electronics

MDPI

Article

Performance-Aware Scheduling of Parallel Applications on Non-Dedicated Clusters

Alberto Cascajo *, David E. Singh and Jesus Carretero

Computer Science and Engineering Department, Universidad Carlos III de Madrid, Avenida Universidad 30, Leganés, 28911 Madrid, Spain

* Correspondence: acascajo@inf.uc3m.es

Received: 1 August 2019; Accepted: 29 August 2019; Published: 2 September 2019

Abstract: This work presents a HPC framework that provides new strategies for resource management and job scheduling, based on executing different applications in shared compute nodes, maximizing platform utilization. The framework includes a scalable monitoring tool that is able to analyze the platform's compute node utilization. We also introduce an extension of CLARISSE, a middleware for data-staging coordination and control on large-scale HPC platforms that uses the information provided by the monitor in combination with application-level analysis to detect performance degradation in the running applications. This degradation, caused by the fact that the applications share the compute nodes and may compete for their resources, is avoided by means of dynamic application migration. A description of the architecture, as well as a practical evaluation of the proposal, shows significant performance improvements up to 20% in the makespan and 10% in energy consumption compared to a non-optimized execution.

Keywords: scalable tools; monitoring tools; scheduling; malleability

1. Introduction

Currently, one of the key challenges in cluster computing is the development of scalable components that operate in a coordinated manner for a more efficient use of hardware resources. Over the past few years, researchers have studied ways of improving the scheduling model taking into account the application characteristics [1]. Traditionally, in HPC, schedulers assign compute nodes in a static way following two different allocation policies. The first one consists of running different applications on different nodes in order to provide exclusive access to the hardware resources [2]. This provides application isolation at the expense of a degree of under-usage of computational resources when the number of the application processes is smaller than the number of existing cores. The second allocation policy involves sharing the compute nodes between different applications. This allows for more efficient use of the resources given that now it is possible to pack several small-sized applications on the same compute node, maximizing platform utilization [3]. However, with this latter option, the running applications may now interfere competing for the compute node resources (like the last-level cache memory or the network controller) potentially producing a degradation in the application performance.

This work addresses these challenges by providing a novel interference-aware technique that is able to detect when two applications experience interference (referred to as hot-spots in this work) called *hot-spot*) that produces a performance degradation. Additionally, we present a methodology to overcome this degradation by migrating one of the applications to a different compute node. The solution presented in this paper includes a prototype of a malleable scheduler and an extension of CLARISSE [4], a middleware for data-staging coordination and control on large-scale HPC platforms.

This extension includes new functionalities in CLARISSE, and provides its integration with the application scheduler, the LIMITLESS monitor [5] and the FlexMPI runtime [6].

LIMITLESS is a scalable light-weight monitor that analyzes the compute nodes and detects interference-related hazards on the executing applications. FlexMPI provides MPI application with monitoring and malleable capabilities. The hot-spot detection is carried out by a combination of compute node and application monitoring, performed by LIMITLESS and FlexMPI, respectively. The conflicting applications are migrated by means of the malleable capabilities provided by FlexMPI. This paper provides a detailed description of the architecture as well as a practical evaluation on a real platform. As far as we know, this is the first work that coordinates these three features (system/application-level monitoring, scheduling and malleability) in a unified framework to enhance the system utilization.

To summarize, the main contributions of this work are:

- *LIMITLESS*, a light-weight scalable monitor that evaluates the compute-node status and provides resource allocation based on different policies.
- An extension of *CLARISSE* that includes control components integrated with LIMITLESS and FlexMPI, as well as a performance-aware policy for avoiding conflicts between applications.
- An *execution framework* that includes a scheduler that works in cooperation with the previous components. This framework permits execution of workflows and dynamically migrates the running applications when CLARISSE detects interference-related hot-spots.
- An *extensive performance evaluation* that demonstrates the capabilities of CLARISSE to enhance application performance and the cluster throughput.

The structure of the paper is as follows: Section 2 describes the architecture organization; Section 3 provides a detailed description of the framework components; Section 4 provides a practical evaluation of the CLARISSE middleware and LIMITLESS monitor; Section 5 shows relevant works related to our proposal; Finally, Section 6 summarizes the main conclusions.

2. Architecture Organization

2.1. Software and Management Tools

The previous implementation of CLARISSE provided an elastic load-aware collective I/O technique and cross-application parallel I/O scheduling policies. This work extends CLARISSE's functionalities integrating system and application monitoring as well as an interference-aware policy. Figure 1 shows CLARISSE's structure that consists of three components corresponding to data, control, and policy layers. The data plane is responsible for staging data between applications and storage or between different applications. This component is related to I/O management, a topic that is not considered in this paper.

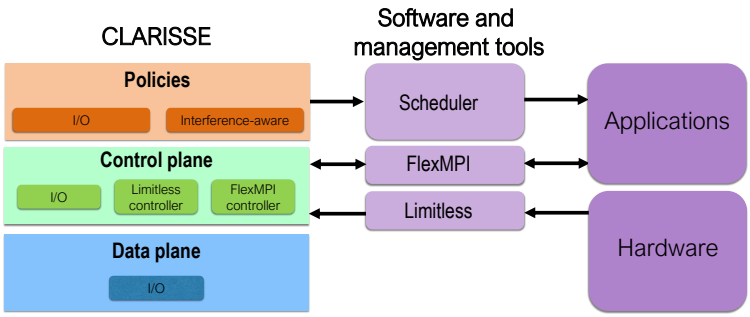

Figure 1. Diagram that illustrates the separation of data plane, control, and policy in CLARISSE and the interaction with the different software and management tools of the platform.

The control plane consists of three controllers: I/O, LIMITLESS and FlexMPI. Originally, the I/O controller acted as a coordination framework of data staging, including the functionalities of I/O

load balancing, I/O scheduling, and resilience. The new functionalities incorporate the LIMITLESS controller that gathers system-wide monitoring alerts and provides a global view of the platform status.

FlexMPI controller is related to application-oriented control actions that include the use of monitoring, dynamic load balancing and malleable reconfiguration. The monitoring can be selectively activated in order to obtain application-related performance metrics like execution time as well as hardware-counter information such as FLOPS and cache misses. Dynamic load balance permits the distribution of the application workload based on the performance of each compute-node and takes into account whether the CPU resources are exclusive or shared between different applications [7]. By means of malleability, FlexMPI controller can dynamically send commands to create or destroy the applications processes and, in the case of creation, allocate them to certain compute nodes.

Finally, the policy layer includes different policies that leverage the information gathered by the previous layers. The I/O-related policies include an elastic collective I/O and a parallel I/O scheduling policy [4]. In this work, we extend this layer with a novel interference-aware policy that is able to detect hot-spots and prevent performance degradation of the running applications.

2.2. Overview

Figure 2 shows a general overview of our scalable framework. The central part corresponds to the cluster's compute nodes organized into two racks. The upper half of the figure corresponds to LIMITLESS middleware, responsible for the monitoring of the platform resources. It consists of one *LIMITLESS Daemon Monitor* (LDM) per compute node that periodically gathers different metrics related to the compute node status, another *LIMITLESS DaeMon Aggregators* (LDA) that gathers information on the node's LDMs (arrows 1) and sends them to the *LIMITLESS DaeMon Server* (LDS) (arrow 2), which processes and stores the information in a database and includes a GUI. Moreover, the LDS locally analyzes the monitor metrics and notifies the *LIMITLESS controller* (arrow 3) when a compute node has a hot spot. In this work, we consider three different classes of interference: use of nearly all the existing node memory (RAM hot spot), conflicting accesses to the node network interface (NET hot spot), and cache-related conflicts (CACHE hot spot).

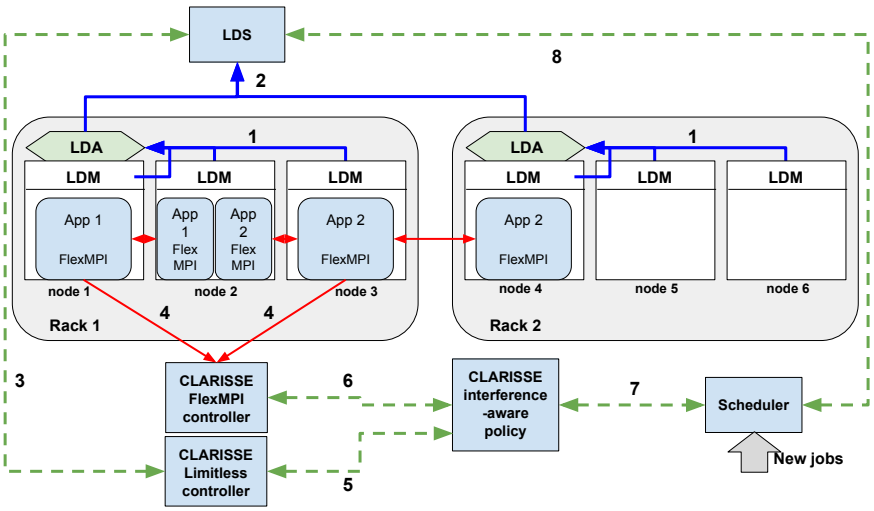

Figure 2. System architecture.

