the I/O in an asynchronous manner, increasing the overlap of computation and I/O, and reducing the I/O variability. ASCAR (Li et al., 2015)

is a storage traffic management system that coordinates the I/O traffic of the running application. The system uses traffic controllers on storage clients to detect I/O congestion and introduces traffic rules to reduce the I/O congestion. These solutions are also complementary to our proposal.

Several studies (Isaila et al., 2016; Dorier et al., 2014; Gainaru et al., 2015) have investigated parallel I/O scheduling solutions based on centralized control and coordination of I/O access to the parallel file system. By means of these approaches, I/O accesses are dynamically scheduled to reduce the effects of I/O interference on applications. These solutions either provide applications with exclusive access to a shared resource such as a parallel file system or allow the partial or total overlap of operations, depending on the expected impact of the interference. Although these approaches have been shown to be effective, they introduce delays in I/O operations that are blocked. Note that when an I/O access is blocked, the associated application may also be blocked, misusing the application's computational resources. In this work we evaluate and compare the performance of our proposal against the I/O blocking technique presented by Isaila et al. (2016) , proving that the proposed framework reduces the I/O interference without blocking running applications.

6.3. *Resource-aware job scheduling*

Several works address the problem of scheduling jobs considering the availability of multiple types of resources that could limit the job performance. However, this integration is mostly focused on the use of CPU resources, not considering the I/O. Carroll and Grosu (2010) propose a method for the online scheduling of malleable jobs where the main goal was to assign compute resources to jobs so that the total running time is reduced. Sun et al. (2011) proposes a scheduling strategy in which the compute resources are equipartitioned among malleable jobs. Following this, the job-resource assignment is periodically adjusted based on the application scalabilities. Similar approaches were taken in (Mounie et al., 1999; Blazewicz et al., 2001). Iserte et al. (2017) integrate the Nanos++ runtime with Slurm RMS for increasing the system throughput via a better resource utilization. Benoit et al. (2018) introduce a model for scheduling malleable applications on a failureprone platform. The novelty of this work is to consider the effect of failures in the applications on the scheduling policy. In (Gupta et al., 2014) a technique implemented on top of CHARM++ is introduced for providing malleability to parallel applications. This technique includes the integration of the job scheduler, the resource manager and the malleable runtime system to enhance the system utilization.

In the same context, Sudarsan and Ribbens (2016) introduce an application scheduler that supports dynamic resizing of parallel applications. This scheduler does not consider the application I/O. One of the scheduling policies included in it (Scenario S_3 , based on the Max-Benefit expansion policy) is similar to the Speedup-based scheduling introduced and evaluated in this work. In our experiments we show that an I/O-aware scheduling is more efficient than the Speedup-based counterpart.

In the context of scheduling jobs considering I/O constraints, Sun et al. (2018) introduce new scheduling algorithms that consider constraints in both list and pack scheduling paradigms. Tetris (Grandl et al., 2014) and LoTES (Tran et al., 2018) consider constraints in CPU, memory, disk, and network bandwidth for packing tasks and improving the cluster efficiency. In (Zhou et al., 2015) an I/O-aware batch scheduling system is presented that coordinates the I/O requests in order to avoid I/O congestion. This coordination is performed by selecting the jobs that are being executed. In a similar way, AIS (Liu et al., 2016) is a I/O-aware scheduler that identifies the I/O characteristics of the applications by means of an offline analysis of the I/O traffic logs. Using this information, the applications are scheduled, reducing the I/O contention. Note that these works are complementary to our proposed framework given that they tackle the problem using a different perspective to our work (we study how to efficiently provision new resources to running applications, not waiting ones).

