Storage-Heterogeneity Aware Task-based Programming Models To Optimize I/O Intensive Applications

Hatem Elshazly, Jorge Ejarque, and Rosa M. Badia

Abstract—Task-based programming models have enabled the optimized execution of the computation workloads of applications. These programming models can take advantage of large-scale distributed infrastructures by allowing the parallel and distributed execution of applications in high-level work components called *tasks*. Nevertheless, in the era of Big Data and Exascale, the amount of data produced by modern scientific applications has already surpassed terabytes and is rapidly increasing. Hence, I/O performance became the bottleneck to overcome in order to achieve more total performance improvement.
New storage technologies offer higher bandwidth and faster solutions than traditional Parallel File Systems (PFS). Such storage devices are deployed in modern day infrastructures to boost I/O performance by offering a fast layer that absorbs the generated data. Therefore, it is necessary for any programming model targeting more performance to manage this heterogeneity and take advantage of it to improve the I/O performance of applications.
Towards this goal, we propose in this paper a set of programming model capabilities that we refer to as *Storage-Heterogeneity Awareness*. Such capabilities include: (i) abstracting the heterogeneity of storage systems, and (ii) optimizing I/O performance by supporting dedicated I/O schedulers and an automatic data flushing technique.
The evaluation section of this paper presents the performance results of different applications on the MareNostrum CTE-Power heterogeneous storage cluster. Our experiments demonstrate that a storage-heterogeneity aware programming model can achieve up to almost 5x I/O performance speedup and 48% total time improvement compared to the reference PFS-based usage of the execution infrastructure.

Index Terms—Heterogeneous Storage Systems, Task-based Programming Models, I/O Intensive Applications, I/O scheduling, Task Scheduling, Automatic Data Movement, Heterogeneity Abstraction, Resource Pooling, Checkpointing

1 Introduction

Modern scientific and big data applications applications process and generate huge amounts of data [1]. These applications aim for resilience (e.g., checkpointing the applications intermediate data to enable restart after failure) [2]. In addition to that, storage systems that rely on Parallel File Systems (PFS) (e.g., Lustre [3] or GPFS [4]) face significant challenges in terms of limited performance [5]. Therefore, applications have gone through a paradigm shift where improving I/O performance becomes critical for enhancing the whole application performance [6].

As a response to the need for absorbing large amounts of data and optimizing I/O performance, large-scale systems have incorporated newly emerging storage technologies such as Non-Volatile RAM (NVRAM) and Solid-State Drivers (SSD) into their underlying base storage system of the PFS. These storage devices can help reducing the gap between compute and I/O performance because of their high I/O bandwidth and low latency capabilities. They act as *Burst Buffers* [7] that absorb the data produced by applications in their I/O-dominant phase. This approach can improve applications I/O performance by providing a fast solution to write data from memory and enhancing applications reliability, for instance, through faster checkpointing.

Even though this heterogeneity in the storage systems design would benefit applications I/O performance, it comes with additional complexity that could prevent achieving enhanced I/O performance [8]. Each storage device needs to be provisioned according to its own capabilities in order to achieve maximum performance. For instance, because of the limited capacity of each device, I/O workload should be distributed in a manner to optimize overall I/O execution. In addition to that, each storage device has to be provisioned for bandwidth to avoid the problem of I/O congestion that negatively impacts the performance [9].

Due to the lack of sufficient mechanisms and techniques to optimize I/O performance in traditional task-based programming models (see Section 2), the aforementioned complexities are exposed to application programmers. It becomes their responsibility to carry the burden of planning and optimizing applications execution on heterogeneous storage systems by manually deciding which and how much data should be written to each storage device. Such an approach can be possible with applications that exhibit a small I/O workload. However, it is prohibitive in I/O intensive applications that periodically produce large amounts of data with varying sizes. Leading not only to a complex development process, but also to possible underutilization of the storage system and wasted performance

• *All authors are affiliated with the Barcelona Supercomputing Center (BSC), Barcelona, Spain.*
E-mail: {hatem.elshazly,jorge.ejarque,rosa.m.badia}@bsc.es

© 2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes,creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. http://dx.doi.org/10.1109/TPDS.2022.3161123

improvement opportunities.

In this paper, we address the need for seamlessly and transparently managing heterogeneous storage devices and taking advantage of them to optimize I/O performance. To this end, we propose enabling *Storage Heterogeneity-Awareness* in task-based programming models. The main idea herein is twofold: from the one hand, abstract all the details of the storage infrastructure from applications. From the other hand, provide programming models mechanisms and scheduling techniques to exploit the heterogeneity of the storage system to improve performance.

Following this approach has the following advantages:

- First, reducing infrastructure complexities and easing applications development. Application code becomes agnostic to the underlying storage system. Hence, no code modifications are required to adapt to changes in the storage infrastructure.
- Second, maximizing I/O performance given a set of storage devices with different capabilities.

The main contributions of this paper can be summarized as follows:

- A proposal to enable task-based models to abstract the heterogeneity of storage systems.
- Dedicated I/O schedulers that target the optimization of the I/O phases in applications.
- An automatic data movement flushing mechanism to maximize the utilization of the storage system.