The lower half of Figure 2 shows the scheduler that is responsible for executing the new applications according to the availability of resources. FlexMPI consists of a library that is executed with the applications and is connected to CLARISSE's FlexMPI controller (arrow 4) that collects the application-related monitoring metrics. Finally, CLARISSE's policy manager performs the analysis and coordination of the running applications. It leverages the information provided by both the LIMITLESS monitor and FlexMPI (arrows 5 and 6) in order to verify whether a hot-spot in a compute

node produces a real performance degradation of the applications running on it. If the degradation exists, then the policy manager is responsible for sending the control command to the scheduler (arrow 7) and FlexMPI (arrow 6) for migrating one of the applications to a different compute node. In this work, LIMITLESS also acts as a resource manager that provides (arrow 8) resource allocation for each application that is executed or migrated.

We now introduce how the framework works with a practical example. Let's assume that each compute node in Figure 2 consists of 12 cores and that there are two applications with 18 and 30 processes ready to be executed. Application 1 is firstly executed in nodes 1 and 2 with 12 and 6 processes, respectively. The scheduler monitors the application (using FlexMPI) and extracts different performance metrics. Then, application 2 is executed in a similar way, being allocated in compute nodes 3, 4 and 5, with 12, 12, and 6 processes, respectively. Note that all the nodes are exclusively used by each application, thus, when the performance metrics are collected by FlexMPI, there is no risk of interference with other applications. After that, the six processes in compute node 5 are dynamically migrated (using FlexMPI) to compute node 2, which becomes a shared node. Note that, with this strategy, only four compute nodes are used (instead of five which would be the case if the application was be exclusively allocated), saving resources and energy. Note also that this stage can be avoided when the runtime has previous information about the application performance, for instance, by recording performance from previous executions. In this case, the applications could be directly allocated in shared nodes.

Let's now assume that applications 1 and 2 create a cache-related interference in node 2. This interference is detected by the LDM and is notified to the LDS that subsequently alerts the LIMITLESS controller to the risk. Following this, CLARISSE selectively activates the application-level monitoring for both applications and compares the metrics with the original ones. After detecting a performance degradation, the six processes of application 2 are moved from node 2 to another free node (for instance, node 6) that is allocated exclusively for this application. Now, node 2 has six cores available for other new applications.

Note that the framework is scalable because the LDAs receive as many connections as nodes per rack, and they are connected with the LDS in a hierarchical way. On the application side, the FlexMPI controller has only one connection per application, and the rest of the components are interconnected according to a peer-to-peer basis. The application monitoring is performed on demand, for just a short duration if an interference risk exists.

3. Component Description

This section provides a detailed description of the different components of the execution framework.

3.1. Limitless Monitor

LIMITLESS is designed to provide scalability in two different ways. First, the monitor logic is distributed in a topological-aware configuration. The different monitor components are interconnected using a graph-based scheme that can be mapped to the cluster's network topology. The second scalability mechanism is aimed to reduce the memory and network monitoring overhead for large-scale platforms. The LDMs process the monitored data applying in-node filtering that reduces the network traffic based on the predefined tolerance. Additionally, the monitor includes a heartbeat protocol to detect faulting nodes.

Algorithm 1 shows the LIMITLESS Daemon Monitor distributed logic that collects the different node-related performance metrics. Variable $sample_i$ is a vector of metrics collected at the i-th sample interval that includes, among others, CPU, memory and network utilization as well as last-level cache misses and energy consumption. Once the LDM collects a new sample, it evaluates whether any of the metrics are not within the given tolerance (given by tol_j where j is the metric index) of a given reference metric sample *ref_metric*. In this case, the complete sample is sent to the LDA. Then, the daemon sleeps until new collection time. The reference samples are created when the monitor is deployed

and updated periodically based on the average value of the previous samples. The user defines the frequency that the metrics are collected, it is an argument of the program. In our experiments, a new sample is collected every second.

Figure 3 shows the LDS architecture which is a multi-thread application with four types of components: *receiver*, *scheduler communicator*, *client communicator* and *processors*. The receiver thread is in charge of receiving all the monitoring packets and storing them in a buffer. The processor threads are executed using a pool of threads in order to leverage the machine concurrency. They are responsible for extracting the packets from the buffer and processing them, evaluating the existence of hot-spots, and saving the monitoring data into a database. The Client communicator is in charge of receiving the petitions from the clients (GUI, CLARISSE, etc.) and retrieving the requested information from the system.

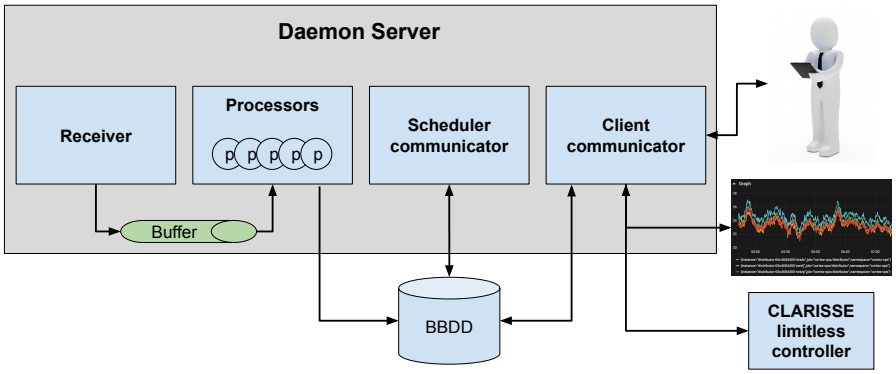

Figure 3. LIMITLESS server architecture.

The scheduler communicator acts as a resource manager, allocating the resources required by the scheduler. Algorithm 2 describes the Scheduler communicator logic, which allows allocation or reallocation of compute nodes for applications. Function $allocate()$ receives requests from the scheduler to allocate new resources. The input arguments are the application id (app_i), the number of processes to be allocated (Δp) and a flag that indicates whether the compute nodes must be non-exclusive ($excl = 0$) or exclusive ($excl = 1$). Using the app_i value, it is possible to distinguish whether the request is related to a new application that should be executed, or an existing running application that should be migrated by means of malleability. Note that the allocation policy depends on this. On one hand, for new applications, $allocate\_new()$ returns the related compute nodes. These nodes may not be exclusively for the application. On the other hand, for existing applications, $reallocate()$ returns the related compute nodes that are exclusive based on the value of $excl$ argument. Function $return\_scheduler()$ sends the list of selected nodes to the system scheduler. Note that Δp may be negative in the case of reducing the application size or may be positive. For positive Δp, this function takes into account the existing application layout and tries to allocate the new processes in the same compute nodes as those currently used by the application. If it is not possible, it allocates the nearest ones using the topological information of the system.

Algorithm 1 LIMITLESS DaeMon Monitor algorithm.

while *running* **do**
  // Data collection
  $\mathbf{sample}_i = collect\_node\_metrics()$
  $i++$
  // In-node analysis
  for each $metric_j \in \mathbf{sample}_i$ **do**
    if $metric_j \notin [ref\_metric_j - tol_j, ref\_metric_j + tol_j]$ **then**
      $send(\mathbf{sample}_i)$
      break
    end if
  end for
  $update(ref\_metric)$
  $waitTillNewCollectionTime(frequency)$
end while

Algorithm 2 Scheduler communicator algorithm in a LIMITLESS DaeMon Server.

Scheduler communicator
while *running* **do**
  $rcvd\_query = socket\_scheduler\_listener()$
  $[app_i, \Delta p, excl] = exec\_multicriteria\_query(rcvd\_query)$
  if $allocate(app_i, \Delta p, excl)$ **then**
    if $is\_new\_application(app_i)$ **then**
      $nodes = allocate\_new(\Delta p)$
    else
      $nodes = reallocate(app_i, \Delta p, excl)$
    end if
    $return\_scheduler(nodes)$
  end if
end while

3.2. Flexmpi

FlexMPI is a runtime which implements performance-aware dynamic malleability for iterative MPI applications. This runtime was previously developed in [6] and is implemented as a library on top of the MPICH implementation. FlexMPI includes an Application Programming Interface (API) to access its functionalities from external components. The source code of FlexMPI as well as several use cases can be found in [8].

Figure 4 shows an example of the integration of FlexMPI with an MPI applications. This is achieved by intercepting selected MPI calls through the profiling mechanism of MPI (PMPI) and inserting control points that implement the logic of distributed control algorithms. When an MPI routine is wrapped by FlexMPI (arrows 1 in Figure 4), some actions are performed by the library [6] 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 Figure 4) are not intercepted by the libraries and are directly executed by MPI. Examples of these are synchronization and datatype-management routines.

The control point logic of each library is responsible for coordinating all the processes of the application (arrow 3 in Figure 4). This coordination includes collecting monitoring data from the all

the processes into the application controller, and broadcasting control information received by the application controller. For each parallel application, FlexMPI employs an application controller (one per library) that is executed associated with the rank-0 process. These controllers include the communication API that is responsible (arrow 4 in Figure 4) for sending the monitoring data to the performance modeler, and receiving different commands from FlexMPI central controllers. These commands include the performance of dynamic reconfiguration, load balance and activation/deactivation of the monitoring service.

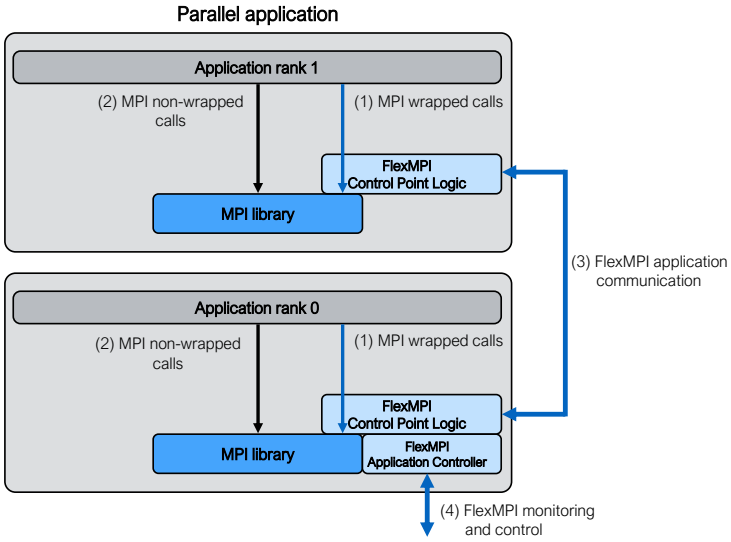

Figure 4. Example of an MPI application executed with the FlexMPI library.