7. Conclusions and future work

This work presents the integration of CLARISSE and FlexMPI runtimes into a single framework. This integration is performed both at application level as well as central-controller level. At application level, the proposed framework allows the programs to be monitored, in order to determine the weight and performance of the CPU, communication and I/O phases. In addition, these programs acquire new capabilities for dynamically changing the number of processes by means of malleability and scheduling the I/O operations in a coordinated manner. All these features are transparently applied, without user intervention and with minor code modifications. At central-controller level, we introduce different scheduling techniques that, instead of considering the execution of new incoming applications, focus on selectively increasing the number of processes of the already executing jobs using the available computational resources in a temporary or permanent way. Two different (and complementary) strategies are

presented in this work to leverage both application malleability and cross-application coordination for mitigating the I/O interference and improving the system throughput. To the best of our knowledge this is the first work that leverages malleability for improving the I/O and overall system performance.

By avoiding I/O conflicts, the overall I/O time can be reduced by up to 49% with respect to the default execution scheme used by HPC platforms. A reduction of up to a 10% of I/O time and 56% of delay time are achieved compared to the more advanced I/O blocking strategy of Isaila et al. (2016) . By means of a selective distribution of the available computational resources, the aggregated FLOPS can be increased by 10% with respect to the baseline case and 8% compared to a more advanced technique that considers the CPU performance.

As a future work, we first plan to integrate our prototype into a real workload manager like Slurm considering a more complex execution environment with diverse classes of applications. Additionally, further optimization techniques could be developed based on the holistic characterization of the execution applications. Examples of these are elastic I/O resource provisioning and locality-aware application placement. Finally, applications with non-periodic I/O access and additional performance metrics, like energy consumption, could be considered and included in our analysis.

Acknowledgment

We would like to thank to Florin Isaila for providing us CLARISSE implementation as well as his invaluable comments for enhancing the quality of this work. Forever in our thoughts. The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work has been partially supported by the Spanish “Ministerio de Economia y Competitividad” under the project grant TIN201679637-P “Towards Unification of HPC and Big Data paradigms” and EU under the COST Program Action IC1305, Network for Sustainable Ultrascale Computing (NESUS).

References

Alves, M.M., de Assumpo Drummond, L.M., 2017. A multivariate and quantitative model for predicting cross-application interference in virtual environments. J. Syst. Softw. 128, 150–163. doi: 10.1016/j.jss.2017.04.001 .

Amer, A., Balaji, P., Bland, W., Gropp, W., Guo, Y., Latham, R., Lu, H., Oden, L., Pena, A. J., Raffenetti, K., Seo, S., Si, M., Thakur, R., Zhang, J., Zhao, X., November 10, 2017. MPICH Users Guide. Mathematics and Computer Science Division, Argonne National Laboratory.

Benoit, A., Pottier, L., Robert, Y., 2018. Resilient co-scheduling of malleable applications. Int. J. High Perform. Comput. Appl. 32 (1), 89–103. doi: 10.1177/1094342017704979 .

Bez, J. L., Boito, F. Z., Schnorr, L. M., Navaux, P. O. A., Mhaut, J., 2017. Twins: Server access coordination in the I/O forwarding layer. In: Proceedings of the 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), pp. 116–123. doi: 10.1109/PDP.2017.61 .

Blazewicz, J., Machowiak, M., Mounie, G., Trystram, D., 2001. Approximation algorithms for scheduling independent malleable tasks. Proceedings of the Euro-Par 2001 Parallel Processing, 191–197.

Buisson, J., André, F., Pazat, J.-L., 2005. A framework for dynamic adaptation of parallel components. In: Proceedings of the International Conference ParCo. Vol. 33, pp. 65–72.

Carroll, T.E. , Grosu, D. , 2010. Incentive compatible online scheduling of malleable parallel jobs with individual deadlines. In: Proceedings of the 39th International Conference on Parallel Processing (ICPP). IEEE, pp. 516–524 .

Casanova, H. , Giersch, A. , Legrand, A. , Quinson, M. , Suter, F. , 2014. Versatile, able, and accurate simulation of distributed applications and platforms. J. lel Distrib. Comput. 74 (10), 2899–2917 .