We demonstrate the benefits of the aforementioned storage-heterogeneity awareness proposals by using a prototype implementation of these capabilities in the PyCOMPSs task-based programming model [10]. All the experiments were run on an execution platform with heterogeneous storage systems: The MareNostrum CTE-Power Cluster of the Barcelona Supercomputing Center. Our experiments show significant performance improvements that reached up to 5x I/O performance speedup and 48% total time improvement compared to the reference PFS-based implementation.

The rest of this paper is organized as follows: Section 2 presents the related work. Section 3 discusses the main programming model abstractions that we used to realize our proposals. Section 4 introduces our proposed storage-heterogeneity awareness capabilities. Sections 5 and 6 present our proposed I/O task schedulers and the flushing mechanism respectively. The performance results are presented and analyzed in Section 7. Finally, Section 8 summarizes the main conclusions of this paper.

2 Related Work

Task-based programming models such as Parsl [11], Dask [12] and Luigi [13] among others, offer different mechanisms to improve computing performance of applications. However, they do not provide similar support that specifically targets I/O performance optimization.

Previous research efforts targeted maximizing I/O performance on heterogeneous storage systems. Hermes [14] and UniviStor [15] present a system for I/O buffering and optimizing data movement between different storage and memory layers. DataWarp [16] and Data Elevator [17] are burst buffer management software that enhance applications writes by redirecting them to remote-shared burst buffers from PFS. Harmonia [18] is a system-wide scheduler to minimize cross-application interference. TRIO [19] is an orchestration framework to efficiently transfer checkpointing dataset to PFS.

In contrast to previous work, we propose in this paper an ***application-level programming model support*** to transparently optimize application performance based on its execution pattern. Unlike system-level solutions and middleware, our proposal does not require any system administration efforts nor external software installation. Hence, increasing applications portability. Ideally, our proposed programming model should be used along side a system-wide solution, specially in an environment where multiple applications are running to minimize cross-application interference and improves overall system utilization.

In addition to that, the aforementioned efforts support flushing mechanisms that are triggered when a file close operation is detected even if the file is needed for later computation. On the other hand, our proposed flushing mechanism is triggered when no I/O activity is being done in the application to avoid interference, and overlaps data transfer with computation. Our proposed mechanism only flushes data that are not required for future tasks executions, hence, tasks take advantage of the bandwidth offered by the storage layer where the data reside.

I/O libraries such as MPI-IO [20], HDF5 [21], ADIOS [22] enable I/O optimization based on access patterns and data sizes. We consider that our proposal is complementary to such libraries. Indeed, our proposal can be used to provide coarse-grained parallelism over distributed and heterogeneous infrastructure, whereas the aforementioned I/O libraries can provide fine-grained parallelism inside tasks. An example of this usage and its performance benefits can be found in [23].

3 Programming Model Abstractions

In order to realize the proposals of this paper, we relied on the main abstractions that are provided by the PyCOMPSs task-based programming model [10]. The objective of PyCOMPSs is to allow the distributed execution of applications without compromising their programmability and ease of development.

Functions can be declared as tasks by the functions can be declared as tasks by the means of the `@task` decorator of the PyCOMPSs programming model (Listing 1). In the `@task` decorator, the return types of the task outputs and the directionality of the task parameters have to be specified. The directionality of a certain parameter describes how it will be accessed inside the task/function code: read (`IN`), updated (`INOUT`) or written (`OUT`). Furthermore, task parameters can be files, in which case the directionality should be specified to indicate if the file will be: read (`FILE_IN`), updated (`FILE_INOUT`) or written (`FILE_OUT`). The directionality parameters are used later by PyCOMPSs to identify dependencies between tasks. All data dependencies and data transfer between tasks are done by the PyCOMPSs runtime in a transparent manner to application developers.

```
@task(data=IN, returns=list)
def compute(data):
    # perform computation
    ...
```

Listing 1: Task Decorator of PyCOMPSs

The PyCOMPSs programming model also provides a set of I/O awareness abstractions to optimize I/O execution [24]. Such abstractions separate the handling of I/O and compute workloads in terms of scheduling and execution by introducing two concepts:

- *I/O tasks* that exclusively perform I/O operations. If dependency-free, their execution can overlap with the execution of compute tasks that exclusively perform computation.
- *Storage Bandwidth Constraints* to mitigate I/O contention.

Listing 2 depicts an example of an I/O task in PyCOMPSs. The `@IO` decorator (Line 3) is added on top of the `task` decorator (Line 4) to declare the `checkpoint` function as an I/O task. Moreover, the `@constraint` decorator (Line 1) is used to specify certain requirements of task execution such as storage bandwidth (`storageBW`) and the storage size (`storageSize`). The `storageBW` can be set to `auto`, in this case, the programming model launches an automatic mechanism for setting and auto-tuning the constraint value to improve I/O performance by minimizing I/O congestion.

```
@constraint(storageBW="auto",
            storageSize=..)
@IO()
@task(filename=FILE_OUT)
def checkpoint(filename, data):
    # perform I/O
    ...
```

Listing 2: I/O Task in PyCOMPSs

4 Storage Heterogeneity Awareness

Defining a task-based model as storage-heterogeneity aware means that it is able to exploit the capabilities of the underlying storage infrastructure to improve I/O performance in a transparent manner to application developers. More specifically, Storage-heterogeneity awareness describes two main capabilities:

- Heterogeneity Abstraction.
- Optimizing I/O performance.