3.3. Clarisse Interference-Aware Logic

Algorithm 3 shows the interference-aware logic. This algorithm requires, for each new running application, the related performance metrics (S_i^{init}) for when it is being executed without interference. These metrics include application user, system, communication times and other performance-counter related values. There are two ways of obtaining these metrics. The first one is from a previous execution of the application. In this case, function $query\_metrics()$ returns true, and $recorded\_metrics()$ fetches the metrics from a record.

If these metrics are not available, $FlexMPI\_Monitor()$ instruments the application and collects the metrics during its execution. This monitoring is only active for a short period of time when the application starts executing. In order to obtain performance metrics unaffected by interference, the application is initially executed in exclusive nodes. Then, when the metrics are obtained, the processes running in the last assigned node (the one that runs a number of processes smaller than the compute-node capacity) are migrated to a shared compute node. Note that this step could be avoided if the performance metrics are already available from a previous execution.

The $notify\_scheduler()$ function (line 7) returns information provided by the monitor about compute nodes ($node_i$) with potential hot-spots. The class of interference is coded in the $interference$ variable and, according its value, different actions are taken.

For memory-related interference, one conflicting application is immediately migrated to another node in order to decrease the used RAM. For cache and network-related interference, by means of $FlexMPI\_Monitor()$, all the conflicting applications are monitored—obtaining, for each one of them, the current performance metrics S_i^{curr}. Function $evaluate\_interference()$ determines if any applications in the compute node have a slowdown comparing the metrics against the original ones. If there are no changes in performance, then the interference signal is subsequently inhibited for a certain amount of time or until a new application is executed in the node. In an alternative scenario that, in our experiments, corresponds to a performance degradation of over 10%, one application

is selected (app_j) and migrated to an exclusive node. There are several policies for selecting the migrated application. In our current implementation of the framework, the policy consists of migrating the latest executing application. The migration operation is performed by two steps, first creating Δp processes (by means of malleability) in the new exclusive node provided by the monitor (lines 16 and 17) and then removing Δp processes (lines 18 and 19) from the shared node where the conflict exists. These operations rely on the FlexMPI controller for performing the application reconfiguration. In addition, the scheduler is notified in order to update the running application's status.

Algorithm 3 CLARISSE interference-aware algorithm.

1: **if** $query\_metrics(app_i)$ **then**
2: $S_i^{init} = recorded\_metrics(app_i)$
3: **else**
4: $S_i^{init} = FlexMPI\_Monitor(app_i, 0)$
5: **end if**
6: **while** (1) **do**
7: **if** $\{node_i, interference\} = notify\_scheduler()$ **then**
8: **if** $interference == RAM$ **then**
9: $app_j = latest\_application\_id(node_i)$
10: **else if** ($interference == CACHE \ || \ interference == NET$) **then**
11: $S_i^{curr} = FlexMPI\_monitor(app_i, interference) \quad \forall app_i \in node_i,$
12: $app_j = evaluate\_interference(S^{init}, S^{curr})$
13: **end if**
14: **if** $app_j \neq 0$ **then**
15: $\Delta p = num\_processes(app_j, node)$
16: $nodes = allocate(app_j, \Delta p, 1)$
17: $FlexMPI\_spawn(nodes, app_j, \Delta p, 0)$
18: $nodes = return\_scheduler(app_j, -\Delta p, 0)$
19: $FlexMPI\_remove(nodes, app_j, -\Delta p, 0)$
20: **end if**
21: **end if**
22: **end while**

4. Evaluation

In this section, we present the platform and applications used as well as the experiments performed and the results obtained from them.

4.1. Experimental Environment

For the experiments, we have used a heterogeneous cluster of eight nodes divided in two racks. The connection between nodes in the same rack is a 10 Gbps Ethernet, whilst the connection between racks is made through a 1 Gbps Ethernet. The cluster contains two nodes with Intel(R) Xeon(R) E5 with eight cores and 256 GB of RAM in one rack and six nodes with Intel(R) Xeon(R) E7 with 12 cores and 128 GB of RAM in the other.

As use cases, we have chosen a collection of applications and real and synthetic kernels. All of them have been integrated with FlexMPI. Table 1 shows the use cases' characteristics. *EpiGraph* [9] is a parallel and stochastic simulator of the propagation of the flu virus. It is currently configured to carry out the simulation in Bilbao, Spain, using an un-directed weighted graph of 703,258 nodes and 8,806,520 edges that corresponds to the individual-connections in the simulation. *Jacobi* is the kernel of the Jacobi iterative method that operates with dense matrices.

Table 1. Use cases considered in the evaluation.

Code	Name	Class	Access Pattern	Size	Intensive on
A	EpiGraph	Application	irregular	small	CPU & network
B_{ll}, B_{hl}	CG	Kernel	irregular	small	CPU
C	Jacobi	Kernel	regular	medium	CPU
D_m	CPU	Synthetic	regular	medium	CPU
D_{xl}	CPU	Synthetic	regular	large	CPU & memory
E	CPUNET	Synthetic	regular	medium	CPU & network
F	IMEM	Synthetic	irregular	medium	memory

CG is the kernel of the *Conjugate Gradient* iterative method that operates with sparse matrices and performs sparse-matrix-vector multiplications (SpMV). In order to analyze the impact of data locality on interference, two different input matrices have been used. The first one, executed with code B_{ll} (CG kernel with *l*ow *l*ocality), is a square sparse matrix with 500,000 rows and 39,996,827 non-zero elements. The non-zero entries are randomly distributed across all the matrix and produce low data locality on the vector accesses during the SpMV. The second matrix, executed with code B_{hl} (CG kernel with *h*igh *l*ocality), corresponds to a random sparse matrix with 500,000 rows and 39,967,238 non-zero elements. The non-zero entries are randomly distributed conforming a block diagonal matrix with a block size of 20,000 entries. This pattern provides much better data locality on the vector accesses.

We have also used a set of kernels to characterize main features. *CPU* is a synthetic kernel that is similar to Jacobi but without communications. It represents a pure CPU application with two subclasses: D_m with a medium memory footprint that uses six 20,000-entry square dense matrices and D_{xl} (e*x*tra-*l*arge) that operates with six 50,000-entry square dense matrices (120 GB). CPUNET is a variation of the D_m kernel that alternates CPU and communication-intensive phases. This kernel is used for creating and evaluating network interference. *IMEM* is a memory-intensive kernel that accesses several matrices using indirections. It is used to create interference in the processor cache memory. The kernel operates with six 20,000-entry square dense matrices (19.2 GB).

4.2. Interference-Aware Scheduling

We have first evaluated the ad hoc use case that illustrates the impact of each class of interference as well as the overhead related to the proposed framework. For the experiments, the code was compiled with gcc 7.4.0 and MPICH 3.2. This use case consists of a sequence of applications that are executed in three shared compute nodes. Table 2 shows the structure and results for this use case. The execution order (also used as application id) is shown in the first column of the table. For each code, the table includes the name, number of processors and memory footprint of each application. *Shared* label shows the application id that is executed in the same compute node. For example, applications 1 and 2 are located in the same shared node, producing an increase in the last-level cache miss ratio detected by the monitor. Once the notification is received by the scheduler, FlexMPI is used to monitor both applications. T1, T2 and T3 are the execution times before the interference, during the interference and after the interference, respectively. We can observe in the table that application 1 doubles the execution time while application 2 is unaffected (mainly because it does not have temporal data locality). In order prevent this performance degradation, application 2 is moved to an exclusive node. The reconfiguration overhead quantifies the cost of moving an interfering application to another compute node. This overhead is shown in the table in the column labelled "Overhead". Note that this overhead corresponds to four reconfigurations: (1) creating eight processes in the shared node, (2) destroying eight processes in the initial exclusive node, and, following this, after the analysis, (3) creating eight processes in the new exclusive node, and (4) destroying eight processes in the shared node. Note that the first two reconfigurations can be avoided if the performance metrics of

the application were previously recorded. For application 2, the overhead is mainly related to the data redistribution.

Table 2. Use case example of interference. T1, T2, T3 are the execution times per 10 iterations. Overhead is the application reconfiguration overhead measured in seconds.

Id.	Code	Procs.	Size (Gb)	Shared	T1 (s)	T2(s)	T3(s)	Overhead	# Iterations
1	B_{ll}	4	0.3	2	3.2	6.4	3.2	-	15,000
2	F	20	17.9	1	29.6	29.2	29.1	32.2	600
3	B_{hl}	4	0.3	4	2.8	2.8	2.8	-	12,000
4	F	20	17.9	3	29.5	29.6	29.5	-	500
5	E	20	1.2	6–7	6.9	9.0	7.0	-	400
6	E	20	1.2	5	7.0	8.78	7.6	2.8	4000
7	D_m	20	1.2	5	0.03	0.03	0.03	-	2,000,000
8	D_{xl}	6	94.6	9	21.0	27.8	21.6	-	500
9	D_{xl}	16	111.8	8	9.4	9.6	9.4	101.1	1500

Applications 3 and 4 produce a conflict in a shared node that does not produce any performance degradation (due to the existence of good data locality for both of them), thus no reconfiguration is needed. Applications 5 and 6 produce communication interference that affects both of them negatively. Application 6 is migrated to an exclusive node that is located in a different rack, increasing the final execution time (T3) compared with the initial one (T1) due to a slower network bandwidth. Given the small amount of data, the overhead is mainly related to process creation/destruction. After this migration, application 7 is placed in the same node as 5, producing no degradation given that they have different profiles.