Cheng, Z., Luan, Z., Meng, Y., Xu, Y., Qian, D., Roy, A., Zhang, N., Guan, G., 2012. ERMS: an elastic replication management system for HDFS. In: Proceedings of the IEEE International Conference on Cluster Computing Workshops. IEEE, pp. 32–40.

Cirne, W. , Berman, F. , 2002. Using moldability to improve the performance of percomputer jobs. J. Parallel Distrib. Comput. 62 (10), 1571–1601 .

Cores, I. , González, P. , Jeannot, E. , Martín, M.J. , Rodríguez, G. , 2017. An tion-level solution for the dynamic reconfiguration of MPI applications. In: ceedings of the High Performance Computing for Computational PAR 2016. Springer International Publishing, pp. 191–205 .

Diener, M., White, S., Kale, L.V., Campbell, M., Bodony, D.J., Freund, J.B., 2017. Improving the memory access locality of hybrid MPI applications. In: Proceedings of the 24th European MPI Users’ Group Meeting. ACM, New York, NY, USA, pp. 11:1–11:10. doi: 10.1145/3127024.3127038 .

Dorier, M. , Antoniu, G. , Cappello, F. , Snir, M. , Sisneros, R. , Yildiz, O. , Ibrahim, S. , Peterka, T. , Orf, L. , 2016. Damaris: addressing performance variability in data management for post-petascale simulations. ACM Trans. Parallel Comput. 3, 15:1–15:43 .

Dorier, M. , Antoniu, G. , Ross, R. , Kimpe, D. , Ibrahim, S. , 2014. CALCioM: mitigating I/O interference in HPC systems through cross-application coordination. In: ceedings of the IPDPS International Parallel and Distributed Processing posium, Phoenix, United States, pp. 155–164 .

Dorier, M., Ibrahim, S., Antoniu, G., Ross, R., 2016. Using formal grammars to predict i/o behaviors in HPC: the omnisc’io approach. IEEE Trans. Parallel Distrib. Syst. 27 (8), 2435–2449. doi: 10.1109/TPDS.2015.2485980 .

Filippidis, C., Tsanakas, P., Cotronis, Y., 2016. IKAROS: A scalable I/O framework for high-performance computing systems. J. Syst. Softw. 118, 277–287. doi: 10.1016/ j.jss.2016.05.027 .

Fox, W., Ghoshal, D., Souza, A., Rodrigo, G.P., Ramakrishnan, L., 2017. E-HPC: a library for elastic resource management in HPC environments. In: Proceedings of the 12th Workshop on Workflows in Support of Large-Scale Science. ACM, New York, NY, USA, pp. 1:1–1:11. doi: 10.1145/3150994.3150996 .

Gainaru, A., Aupy, G., Benoit, A., Cappello, F., Robert, Y., Snir, M., 2015. Scheduling the I/O of HPC applications under congestion. In: Proceeduings of the IEEE International Parallel and Distributed Processing Symposium, IPDPSHyderabad, India, May 25–29, 2015, pp. 1013–1022. doi: 10.1109/IPDPS.2015.116 .

Grandl, R., Ananthanarayanan, G., Kandula, S., Rao, S., Akella, A., 2014. Multiresource packing for cluster schedulers. SIGCOMM Comput. Commun. Rev. 44 (4), 455–466. doi: 10.1145/2740070.2626334 .

Gupta, A., Acun, B., Sarood, O., Kal, L.V., 2014. Towards realizing the potential of malleable jobs. In: Proceedings of the 21st International Conference on High Performance Computing (HiPC), pp. 1–10. doi: 10.1109/HiPC.2014.7116905 .

Habib, S. , Roser, R. , Gerber, R. , Antypas, K. , et al. , 2016. ASCR/HEP Exascale quirements Review Report. Technical Report. National Energy Research Scientific Computing Center and the Office of Advanced Computational Research .

Huang, C., Lawlor, O., Kalé, L. V., 2003. Adaptive MPI. In: Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958, College Station, Texas. pp. 306–322.

Hungershofer, J., 2004. On the combined scheduling of malleable and rigid jobs. In: Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing, SBAC-PAD. IEEE, pp. 206–213.