Each of these capabilities is discussed in the following sections.

4.1 Heterogeneity Abstraction

We propose to abstract the heterogeneity of the underlying storage system and expose it as a single pooled resource. In such a resource pool, different storage devices can be arranged into layers according to their capabilities such that: the top layer would offer the highest bandwidth and lowest capacity, whereas the bottom layer provides the lowest bandwidth and highest capacity.

Figure 1 shows the two approaches of handling heterogeneous storage systems. The traditional heterogeneous non-aware approach (Figure 1(a)) exposes all the storage devices to application developers. With this approach, users manually specify the full path of the storage device to write the data to it. There is no programming model support to control the assignment of tasks to different storage devices. Whereas our proposed heterogeneous-aware approach (Figure 1(b)) organizes the storage devices into one pooled resource. It becomes the responsibility of the programming model to exploit the capabilities of the storage infrastructure to optimize applications execution.

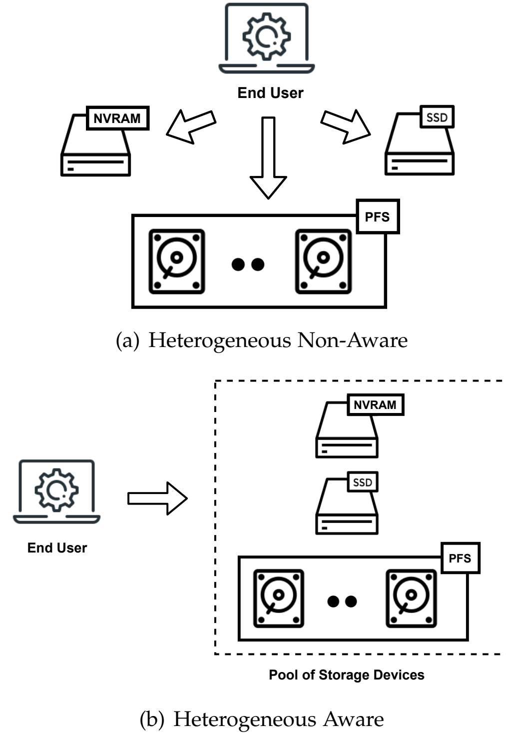

Fig. 1: Different Views Of Storage Systems Heterogeneity

In our implementation, the PyCOMPSs runtime loads the information about available storage devices and their capabilities at application launch time from a resource description XML file provided by the user.

4.2 I/O Performance Optimization

In addition to heterogeneity abstraction, heterogeneous-aware task-based systems should optimize the execution of applications by scheduling I/O tasks in a manner to exploit the capabilities of the different storage devices.

We propose dedicated I/O schedulers that are responsible for scheduling I/O tasks. Each scheduler tries to optimize I/O tasks execution taking advantage of the heterogeneous-aware view of storage infrastructures. The main idea is to increase I/O parallelism by scheduling the I/O tasks to layers that provide high bandwidth without causing I/O contention.

Dedicated I/O schedulers separate the scheduling of compute tasks from the scheduling of I/O tasks. I/O tasks are scheduled taking into consideration their storage requirements and the available storage resources of each device. Therefore, I/O workload is scheduled in a manner that optimizes its execution without affecting the scheduling decisions of the compute workload and vice versa. Section 5 introduces all our proposed I/O schedulers.

Furthermore, in order to maximize the reuse and utilization of higher storage layers, we propose an automatic data movement mechanism to flush the data from higher storage layers to lower storage layers. Consequently, continuously freeing the capacities of higher storage layers so that more tasks can be scheduled to them. This mechanism is described in detail in Section 6.

Figure 2 illustrates the use of the programming model abstractions described in Section 3 to abstract storage heterogeneity from application code. The main part of the application code (Line 14) contains multiple calls to two tasks: (i) A `calculate` task which is computing a certain value. (ii) A `checkpoint` I/O task, which is writing the data that has been produced by the `calculate` task. From the point of view of the programming model, the task dependency graph is built according to the data dependencies. At scheduling time, the `calculate` task is assigned to the compute scheduler whereas the `checkpoint` task is scheduled by the I/O scheduler. The details of where the I/O tasks have been scheduled and where the files are written are completely hidden from the application code. In the task code, files can be accessed as if they were in the current working directory (Line 9), i.e., without having to specify or know the complete path of the storage device .

In this design, All bookkeeping operations are performed by the PyCOMPSs runtime without any intervention from the users such as keeping track of data locations. Also, performing data transfers across storage layers and worker nodes, and monitoring the current state of the infrastructure (i.e., remaining capacity, available bandwidth).

Note that manually setting the constraints to be as high as possible can negatively impact the performance because of the possible sequential execution of tasks. This is explained in detail in our previous work [24].

Indeed, the proposals discussed in this paper can be adopted by other task-based programming models. This requires separating I/O from computation in terms of scheduling and execution as described in [24]. Then, the concepts of this paper can be implemented to exploit heterogeneous infrastructures.