Finally, applications 8 and 9 almost reach the memory capacity in the shared compute node. This is detected by the monitor and application 9 is immediately migrated to a free node. Note that the overhead is more significant because of the large amount of data used by the applications. Taking into account the performance improvement by avoiding the interference (T3 vs. T2 values), the overhead penalty can be overcome in a few iterations.

Figure 5 shows the performance evaluation of the framework for Scenario A that consists of 20 jobs that are executed as a workflow. Each job is an independent application. The x-axis represents the application name and the y-axis is the execution time per 100 iterations. Note that some applications, like 4(A) in Scenario A, may have a much smaller execution time per iteration than the others, but their impact on the overall time might be important due to executing a larger number of iterations. The threshold used by the monitor for generating a hot-spot was a memory use of over 90%, an overall last-level cache miss ratio greater than 40%, and a network use greater than 40% of the maximum network capacity.

Scenario A is a medium-conflict workflow with jobs of classes A, B_{LL} and F. Note that the performance of the first two is degraded by F, and F is unaffected by interference. Figure 6 shows the Gantt diagram associated with this scenario. For this scenario, six conflicting applications created hot-spots that resulted in six cases of performance degradation. Six performance-degraded cases were detected and avoided, except the *7 (B_{LL})* case in which the degradation did not exceed the predefined threshold and no action was taken. These cases are displayed in the figure with two bars. The leftmost one represents the execution times when interference occurs (before redistribution, which is related to the conflicting application), and the rightmost one represents the final time, after avoiding the interference. In order to show that *7 (B_{LL})* did not improve the execution time, the representation is the opposite: the initial time is the execution without interference and the final time is longer because it corresponds to one with interference. The rest of the jobs do not exhibit performance degradation. Consequently, the second bar (after the redistribution) is not shown for these jobs for the sake of clarity.

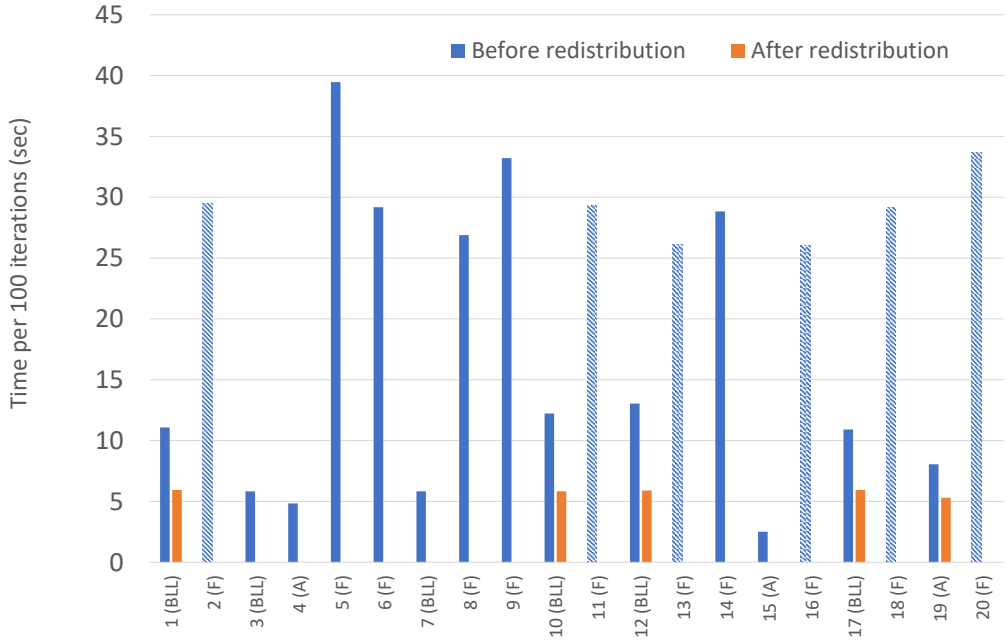

Figure 5. Framework evaluation for Scenario A. Each bar shows the execution time per 100 iterations of each application. Applications with striped bars are applications that create interference. Applications with two bars exhibit change in the execution time due to interferences.

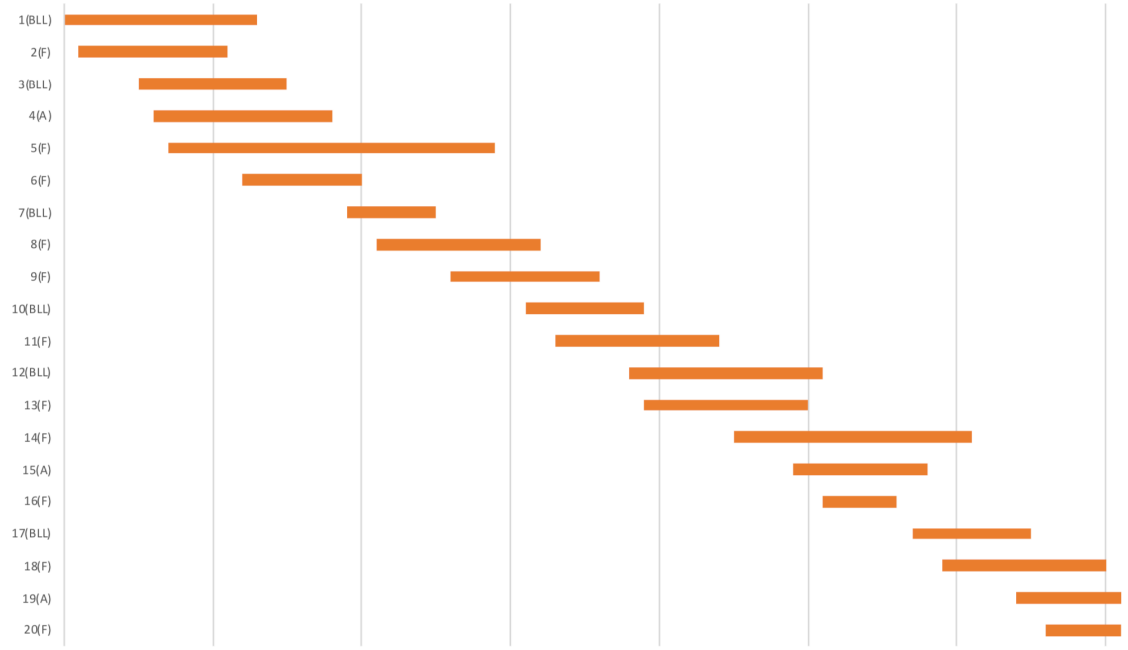

Figure 6. Gantt diagram of Scenario A when CLARISSE is not active and, consequently, the interferences are not avoided. The length of diagram corresponds to the workflow makespan with a value of 14.889 s.

The interfering applications are the ones that, after commencing the execution, reduce the performance of other jobs that were already running in the same compute nodes. These interfering applications are displayed in the figure with striped bars. During the workflow execution, these applications are identified by CLARISSE and are migrated to a spare compute node. Note that the execution time per iteration of the interfering applications is the same after and before the migration process, thus only one striped bar is displayed. The migration has some overhead related to process

creation and data redistribution. Overall, the total overhead (for all the applications of the workflow) was of 17.8 s per process creation/destruction and 183.6 for data redistribution.

Figure 7 shows a diagram with the CPU use of each compute node during the workflow execution. We assume that the user has requested five compute nodes to run the workflow and the scheduler has allocated nodes 1 to 5 for this task. In addition, nodes 6 and node 7 represent spare compute nodes that are only employed by CLARISSE for the execution of conflicting applications. Node 8 (not shown in the figure) is used to execute the shared processes of the application exclusively in order to obtain the initial performance metrics. This evaluation takes a short time and, following this, these processes are migrated to a shared node. In total, the workflow makespan is 11,909 s (including overheads) and the total energy consumption (taking into account the eight compute nodes) is 6.9 MJ.

Figure 8 shows system use when CLARISSE is not used and the conflicts are not avoided. In this case, the makespan is 14,889 s. There are two reasons for the increment of the makespan in relation to the previous strategy. First, given that the conflicts are not avoided, the conflicting applications take longer to complete their execution. Second, nodes 6 and 7 are not used. Note that there is a trade-off between the amount of computational resources involved in the execution and the application execution time. For this scenario, the total energy consumption is 7.7 MJ. Despite using less computational resource, the increase in the conflicting application execution time produces a larger amount of energy consumption.

Figure 9 shows system use when each application is executed in exclusive nodes. Here, the applications do not experience performance degradation due to the lack of conflicts. However, some computational resources may be underused. For example, nodes 1 and 2 have a reduced workload as some jobs only use two processes. Now, the makespan is 15,277 s and the energy consumption is 7.9 MJ. Note that both values are larger than the previous policies that are based on sharing the compute nodes.