Isaila, F., Blas, F.J.G., Carretero, J., keng Liao, W., Choudhary, A., 2010. A scalable message passing interface implementation of an ad-hoc parallel I/O system. Int. J. High Perform. Comput. Appl. 24 (2), 164–184. doi: 10.1177/ 1094342009347890 .

Isaila, F., Carretero, J., Ross, R., 2016. CLARISSE: a middleware for data-staging coordination and control on large-scale HPC platforms. In: Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 346–355. doi: 10.1109/CCGrid.2016.24 .

Iserte, S., Mayo, R., Quintana-Ort, E. S., Beltran, V., Pea, A. J., 2017. Efficient scalable computing through flexible applications and adaptive workloads. In: Proceedings of the 46th International Conference on Parallel Processing Workshops (ICPPW), pp. 180–189. doi: 10.1109/ICPPW.2017.36 .

Kalé, L. V., Kumar, S., DeSouza, J., 2002. A malleable-job system for timeshared parallel machines. In: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid. IEEE, pp. 230.

Klein, C., Pérez, C., 2011. An RMS for non-predictably evolving applications. In: Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER). IEEE, pp. 326–334.

Kougkas, A., Dorier, M., Latham, R., Ross, R., Sun, X.-H., 2016. Leveraging burst buffer coordination to prevent I/O interference. In: Proceedings of the IEEE International Conference on Science. IEEE, pp. 371–380.

Li, Y. , Lu, X. , Miller, E.L. , Long, D.D.E. , 2015. ASCAR: automating contention ment for high-performance storage systems. In: Proceedings of the 31st sium on Mass Storage Systems and Technologies (MSST), pp. 1–16 .

Lim, H. C., Babu, S., Chase, J. S., 2010. Automated control for elastic storage. In: Proceedings of the 7th International Conference on Autonomic Computing. ACM, pp. 1–10.

Liu, Y., Gunasekaran, R., Ma, X., Vazhkudai, S.S., 2016. Server-side log data analytics for I/O workload characterization and coordination on large shared storage systems. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 819–829. doi: 10.1109/SC.2016. 69 .

Maghraoui, K.E., Desell, T.J., Szymanski, B.K., Varela, C.A., 2006. The internet operating system: middleware for adaptive distributed computing. Int. J. High Perform. Comput. Appl. 20 (4), 467–480. doi: 10.1177/1094342006068411 .

Maghraoui, K.E., Desell, T.J., Szymanski, B.K., Varela, C.A., 2007. Dynamic malleability in iterative MPI applications. In: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07), pp. 591– 598. doi: 10.1109/CCGRID.2007.45 .

Martín, G., Singh, D.E., Marinescu, M.-C., Carretero, J., 2015. Enhancing the performance of malleable MPI applications by using performance-aware dynamic reconfiguration. Parallel Comput. 46 (0), 60–77. https://doi.org/10.1016/j.parco. 2015.04.003 .

Mayes, K., Luján, M., Riley, G., Chin, J., Coveney, P., Gurd, J., 2005. Towards performance control on the grid. Philosoph. Trans. R. Soc Lond. A Math. Phys. Eng. Sci. 363 (1833), 1793–1805. doi: 10.1098/rsta.2005.1607 .

Mounie, G. , Rapine, C. , Trystram, D. , 1999. Efficient approximation algorithms for scheduling malleable tasks. In: Proceedings of the Eleventh Annual ACM posium on Parallel Algorithms and Architectures. ACM, pp. 23–32 .

Neuwirth, S., Wang, F., Oral, S., Vazhkudai, S., Rogers, J., Bruening, U., 2016. Using balanced data placement to address I/O contention in production environments. In: Proceedings of the 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp. 9–17. doi: 10.1109/SBAC-PAD. 2016.10 .

Panziera, J.-P. , Nomin, J.-P. , Gilliot, M. , et al. , 2017. ETP4HPC Strategic Research Agenda. Technical Report. European Technology Platform for High Performance Computing .