5 I/O TASKS SCHEDULING

The main goal of I/O tasks scheduling is to take advantage of the bandwidth capabilities of the storage devices in the system to maximize I/O task parallelism while avoiding I/O congestion. Consequently, improving performance.

This section starts by describing the scheduling model that is followed in all of our proposed schedulers (Section 5.1). Then it introduces different I/O schedulers: Section 5.2 presents the *First Come First Served* scheduler. Whereas Sections 5.3 and 5.4 describe a *Modified Priority* and *Backfilling* schedulers respectively.

```
1  @task(returns=list)
2  def calculate(input_data):
3      ...
4
5  @constraint(storageBW=.., storageSize=..)
6  @IO()
7  @task(filename=FILE_OUT)
8  def checkpoint(filename, data):
9      fh = open(filename, 'w')
10     ...
11
12 if __name__ == "__main__":
13     ...
14     for count, in_data in enumerate(input_list):
15         out_data = calculate(in_data)
16         checkpoint("out_{0}".format(count), out_data)
17     ...
```

Application Code

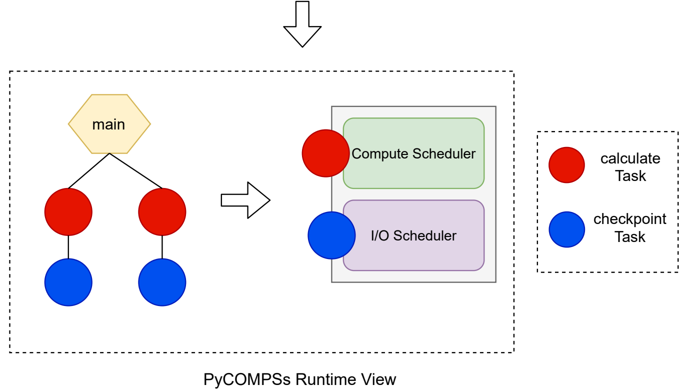

Fig. 2: Programming Model Support For Storage-Heterogeneity Awareness

5.1 Scheduling Model

I/O schedulers can make optimized scheduling decisions based on execution time objectives. Such objectives can be specified as avoiding I/O congestion and maximizing the utilization of higher layers in the storage hierarchical organization.

The scheduling routine is presented in Algorithm 1. It tries to schedule an I/O task t to the highest possible layer of one of the workers in the workers set W. In all workers, Storage devices are organized into layers by the PyCOMPSs runtime as described in Section 4.1. Line 2 defines $W_{candidates}$ which is the set of workers that can currently host task t execution. Line 3 retrieves task t storage requirements, i.e., storage bandwidth BW_{Req} and capacity C_{Req}. In Lines 4 through 12, every storage layer on each worker is considered. For each layer l on worker w, the storage parameters of the layer are retrieved, i.e., current available bandwidth BW_{Avail} and capacity C_{Req} (Line 6). Line 7 describes how the decision of whether to schedule the I/O task t to a certain storage layer l on worker w is made. Such a decision depends on two execution time conditions:

1) Whether the bandwidth requested by the task (BW_{Req}) exceeds the current available bandwidth of the storage layer (BW_{Avail}).
2) Whether there is enough capacity (C_{Avail}) on this layer to satisfy the task required capacity (C_{Req}).

If the aforementioned conditions are met, worker w is added to the set of candidates that can host the task

execution $W_{candidates}$. Line 9 skips the rest of the layers on the current worker because it is guaranteed that the next layers provide less bandwidth.

Line 13 checks if the set of candidate workers $W_{candidates}$ contains any worker. On the one hand, if it does not contain any worker then this means that the requirements of task t cannot be satisfied by any of the storage layers of any worker. Hence, the scheduler decides not to launch the task and waits to the next scheduling iteration, so that enough bandwidth becomes available when some of the currently running tasks finish execution.

On the other hand, if $W_{candidates}$ contains candidate workers, then Line 16 retrieves the candidate worker w_{target} that has the highest available storage layer, and its storage layer l_{target}. Finally, the bandwidth and capacity of the target layer l_{target} are updated (Lines 18, 19).

Algorithm 1: Scheduling Algorithm

Input : t as I/O task, W as the set of available workers
Output: $True$ if the task can be scheduled, $False$ otherwise

```
1  Function Schedule(t, W):
2      W_candidates ⟵ ∅
3      BW_Req, C_Req ⟵ getStorageReqs(t)
4      foreach w ∈ W do
5          foreach l ∈ Layers_w do
6              BW_Avail, C_Avail ⟵
                 getAvailStorageParams(l)
7              if BW_Req ≤ BW_Avail and C_Req ≤ C_Avail
                 then
8                  W_candidates.insert(w)
9                  Continue
10             end
11         end
12     end
13     if W_candidates == ∅ then
14         return False
15     else
16         w_target, l_target ⟵
             getBestCandidate(W_candidates)
17         BW_Avail, C_Avail ⟵
             getAvailStorageParams(l_target)
18         BW_Avail ⟵ BW_Avail − BW_Req
19         C_Avail ⟵ C_Avail − C_Req
20         return True
21     end
22 End Function
```

5.2 First Come First Served I/O Scheduler

The First Come First Served (FCFS) scheduler can be considered as a homogeneous I/O scheduler, i.e., it assumes that all the I/O tasks in the application write the same amount of data and require the same amount of bandwidth. Similar to a traditional FCFS job scheduler, this I/O scheduler schedules I/O tasks depending on the order in which they arrive.