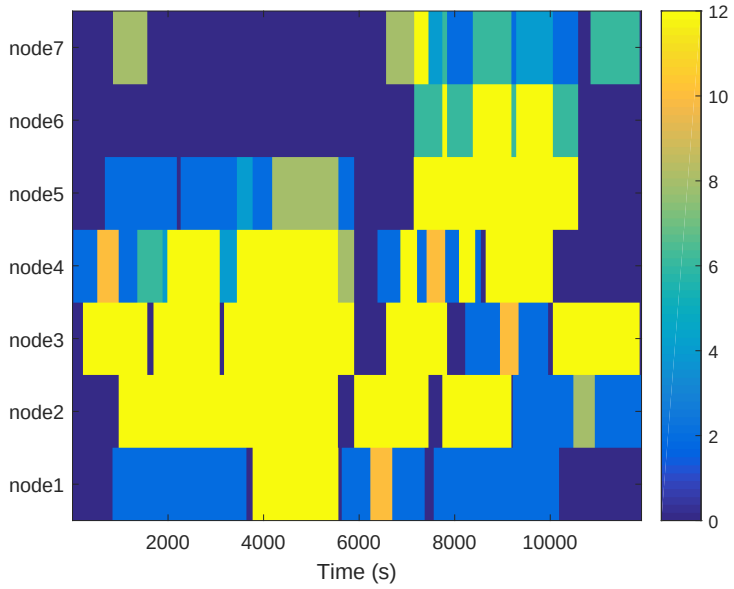

Figure 7. Overall system load for Scenario A using a shared node policy assignment with CLARISSE for avoiding interferences.

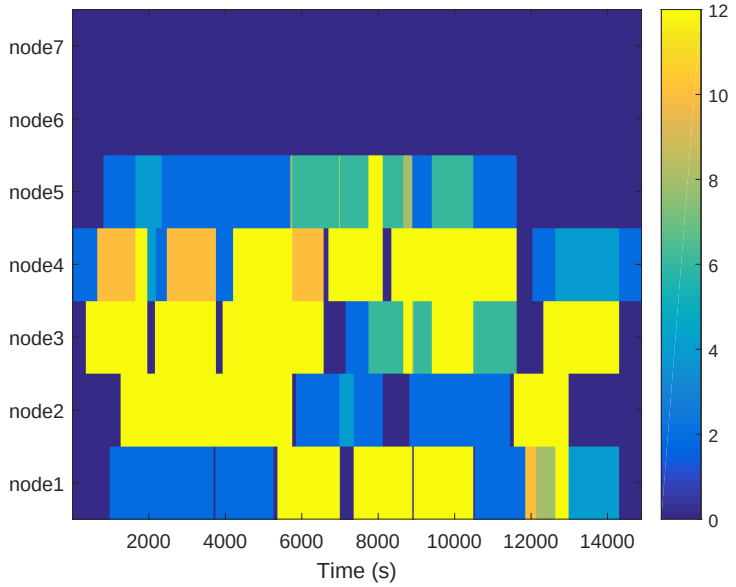

Figure 8. Overall system load for Scenario A using a shared node assignment policy without CLARISSE, thus interferences are not avoided.

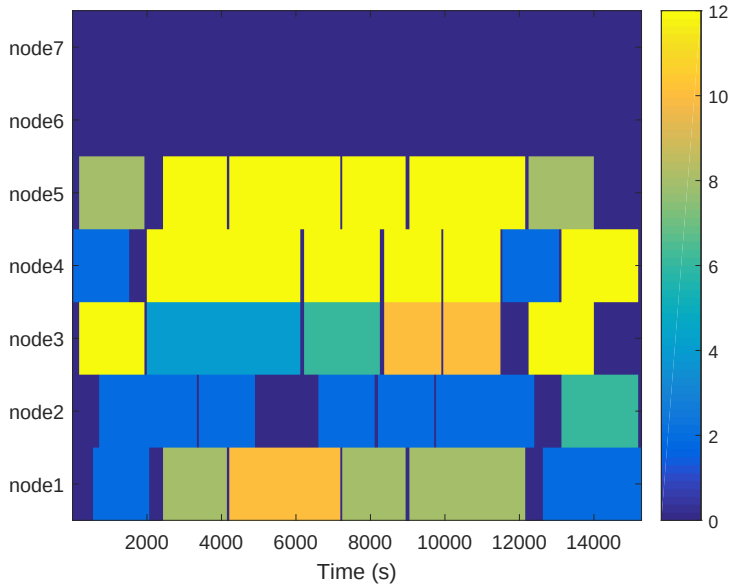

Figure 9. Overall system load for Scenario A using an exclusive node assignment policy.

Scenario B consists of a mixed workload with different applications and input matrices that produce a larger number of conflicts. The idea in this scenario is to evaluate CLARISSE under more intense conditions. In this scenario, nine applications create interference that degrades the performance of the other six applications.

As shown in Figure 10, the combined work of the monitor and scheduler is able to detect these situations and migrate the conflicting applications. The migration overhead is 31.3 s for process creation and 501.9 for data redistribution. For applications $13(B_{LL})$ and $16(B_{LL})$, there are several jobs that produce conflict with them. More specifically, jobs $14(DM)$, $15(DM)$, $17(DM)$ *and* $20(F)$. The scheduler migrates all of them, but, because of their large number, the original applications ($13(B_{LL})$ and $16(B_{LL})$) cannot be executed without conflicts for a significant amount of time. Despite this, the makespan and energy consumption of this scenario is 10,759 s and 6.3 MJ. These values are smaller than those

in which the conflicts are not taken into account (that produce a makespan of 13,256 s and energy of 6.9 MJ) or when the jobs are executed exclusively (makespan of 14,929 s and energy of 7.4 MJ). Overall, CLARISSE provides a more efficient execution in both scenarios. Figure 11 shows the Gantt diagram associated with this scenario.

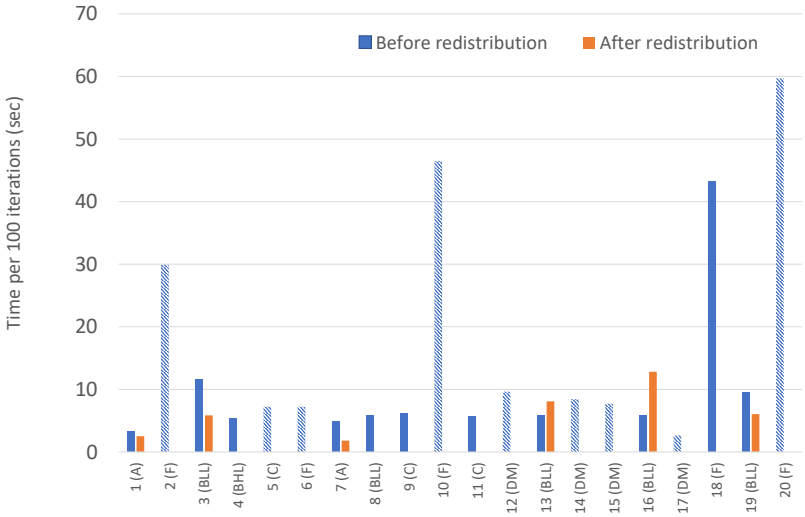

Figure 10. Framework evaluation for Scenario B. Each bar shows the execution time per 100 iterations of each application. Applications with striped bars are applications that create interference. Applications with two bars exhibit change in the execution time due to interferences.

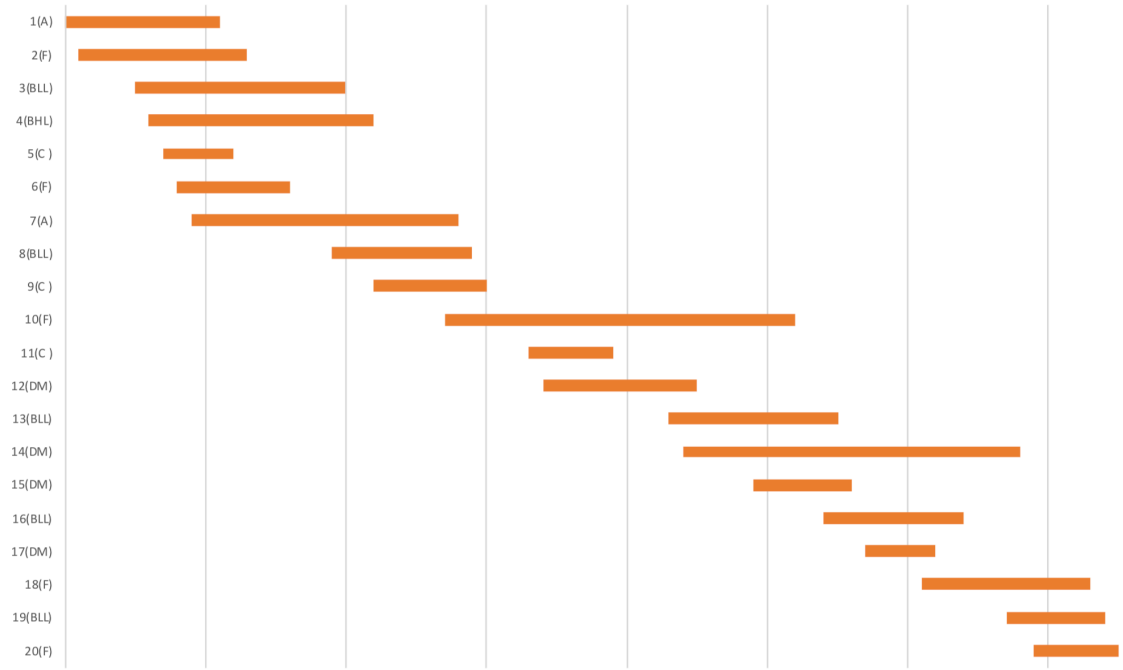

Figure 11. Gantt diagram of Scenario B when CLARISSE is not active and, consequently, the interferences are not avoided. The length of diagram corresponds to the workflow makespan with a value of 13.256 s.