Prabhakaran, S., Neumann, M., Rinke, S., Wolf, F., Gupta, A., Kalé, L. V., 2015. A batch system with efficient scheduling for malleable and evolving applications. In: Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS), Hyderabad, India. IEEE Computer Society, pp. 429–438. doi: 10.1109/IPDPS.2015.34 .

Snyder, S., Carns, P., Latham, R., Mubarak, M., Ross, R., Carothers, C., Behzad, B., Luu, H. V. T., Byna, S., Prabhat, 2015. Techniques for modeling large-scale HPC I/O workloads. In: Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems. ACM, New York, NY, USA, pp. 5:1–5:11. doi: 10.1145/2832087.2832091 .

Son, S.W., Sehrish, S., Liao, W.-k., Oldfield, R., Choudhary, A., 2017. Reducing i/o variability using dynamic i/o path characterization in petascale storage systems. J. Supercomput. 73 (5), 2069–2097. doi: 7 .

Sudarsan, R., Ribbens, C. J., 2007. Reshape: a framework for dynamic resizing and scheduling of homogeneous applications in a parallel environment. In: Proceedings of the International Conference on Parallel Processing (ICPP 2007), pp. 44. doi: 10.1109/ICPP.2007.73 .

Sudarsan, R., Ribbens, C.J., 2016. Combining performance and priority for scheduling resizable parallel applications. J. Parallel Distrib. Comput. 87, 55–66. doi: 10.1016/ j.jpdc.2015.09.007 .

Sun, H. , Cao, Y. , Hsu, W.-J. , 2011. Fair and efficient online adaptive scheduling for multiple sets of parallel applications. In: Proceedings of the IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS). IEEE, pp. 64–71 .

Sun, H., Elghazi, R., Gainaru, A., Aupy, G., Raghavan, P., 2018. Scheduling parallel tasks under multiple resources: list scheduling vs. pack scheduling. In: Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 194–203. doi: 10.1109/IPDPS.2018.0 0 029 .

Tang, K., Huang, P., He, X., Lu, T., Vazhkudai, S.S., Tiwari, D., 2017. Toward managing HPC burst buffers effectively: draining strategy to regulate bursty I/O behavior. In: Proceedings of the IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), pp. 87–98. doi: 10.1109/MASCOTS.2017.35 .

Thapaliya, S., Bangalore, P., Lofstead, J., Mohror, K., Moody, A., 2016. Managing I/O interference in a shared burst buffer system. In: Proceedings of the 45th International Conference on Parallel Processing (ICPP), pp. 416–425. doi: 10.1109/ICPP. 2016.54 .

Tran, T.T., Padmanabhan, M., Zhang, P.Y., Li, H., Down, D.G., Beck, J.C., 2018. Multistage resource-aware scheduling for data centers with heterogeneous servers. J. Schedul. 21 (2), 251–267. doi: x .

Vadhiyar, S.S. , Dongarra, J.J. , 2003. SRS: a framework for developing malleable and migratable parallel applications for distributed systems. Parallel Process. Lett. 13 (02), 291–312 .

Xu, L., Cipar, J., Krevat, E., Tumanov, A., Gupta, N., Kozuch, M. A., Ganger, G. R., 2014. SpringFS: bridging agility and performance in elastic distributed storage. In: Proceedings of the FAST, pp. 243–255.

Yildiz, O., Dorier, M., Ibrahim, S., Ross, R., Antoniu, G., 2016. On the Root Causes of Cross-Application I/O Interference in HPC Storage Systems. In: Proceedings of the 30th IEEE International Parallel and Distributed Processing Symposium (IPDPS), Chicago, United States. pp. 750–759.

Zhou, Z., Yang, X., Zhao, D., Rich, P., Tang, W., Wang, J., Lan, Z., 2015. I/O-aware batch scheduling for petascale computing systems. In: Proceedings of the IEEE International Conference on Cluster Computing, pp. 254–263. doi: 10.1109/CLUSTER. 2015.45 .