The pseudocode in Algorithm 2 describes the details of how the FCFS I/O scheduler works. For all the tasks in the task set T, try to schedule current task t_i on all the workers W by calling the `Schedule` routine that is described in Algorithm 1. Each storage layer is considered in terms of current available capacity and storage bandwidth, if either properties does not satisfy the task requirements, then the next layer is considered. If the current task t_i can be scheduled, then it is added to the set of scheduled tasks T_{Sched} to be launched for execution. Otherwise, task t_i waits in the queue until the next scheduling iteration.

Algorithm 2: First Come First Served I/O Scheduler

Input : T as the set of I/O Tasks ready for scheduling, W as the set of available workers
Output: T_{Sched} set of scheduled I/O tasks

```
1  Function FCFS(T, W):
2      foreach t_i ∈ T do
3          if Schedule(t_i, W) then
4              add t_i to T_Sched
5          end
6      end
7      return T_Sched
8  End Function
```

5.3 Modified Priority I/O Scheduler

This scheduler is similar to a classical priority scheduler. Tasks are scheduled according to a certain priority. In our case, a task has a higher priority if it has a higher storage bandwidth requirement.

We modified the behaviour of the priority scheduler to maximize the utilization of higher storage layers. If a task has been already scheduled to a certain storage layer of a certain worker, this task will not be immediately launched to the assigned layer, instead, it will wait until its requirements become available on a higher storage layer. The decision of whether to immediately launch the task or not is based on whether there will be an execution time benefit from waiting until a higher storage layer becomes available.

Algorithm 3 depicts the pseudocode for the priority scheduler. The first difference between this scheduler and the FCFS scheduler is that I/O tasks have to be sorted according to their bandwidth requirement before the start of the scheduling process. Hence, while going through the storage layers of the workers from top to bottom, the

scheduler first considers tasks with higher bandwidth for each layer. If a task cannot be scheduled to a certain layer then the scheduler aborts until a currently running I/O task finishes execution and releases bandwidth resources.

Algorithm 3: Priority Scheduler

Input : T as the set of I/O Tasks ready for scheduling, W as the set of available workers
Output: T_{Sched} as the set of scheduled I/O tasks

```
1  Function PRIORITY(T, W):
2      T_Sorted ⟵ sortBWDescending(T)
3      foreach t_i ∈ T_Sorted do
4          if Schedule(t_i, W) then
5              if canMaximize(t_i, W) then
6                  Break
7              else
8                  add t_i to T_Sched
9              end
10         else
11             Break
12         end
13     end
14     return T_Sched
15 End Function
```

Another notable difference in Algorithm 3 is the *canMaximize* routine (Line 5). The purpose of this routine is to check if a task can benefit from waiting until a higher layer can host it.

Algorithm 4 shows the pseudocode of the *canMaximize* routine. For all available resources, it goes through all the storage layers that are higher than the task's assigned storage layer $l_{assigned}$. For each layer, the benefit of making the task wait until a higher layer becomes available is checked (Line 9). This is done by comparing the time a task has to wait to be executed on the candidate layer (w_l) added to the average execution time on the candidate layer (e_l), to the average execution time on the assigned layer ($e_{assigned}$). If there is a time benefit from not immediately scheduling the task, then the scheduler returns to the main scheduling routine in Algorithm 3.

The waiting time for a task, i.e., the time a task has to wait to be executed on a certain storage layer, is calculated by calling the `estimateWaitingTime` routine (Line 8). This routine identifies the waiting time by calculating the minimum remaining execution time of all the tasks currently running on that storage layer. The following formula can be used to calculate the remaining execution time of a running task on a certain storage layer l:

$$remainingTime_t = averageTime_t - startTime_t$$

where:

$remainingTime_t$: remaining execution time of task t on the storage layer.

$averageTime_t$: average execution time of task t on the storage layer.

$startTime_t$: start execution time of task t on the storage layer.

Algorithm 4: Storage Layer Maximization

Input : t as I/O task, W as set of available workers
Output: $True$ If it is better to wait for higher layer, $False$ otherwise

```
1  Function canMaximize(t, W):
2      l_assigned ⟵ getAssignedStorageLayer(t)
3      foreach w ∈ W do
4          foreach l ∈ Layers_w do
5              if l ≥ l_assigned then
6                  Continue
7              else
8                  waitingTime ⟵
                     estimateWaitingTime(l)
9                  if waitingTime + e_l < e_assigned then
10                     return True
11                 end
12             end
13         end
14     end
15     return False
16 End Function
```

5.4 Backfilling I/O Scheduler

The backfilling scheduler allows other tasks to be scheduled and launched as long as they will not delay the start of the tasks in the head of the scheduling queue. Unlike the priority scheduler, if a task cannot run because its storage requirement is not satisfied, then the next task in the scheduling queue is considered for scheduling. Tasks with less bandwidth are scheduled and launched if and only if they will not delay the start of the waiting task.