4.3. Daemon Monitor

In this section, we present a quantitative analysis of LIMITLESS's performance. We evaluate the monitor overhead as well as the effect of different monitor features on reducing the network usage. In order to evaluate the effect of the in-node analysis, we have used a benchmark executed in

two nodes of the cluster, one to manage the LDS and another to evaluate the LDM. The experiment consists of three different compute phases: up until minute 10, the compute node is idle. Then, a CPU benchmark with a constant workload is executed for ten minutes. Finally, at minute 20, another CPU benchmark with a variable workload is executed. The CPU workload of the compute node is shown in Figure 12.

Figure 13 shows the network traffic related to the monitor when in-node analysis is enabled as well as when in-node analysis is disabled. The results of the experiment show that the in-node analysis reduces the network traffic dramatically (up to 90% in phases 1 and 2). Even with variable load (phase 3), almost 50% of the monitor traffic can be reduced. Note that there is a trade-off between the tolerance value and the amount of information that the server receives. In our experiments, we found that tolerance values between 5% and 10% provided accurate measurements with important reductions in network use.

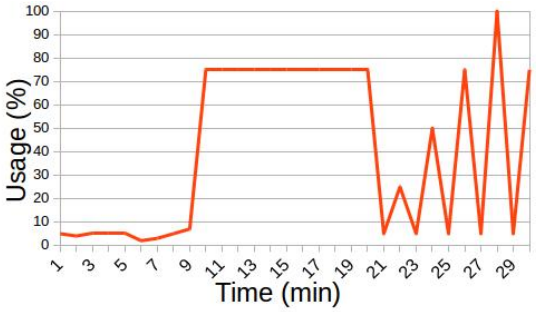

Figure 12. CPU evolution during evaluation of the tracks.

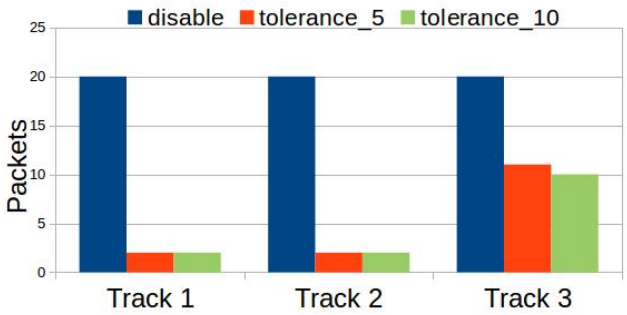

Figure 13. Network traffic with/without in-node analysis of redundancy with two different tolerances.

In Figure 14, the CPU load of a single compute node for a 10% tolerance is represented for a period of 24 h. During this period, the previous scenarios A and B were running in the system. Figure 15 represents the difference between the metrics obtained with 10% tolerance and without tolerance. Note that these values are the error produced by using the tolerance threshold. As we can see, the biggest error obtained in some samples is 10% , but, on average, is 0.27%—and the total percentage of metrics with an error is 5.6%. In terms of network traffic, the use of tolerance drastically decreases the amount of packets sent from the monitors to the aggregator. In this example, this number decreases by more than 87% (from 5314 packets to 680). In our experiments, we found that tolerance values between 5% and 10% provide accurate measurements with important reductions in the network use.

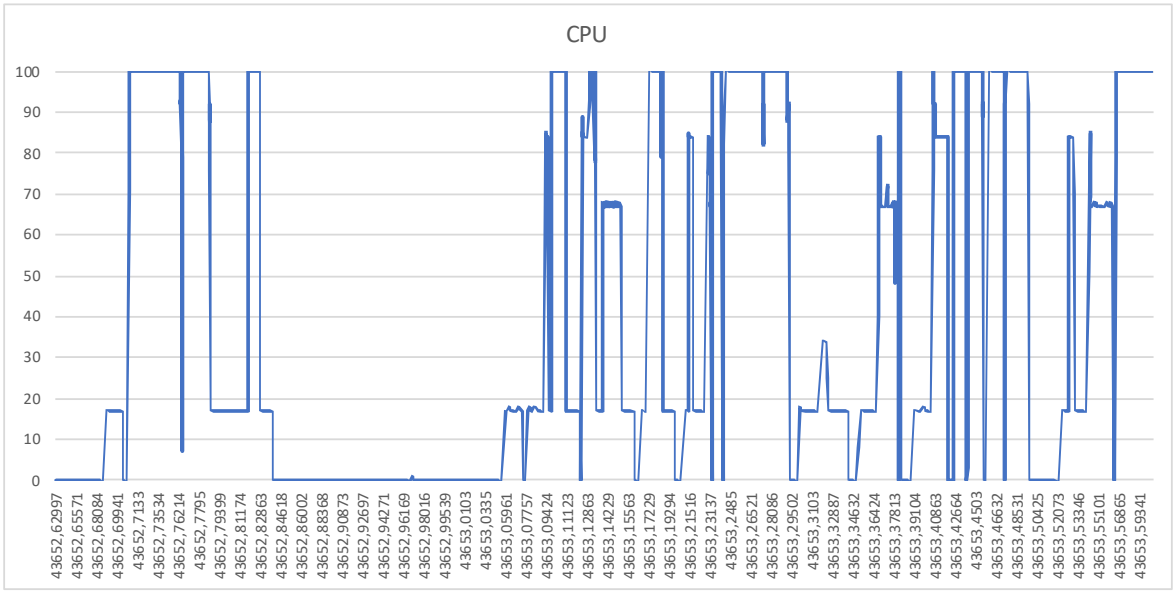

Figure 14. Collected CPU use with a 10% tolerance for a single compute node during a 24 h period.

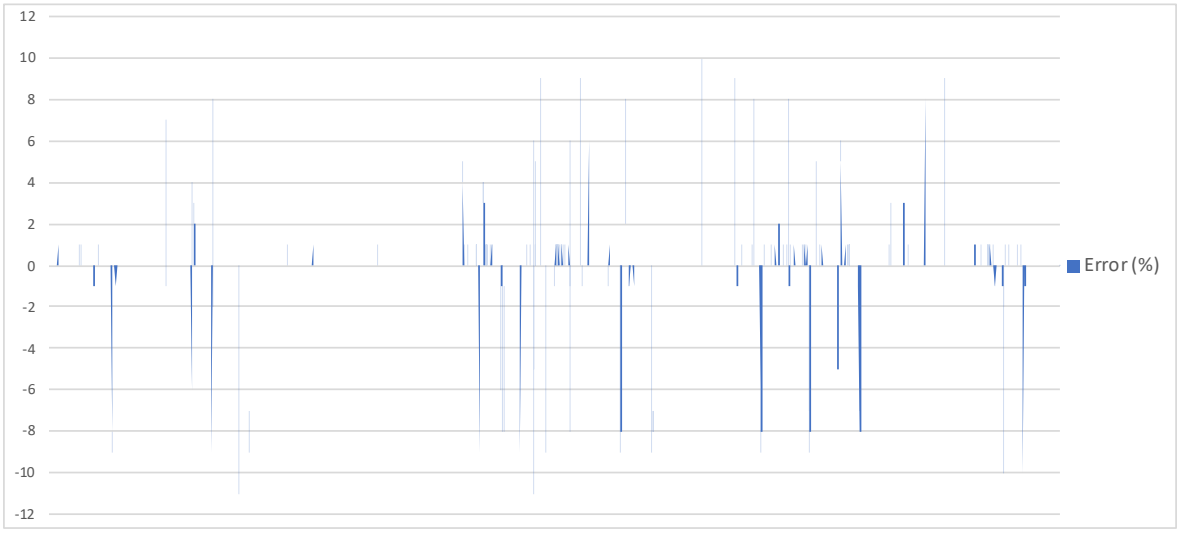

Figure 15. Error expressed as the difference of CPU use for a single compute node between the use of 10% (Figure 14) and 0% tolerances.

LIMITLESS scalability has been tested by simulation using the tool described in [10] and OMNET++ [11]. Simulation results show that both server and aggregators support connection with up to 200 aggregators and monitors, respectively. This corresponds to the worst case configuration scenario with samples collected every second and a server with just one processor thread. This means that scalability will increase for servers with more processor threads and larger intervals between successive samples.

The global overhead of the monitoring module depends on the interval time for measuring. Our experimentation has been done with three different values, the minimum interval time that is one second, a sort interval time that is 5 s, and an acceptable interval time for a general purpose that is 10 s. The worst case for processing is the first, and the monitoring module consumes less than 1.0% of CPU. The rest of the cases, with higher values of interval time, consume less CPU percentage. Regarding the memory usage, it is despicable because the value is below 0.1%.

5. Related Work

System monitoring for large-scale HPC platforms is a difficult task, which becomes increasingly challenging as the scale and complexity of the infrastructure increases. From a technological point of view, monitoring information in HPC systems has been addressed by many approaches and tools. One example of this is Ganglia [12], which is one of the most-used monitoring tools for HPC systems. It uses a listen/announce protocol for the monitor state and a point-to-point connected tree system between representative nodes, in order to make groups from them. One feature of Ganglia is that all the nodes receive information from the others. This increases system reliability because in the case of failure, the system (or a part of it) is updated, but it implies an important quantity of overhead.