Algorithm 5 shows the pseudocode of the backfilling scheduling policy. Similar to the priority scheduler, all tasks are sorted according to their bandwidth requirements in a descending order.

The pseudocode in Algorithm 5 describes three situations that should be considered when a task t_i is being considered for scheduling:

- If the current task t_i can be scheduled and t_i is the waiting task from the previous scheduling iterations, this means that this iteration is a new scheduling iteration and t_i is not a waiting task anymore (Lines 7-11). Therefore, task t_i can be added to the scheduled tasks T_{Sched} to be launched for execution and the algorithm proceeds to schedule the following remaining tasks.

Algorithm 5: Backfilling Scheduler

Input : T as the set of I/O Tasks ready for scheduling, W as the set of available workers
Output: T_{Sched} as the set of scheduled I/O tasks

```
1  Function BACKFILLING(T, W):
2      T_Sorted ⟵ sortBWDescending(T)
3      waitingTask ⟵ Null
4      waitingTime ⟵ 0
5      foreach t_i ∈ T_Sorted do
6          if Schedule(t_i, W) then
7              if waitingTask != Null and
                 waitingTask == t_i then
8                  waitingTask ⟵ Null
9                  add t_i to T_Sched
10                 Continue
11             end
12             e_estimated ⟵
                 getEstimatedExecutionTime(t_i)
13             if waitingTask == Null or
14             (waitingTask != Null and
                 e_estimated < waitingTime) then
15                 add t_i to T_Sched
16                 Continue
17             end
18         else
19             if waitingTask == Null then
20                 waitingTask ⟵ t_i
21                 waitingTime ⟵
                     estimateWaitingTime()
22                 Continue
23             end
24         end
25     end
26     return T_Sched
27 End Function
```

- If there is no $waitingTask$ (Line 13), tasks are scheduled directly. Otherwise, a task t_i will only be scheduled if and only if its execution on the target layer ($e_{estimated}$) will not delay the waiting task (Line 14). The estimated execution time can be identified by keeping a profile of previous task execution times on each storage layer.
- Finally, if a task cannot be scheduled (Lines 18-24), then t_i will be marked as the waiting task if there is no previously assigned waiting task at the head of the tasks set. The $waitingTime$ of a task is estimated by the system by a call to the `estimateWaitingTime` routine that is similar to the one described in the priority scheduling policy.

6 Automatic Flushing Mechanism

During application execution, the capacities of higher storage layers get consumed until no more tasks can be scheduled to these layers anymore. As a result, in the next scheduling iterations, tasks will be scheduled to the bottom layers that have more free capacity but offer less bandwidth. Hence, less task parallelism can be achieved.

In order to overcome this issue and maximize the reuse and utilization of higher storage layers, we propose an automatic data flushing mechanism. The main idea of this mechanism is to transparently and periodically flush the data written in previous I/O tasks executions from higher storage layers to lower storage layers. The programming model will only flush the data that are not be needed by any future task execution, i.e., final data. Such information is possible to acquire because the programming model already has information about the data dependencies between tasks. As the system continuously frees up capacities in higher storage layers, more I/O tasks can be scheduled to them in the next I/O scheduling iterations. Consequently, increasing task parallelism and improving I/O performance.

Algorithm 6 presents the pseudocode of the flushing mechanism. D is the set of output data of all executed tasks so far. For each data d, the pseudocode is checking if there are any tasks that will require d by a call to the `hasReaders` routine (Line 3). If there are no future task executions that have d as input, the pseudocode retrieves the current storage layer on which d is stored (Line 6). Then, each storage layer gets considered starting from the bottom/lowest layer ($BottomLayerIndex$) to the layer directly below where the data currently resides ($currentLayer + 1$). Data d will be flushed if and only if there is enough capacity on the layer i to store it (Lines 9-14).

Due to the possible impact of the flushing mechanism on the I/O performance because the transfer of data between different storage layers, our prototype implementation in PyCOMPSs automatically starts the mechanism when there is no detected I/O activity, i.e., no I/O tasks are running nor scheduled to run on any of the storage layers.

7 Evaluation

This section presents the evaluation of a storage heterogeneity-aware design in terms of performance and programmability.

Section 7.1 starts with describing the infrastructure of the execution platform. Section 7.3 discusses the results with an application that exhibits homogeneous I/O workload. Whereas Section 7.4 presents the results of a real application that exhibits heterogeneous workload, in addition to a synthetic application to demonstrate the differences between the modified priority and backfilling I/O schedulers.

Algorithm 6: Flushing Mechanism

Input : D as the set of data considered for flushing
Output: D_{flush} as the set of data that is going to be flushed

```
1  Function FLUSH(D):
2    foreach d ∈ D do
3      if hasReaders(d) == True then
4        Continue
5      end
6      currentLayer ⟵ getCurrentLayer(d)
7      foreach i ∈ {BottomLayerIndex ··· currentLayer + 1} do
8        d_size ⟵ getSize(d)
9        if C_i >= d_size then
10         add d to D_flush
11         C_i ⟵ C_i − d_size
12         currentLayer ⟵ currentLayer + d_size
13         Continue
14       end
15     end
16   end
17   return D_flush
18 End Function
```

7.1 Infrastructure

We have used the ***MareNostrum CTE-Power*** [25] cluster. This cluster has a heterogeneous storage infrastructure (Table 1). All the experiments were run on 12 worker nodes. Each node is equipped with two local NVMe devices and one local SSD device. All nodes have access to shared Hard Disk Drives (HDDs) mounted with the IBM General Parallel File System (GPFS).