Exascale requires new monitoring techniques, such as sub-optimal period scheduling [13] and strong usage of HPC system statistics [14] to improve system utilization. Thus, the effort on monitors development has been continuous in HPC systems. A daemon is collected [15] that collects system (and application) performance metrics and provides different mechanisms to store them. Nagios [16] is an open source solution for monitoring networks of hosts and services. It is a well-known framework for gathering monitoring information in large-scale systems. However, Nagios is not designed for HPC systems, thus its scalability in large-scale platforms is unclear.

In [17], authors present a system that they named *Distributed Modular Monitoring system* (DiMMon). The main features of this tool are: data can be sent by different paths and with different purposes, dynamic reconfiguration of the modes, and the capability to calculate performance metrics of an individual job while data are being collected. Its architecture is based on monitoring agents. These agents are processes, but they represent an execution environment which allows monitoring, connection between nodes and message transmissions. In DiMMon, each node can be different, with distinct tasks and metrics, which makes it adaptive.

In [18], authors present the Supermon architecture, a flexible set of tools for cluster monitoring with features like high-speed communications and scalability, thanks to improvement of the rstad performance. It uses a data protocol based on symbolic expressions, from individual nodes to entire clusters. For this reason, the author states that Supermon is scalable and can run in a heterogeneous clusters.

Monitoring clusters and LSDS is an important theme, and there are plentiful related works, more than we have described above. For example, Agelastos et al. produced some similar works. In [19], the authors put the focus on the results obtained after doing *profiling* (at a system and application level) based on global monitoring in an HPC cluster. They collect metrics as we have done, and use a hierarchical model to transmit the data from the compute nodes to aggregating nodes. The results are provided through their tool LDMS [20], and it gets the metrics directly from the kernel system */proc* and */sys*. In [20], a tool named LDMS (Lightweight Distributed Metric Service) is presented, which allows the collection of data, and its transmission and storage. The system is composed of three different kinds of components: *aggregators*, *samplers* and *storage*. Even though there are three kinds of nodes, all of them have the same code and the only difference is the configuration.

There are further related works such as Open-SpeedShop [21] that shows some lack in efficiency, adaptability and scalability. In addition, to improve this lack, the authors use MRNet [22] to create a communication infrastructure based on trees.

In [23], authors propose static scheduling for multiple periodic applications consisting of strictly and non-strictly periodic tasks on time-critical completion time. It is an interesting job because the solution is based on an improved Mixed Integer Linear Programming (MILP) method in order to obtain an optimal scheduling solution. Our work differs as we present a real-time scheduling based on monitoring, and our framework is able to do that scheduling with all tasks, not only periodic tasks.

Authors in [24] present an interesting point of view on energy-aware scheduling based on a hybrid genetic algorithm. Using genetic algorithms is one of the most powerful techniques for solving optimization problems and they apply their solution as a combinatorial optimization problem.

In [25], the authors exploit modern multi-core and many-core processors with threading techniques such as OpenMP and Pthreads that are related to linear algebra operations in HPC, and they

propose a dynamic load-balancing method to reduce the penalty of the NUMA effect in those kinds of operations. This paper is relevant for our work because it could be combined with our framework, not to schedule applications but to balance the load between different parallel tasks.

HPC system information, obtained via a monitoring service, plays an important role in scheduling in order to share resources and provide high-performance computing. A dynamic meta-scheduling architecture model for large scale distributed systems based on monitoring has been described in [26]. In this work, the MonALISA service is used in combination with ApMON to collect customized information to provide automated decisions for improving task scheduling. A distributed resource monitoring and prediction architecture was presented in [27] allowing the detection of the best set of machines to run an application based on the collected information and the result of a prediction algorithm, which evaluates the potential performance of a node.

In this work, we present a closer coordination between the monitor and scheduler based on an interactive refinement of the monitoring that is performed first at a system-wide level and, if necessary, at application-level. The motivation is to estimate possible performance degradation and performance hotspots caused by sharing multicore processors when running HPC workloads [28], and create a system to mitigate poor scaling conditions created by shared-resource contention by using co-scheduling of HPC applications designed for large multicore systems [29].

Co-scheduling inside the Linux kernel for bulk synchronous parallel applications has been proposed in [30]. However, this work did not include monitorization information. Co-scheduling of CPU and memory intensive applications in the same node using monitoring information has been proposed in [31] to improve energy efficiency and overall throughput of a supercomputer. Monitoring information was used to provide a detailed characterization of HPC applications for co-scheduling [32]. Different alternatives are Tetris [33] and LoTES [34], which consider constraints in CPU, memory, disk, and network bandwidth for packing tasks and improving the cluster efficiency. Sedighi et al. present in [35] the Fairness, Utilization and Dynamicity (FUD) theorem to balance scheduling parameters in shared computing Environments. Alternatives have also been proposed for distributed systems to provide resource-aware hybrid scheduling algorithms in heterogeneous distributed computing [36][37].

Malleability is a major topic in Cloud computing and virtualized environments. A framework for elastic execution of existing MPI programs was proposed in [38] for cloud frameworks. This framework also monitors the performance looking for loaded resources. If a severe conflict is detected, the MPI job is terminated and a new program is restarted on a different number of instances. Our approach is different, as we do not kill the MPI job, but block the process as it is moved. Restarting a new job could require waiting for a long time in a real-world HPC cluster.

Another proposal for automatic resource elasticity for high performance applications in the cloud has been presented in [39]. Its differential approach consists of providing elasticity for high performance applications without user intervention or source code modification applying aging. The performance increase obtained a range of 26%.

Blagodurov et al. proposed in [40] a methodology to provide contention aware scheduling in HPC cluster environments when jobs are concurrently executing on the same multicore node. However, different to our proposal, they assume a virtualized HPC cluster, where applications are deployed using virtual machines that can be easily migrated from the contended servers of the MPI cluster. However, the performance penalization is high and they don't get a performance increase. In addition, their scheduler does not consider contention effects in its scheduling decisions and it is also not able to migrate the load across the cluster on-the-fly. This is one of the LIMITLESS's features.

An idea similar to our proposal is presented in [41], including a mechanism for enabling shrink/expand capability in the parallel runtime system using task migration and dynamic load balancing, checkpoint-restart, and Linux shared memory. However, the scheduler proposed is not contention aware, as they assume dedicated resources for each MPI process.

Our work differs from the previous ones as we introduce a framework that provides coordination between system-wide and application-level monitoring, scheduling and malleability in

physical clusters to provide resource-aware scheduling when jobs are commenced and throughout their execution. Our approach provides better performance than previous approaches and does not need virtualization. Moreover, we avoid killing the MPI job and starting a new one, which could create a long waiting time to get resources.

6. Conclusions

This work is aimed to address some of the European research priorities in the area of HPC technology. In the presented framework, we introduce four different components that work in a coordinated way: LIMITLESS that provides system-wide monitoring; CLARISSE that performs data-staging coordination and control; FlexMPI that provides application malleability and monitoring; and the job scheduler. Thanks to the close coordination of the components, it is possible to obtain a holistic vision of the platform and take global performance-oriented actions. In this work, we present a technique that permits different applications to execute in the same compute nodes avoiding any interference between them. Experimental results show that is is possible to improve application performance as well as overall platform throughput, and, consequently, to use system resources more efficiently, reducing the consumed energy. Note that, in this work, we assume that there are only intra-node interference hazards.

As a future work, we plan to extend this work to consider I/O interference, including solutions, like I/O scheduling [4], in this framework. We also plan to include new performance metrics (like I/O bandwidth) in the scheduler decisions and to improve the application placement using a more refined analysis that takes into account the overhead of the reallocation process.

Author Contributions: Conceptualization, J.C. and D.E.S.; methodology, J.C.; software, D.E.S. and A.C.; validation, J.C., D.E.S. and A.C.; formal analysis, J.C. and D.E.S.; investigation, J.C., D.E.S. and A.C; resources, J.C.; writing: original draft preparation, J.C., D.E.S. and A.C.; writing: review and editing, J.C., D.E.S. and A.C.; supervision, J.C.; funding acquisition, J.C.

Funding: This work was partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under the grant TIN2016-79637-P "Towards Unification of HPC and Big Data Paradigms" and the European Union's Horizon 2020 research and innovation program under Grant No. 801091, project "Exascale programming models for extreme data processing" (ASPIDE).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Ari, I.; Kocak, U. Hybrid Job Scheduling for Improved Cluster Utilization. In *Euro-Par 2013: Parallel Processing Workshops*; Springer: Berlin/Heidelberg, Germany, 2014; pp. 395–405.
2. Yoo, A.B.; Jette, M.A.; Grondona, M. Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*; Springer: Berlin/Heidelberg, Germany, 2003; pp. 44–60.
3. Cha, K.; Maeng, S. Reducing communication costs in collective I/O in multi-core cluster systems with non-exclusive scheduling. *J. Supercomput.* **2012**, *61*, 966–996. [CrossRef]
4. Isaila, F.; Carretero, J.; Ross, R. CLARISSE: A Middleware for Data-Staging Coordination and Control on Large-Scale HPC Platforms. In Proceedings of the 16th International Symposium on Cluster, Cloud and Grid Computing (CCGrid), Cartagena, Colombia, 16–19 May 2016; pp. 346–355.
5. Cascajo, A.; DaeMon—User Manual. Available online: https://www.arcos.inf.uc3m.es/acascajo/daemon/ (accessed on 12 July 2019).
6. Martín, G.; Singh, D.E.; Marinescu, M.C.; Carretero, J. Enhancing the performance of malleable MPI applications by using performance-aware dynamic reconfiguration. *Parallel Comput.* **2015**, *46*, 60–77. [CrossRef]
7. Martín, G.; Marinescu, M.C.; Singh, D.E.; Carretero, J. FLEX-MPI: An MPI Extension for Supporting Dynamic Load Balancing on Heterogeneous Non-dedicated Systems. In *Euro-Par 2013 Parallel Processing*; Wolf, F., Mohr, B., An Mey, D., Eds.; Springer: Berlin/Heidelberg, Germany, 2013; pp. 138–149.
8. Singh, D.E.; Martín, G.M.; Marinescu, M.C.; Carretero, J. FlexMPI Source Code Software. 2019. Available online: http://www.arcos.inf.uc3m.es/flexmpi/ (accessed on 21 April 2019).

9. Martin, G.; Singh, D.E.; Marinescu, M.C.; Carretero, J. Towards efficient large scale epidemiological simulations in EpiGraph. *Parallel Comput.* **2015**, *42*, 88–102. [CrossRef]
10. Núñez, A.; Fernández, J.; Filgueira, R.; García, F.; Carretero, J. SIMCAN: A flexible, scalable and expandable simulation platform for modelling and simulating distributed architectures and applications. *Simul. Model. Pract. Theory* **2012**, *20*, 12–32. [CrossRef]
11. Varga, A. OMNeT++. In *Modeling and Tools for Network Simulation*; Wehrle, K., Gunes, M., Gross, J., Eds.; Springer: Berlin/Heidelberg, Germany, 2010; pp. 35–59, doi:10.1007/978-3-642-12331-3_3.
12. Massie, M.L.; Chun, B.N.; Culler, D.E. The ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Comput.* **2004**, *30*, 817–840. [CrossRef]
13. Jones, W.M.; Daly, J.T.; DeBardeleben, N. Application Monitoring and Checkpointing in HPC: Looking Towards Exascale Systems. In Proceedings of the 50th Annual Southeast Regional Conference ACM-SE '12, Tuscaloosa, AL, USA, 29–31 March 2012; ACM: New York, NY, USA; pp. 262–267.
14. Evans, T.; Barth, W.L.; Browne, J.C.; DeLeon, R.L.; Furlani, T.R.; Gallo, S.M.; Jones, M.D.; Patra, A.K. Comprehensive Resource Use Monitoring for HPC Systems with TACC Stats. In Proceedings of the First International Workshop on HPC User Support Tools HUST '14, New Orleans, LA, USA, 16–21 November 2014; pp. 13–21.
15. Forster, F.; Harl, S. Collectd—The System Statistics Collection Daemon. 2012. Available online: https://collectd.org/(accessed on 28 March 2019).
16. Nagios Enterprises LLC; Nagios—The Industry Standard In IT Infrastructure Monitoring. Available online: https://www.nagios.org/(accessed on 17 May 2019).
17. Stefanov, K.; Voevodin, V.; Zhumatiy, S.; Voevodin, V. Dynamically Reconfigurable Distributed Modular Monitoring System for Supercomputers (DiMMon). *Procedia Comput. Sci.* **2015**, *66*, 625–634. [CrossRef]
18. Sottile, M.J.; Minnich, R.G. Supermon: A high-speed cluster monitoring system. In Proceedings of the IEEE International Conference on Cluster Computing, Chicago, IL, USA, 23–26 September 2002; pp. 39–46. [CrossRef]
19. Agelastos, A.; Allan, B.; Brandt, J.; Gentile, A.; Lefantzi, S.; Monk, S.; Ogden, J.; Rajan, M.; Stevenson, J.; Continuous whole-system monitoring toward rapid understanding of production HPC applications and systems. *Parallel Comput.* **2016**, *58*, 90–106. [CrossRef]
20. Agelastos, A.; Allan, B.; Brandt, J.; Cassella, P.; Enos, J.; Fullop, J.; Gentile, A.; Monk, S.; Naksinehaboon, N.; Ogden, J.; et al. The Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications. In Proceedings of the SC '14: International Conference for High Performance Computing, Networking, Storage and Analysis, New Orleans, LA, USA, 11–16 November 2014; pp. 154–165. [CrossRef]
21. Schulz, M.; Galarowicz, J.; Maghrak, D.; Hachfeld, W.; Montoya, D.; Cranford, S. Open | SpeedShop: An Open Source Infrastructure for Parallel Performance Analysis. *Sci. Program.* **2008**, *16*, 105–121. [CrossRef]
22. Roth, P.C.; Arnold, D.C.; Miller, B.P. MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools. In Proceedings of the SC '03: 2003 ACM/IEEE Conference on Supercomputing, Phoenix, AZ, USA, 15–21 November 2003; p. 21. [CrossRef]
23. Jiang, X.; Huang, K.; Zhang, X.; Yan, R.; Wang, K.; Xiong, D.; Yan, X. Energy-Efficient Scheduling of Periodic Applications on Safety-Critical Time-Triggered Multiprocessor Systems. *Electronics* **2018**, *7*, 98. [CrossRef]
24. Mahmood, A.; Khan, S.; Albalooshi, F.; Awwad, N.; Mahmood, A.; Khan, S.A.; Albalooshi, F.; Awwad, N. Energy-Aware Real-Time Task Scheduling in Multiprocessor Systems Using a Hybrid Genetic Algorithm. *Electronics* **2017**, *6*, 40. [CrossRef]
25. Su, X.; Lei, F.; Su, X.; Lei, F. Hybrid-Grained Dynamic Load Balanced GEMM on NUMA Architectures. *Electronics* **2018**, *7*, 359. [CrossRef]
26. Pop, F.; Dobre, C.; Stratan, C.; Costan, A.; Cristea, V. Dynamic Meta-Scheduling Architecture Based on Monitoring in Distributed Systems. In Proceedings of the 2009 International Conference on Complex, Intelligent and Software Intensive Systems, Fukuoka, Japan, 16–19 March 2009.
27. Rajkumar, S.; Rajkumar, N.; Suresh, V.G. Automated object counting for visual inspection applications. In Proceedings of the International Conference on Information Communication and Embedded Systems (ICICES2014), Chennai, India, 27–28 February 2014.

28. Dwyer, T.; Fedorova, A.; Blagodurov, S.; Roth, M.; Gaud, F.; Pei, J. A practical method for estimating performance degradation on multicore processors, and its application to hpc workloads. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Salt Lake City, UT, USA, 10–16 November 2012; p. 83.
29. Bhadauria, M.; McKee, S.A. An approach to resource-aware co-scheduling for CMPs. In Proceedings of the 24th ACM International Conference on Supercomputing, Tsukuba, Japan, 2–4 June 2010; ACM: New York, NY, USA; pp. 189–199.
30. Jones, T. Linux kernel co-scheduling for bulk synchronous parallel applications. In Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers, Tucson, AZ, USA, 31 May 2011; ACM: New York, NY, USA, 2011; pp. 57–64.
31. Breitbart, J.; Weidendorfer, J.; Trinitis, C. Case Study on Co-scheduling for HPC Applications. In Proceedings of the 2015 44th ICPP Conference Workshops, Beijing, China, 1–4 September 2015; pp. 277–285.
32. Weidendorfer, J.; Breitbart, J. Detailed characterization of hpc applications for co-scheduling. In Proceedings of the 1st COSH Workshop on Co-Scheduling of HPC Applications, Prague, Czech Republic, 19 January 2016; p. 19.
33. Grandl, R.; Ananthanarayanan, G.; Kandula, S.; Rao, S.; Akella, A. Multi-resource Packing for Cluster Schedulers. *SIGCOMM Comput. Commun. Rev.* **2014**, *44*, 455–466. [CrossRef]
34. Tran, T.T.; Padmanabhan, M.; Zhang, P.Y.; Li, H.; Down, D.G.; Beck, J.C. Multi-stage Resource-aware Scheduling for Data Centers with Heterogeneous Servers. *J. Sched.* **2018**, *21*, 251–267. [CrossRef]
35. Sedighi, A.; Smith, M.; Deng, Y. FUD—Balancing Scheduling Parameters in Shared Computing Environments. In Proceedings of the 2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud), New York, NY, USA, 26–28 Jun 2017; p. 363.
36. Klusáček, D.; Rudová, H.; Baraglia, R.; Pasquali, M.; Capannini, G. Comparison of multi-criteria scheduling techniques. In *Grid Computing*; Springer: Berlin/Heidelberg, Germany, 2008; pp. 173–184.
37. Vasile, M.A.; Pop, F.; Tutueanu, R.I.; Cristea, V.; Kołodziej, J. Resource-aware hybrid scheduling algorithm in heterogeneous distributed computing. *Future Gener. Comput. Syst.* **2015**, *51*, 61–71. [CrossRef]
38. Raveendran, A.; Bicer, T.; Agrawal, G. A Framework for Elastic Execution of Existing MPI Programs. In Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, Anchorage, AK, USA, 16–20 May 2011; pp. 940–947. [CrossRef]
39. Da Rosa Righi, R.; Rodrigues, V.F.; Da Costa, C.A.; Galante, G.; De Bona, L.C.E.; Ferreto, T. AutoElastic: Automatic Resource Elasticity for High Performance Applications in the Cloud. *IEEE Trans. Cloud Comput.* **2016**, *4*, 6–19. [CrossRef]
40. Blagodurov, S.; Fedorova, A. Towards the contention aware scheduling in HPC cluster environment. *J. Phys. Conf. Ser.* **2012**, *385*, 012010. [CrossRef]
41. Gupta, A.; Acun, B.; Sarood, O.; Kalé, L.V. Towards realizing the potential of malleable jobs. In Proceedings of the 2014 21st International Conference on High Performance Computing (HiPC), Goa, India, 17–20 December 2014; pp. 1–10. [CrossRef]