TABLE 1: Storage Infrastructure On The CTE-Power Cluster

	NVRAM	SSD	PFS
Bandwidth	6026 MB/S	2743 MB/S	900 MB/S
Capacity	1 TB	1 TB	8 PB

7.2 Use Cases And Experiments

We implemented two different I/O intensive real applications with PyCOMPSs. Each application exhibits a different I/O workload which allows for evaluating the impact of the storage heterogeneity-aware capabilities in different scenarios. These applications are:

- *Checkpointing HMMER Application*: an application that produces *homogeneous I/O workload*. For a given input size, the checkpointing task writes the same amount of I/O every time it is called during the lifetime of the application.
- *Multi-References Sequence Alignment*: an application that produces *heterogeneous I/O workload*. There are different I/O tasks, each task produces different amount of I/O.

The baseline experiment is the PyCOMPSs implementation that does not use any of the heterogeneity-awareness capabilities. This implementation uses only PFS to write applications data. We compare the baseline version to multiple experiments that used the heterogeneity-aware PyCOMPSs prototype. Each experiment is intended to show how the proposed I/O schedulers behave under certain I/O workloads. These experiments include:

1) FCFS I/O scheduler.
2) Modified priority I/O scheduler.
3) Backfilling I/O scheduler.
4) Using the flushing mechanism with the scheduler that achieved highest performance.

The heterogeneity-aware PyCOMPSs implementation organized the storage infrastructure as described in Section 4.1 into the NVRAM as the top layer, followed by the SSD, then finally the PFS as the bottom layer.

We also added a manual FCFS experiment, which is a traditional PyCOMPSs implementation in which the application code is modified to perform FCFS scheduling of I/O tasks. This is achieved by using the `StorageDevice` argument in the `@constraint` PyCOMPSs decorator to indicate to the runtime that a task should be scheduled to a specific storage layer. This manual implementation is compared to the heterogeneity-aware implementation in terms of code complexity and ease of programming.

It should be noted that all the storage-heterogeneity awareness capabilities that we proposed in this paper are enabled by using command-line arguments at application launch time without having to make any modifications to the code.

7.3 Checkpointing HMMER Application

The *HMMER* Application is used for searching sequence databases given two inputs: a sequence database and a sequence file. Our implementation of the HMMER application (Figure 3) consists of overlapping phases of computation and I/O. In the computation phase, a *hmmpfam* task is called for each sequence and database fragment. The output of each *hmmpfam* task is checkpointed by a *checkpointFrag* I/O task. Finally, all results are grouped by database into a single output file using the *gatherDB* and the *gatherSeq* task respectively.

The experiments of this application used as inputs a 64.5 GB HMM protein-families database (pfam) and a sequence file that contains 140,942,208 sequences with a total size of 50 GB such that each *checkpointFrag* task writes 400 MB of data. Databases and sequence files are publicly accessible from the servers of the European Bioinformatics Institute (EMBL-EBI) servers [26].

Table 2 analyzes the manual and heterogeneity-aware versions of the code in terms of the Halstead metrics that

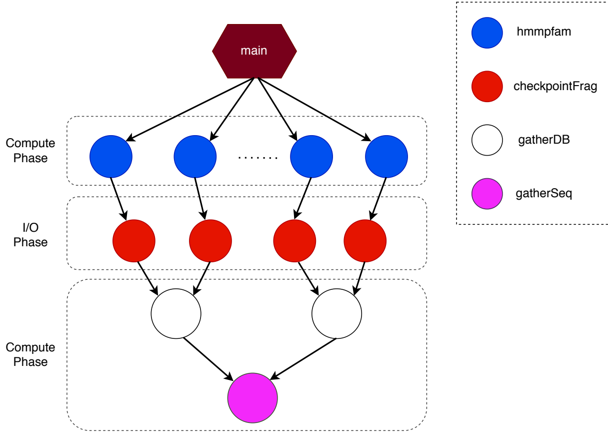

Fig. 3: Task Skeleton Of The HMMER Application

measures the programming complexity, and number of functions declared as I/O tasks. The Halstead metrics is calculated using the Radon tool [27] and includes Lines Of Code (LOC), code difficulty and programming effort. It can be noted that the manual version is significantly complex than the automated heterogeneity-aware version. In the manual version, application developers are responsible for manually setting the target storage layer for all I/O tasks. This is achieved by having as many functions as storage layers such that the constraint of each function/task has a different storage target (i.e., NVRAM, SSD or PFS). Whereas the heterogeneity-aware version is infrastructure-agnostic, i.e., only one I/O function is needed, the storage target of this function will be specified at execution time depending on the used I/O scheduler in a transparent manner to users. Furthermore, in the manual version, users are responsible to perform bookkeeping operations, for instance, the capacity of each storage layers must be monitored by adding the appropriate control logic and if-conditionals. Such details are automatically performed by the PyCOMPSs runtime in the heterogeneity-aware implementation. Consequently, the manual implementation has significantly more lines of code and decision paths. This complexity is reflected as increased code difficulty and programming effort and also decreased portability.

TABLE 2: HMMER Application Code Analysis

Version	LOC	Difficulty	Effort	No. I/O Tasks
Manual	252	4.3	3476.18	3
Heterogeneity-Aware	214	2.5	1786.08	1

The performance results of the application on the CTE-Power cluster is presented in Figure 4. All the experiments that use the heterogeneity-aware capabilities of PyCOMPSs achieve drastic performance improvement over the baseline experiment in both I/O time and total time. As the system is able to take advantage of the heterogeneity of the storage infrastructure, improvements can reach up to almost 3x I/O performance speedup and 44% total time improvement with the FCFS scheduler and flushing experiment. This performance benefit is possible because I/O tasks are first scheduled to storage layers that provide high bandwidth, i.e., NVRAM and SSD. Thus, task parallelism increases because more I/O tasks can run concurrently. Hence, performance improvement is achieved. Once the faster storage layers do not have enough bandwidth or capacity, tasks are scheduled to the PFS. The manual FCFS experiment also achieves a similar performance to the heterogeneity-aware experiments because it uses all the faster storage layer available in the system, however, it has a higher programming complexity as noted in Table 2.

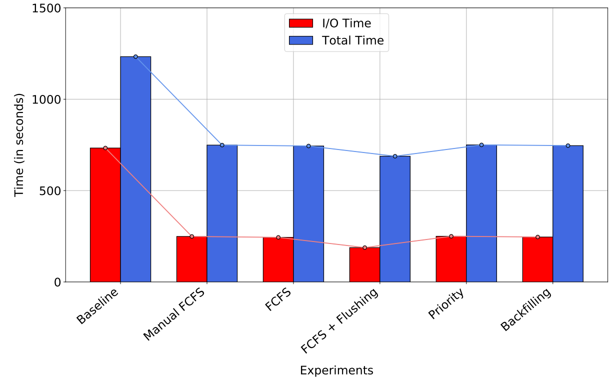

Fig. 4: Performance Results Of The HMMER Application On The CTE-Power Cluster

Taking a closer look at Figure 4, although the priority and backfilling schedulers achieve I/O time and total time improvement over the baseline experiment, they produce similar results compared to the FCFS experiment. As the I/O tasks of this application always write the same amount of data, the benefit of using the heterogeneous I/O schedulers is not apparent.

Furthermore, continuing with Figure 4, it can be noted that enabling the flushing mechanism in the FCFS experiment achieves the best I/O performance and total time. This can be explained because the flushing mechanism continuously frees up capacity in higher storage layers by flushing the final data to bottom layers. Therefore, more I/O tasks can be executed concurrently on the higher storage layers and I/O performance is improved. Due to the compute-I/O pattern of this application, the associated overhead with the flushing mechanism is hidden and does not negatively affect the performance. The flushing process is launched during the compute intensive phases when no I/O tasks are running, hence, avoiding any negative impact on the I/O performance of the application.

7.4 Multi-Reference Sequence Alignment

In the fields of life sciences, short sequences (called reads) are often aligned to multiple reference genomes to identify the similarities between two species. The output of this operation varies in size depending on how much the sequence file matches a certain reference. We developed a PyCOMPSs application (Figure 5) that aligns input sequences to two genomes and writes the alignment results to separate files. It consists of the following tasks: two alignment tasks (*align_ref1*, *align_ref2*); each task aligns the input sequence to a different genome. Each alignment task is then followed by

an I/O task that writes the alignment output to a separate file (*write_res1*, *write_res2*).

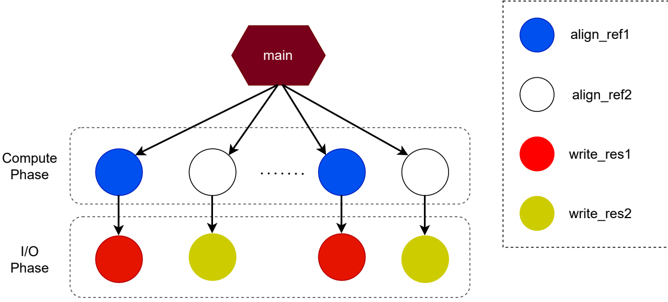

Fig. 5: Task Graph Skeleton Of The Multi-References Sequence Alignment PyCOMPSs Application

We used 2400 sequence inputs in each worker node, each file has a size of 79 MB in compressed gzip format. All input sequences are aligned against two different builds of the human genome reference: HG19 and HG38, each of 38 GB of size. The *write_res1* and *write_res2* produce 200 MB and 800 MB respectively. The total size of data produced at the end of the application is almost 3 Terabytes.

Table 3 presents the code analysis for both the manual and heterogeneity-aware versions. Similar to the previous use case, the manual version is significantly complex than the heterogeneity-aware version. The heterogeneity-aware version is infrastructure-agnostic, and the burden of execution management is removed from the user.

TABLE 3: Multi-Reference Sequence Alignment Code Analysis

Version	LOC	Difficulty	Effort	No. I/O Tasks
Manual	215	4.57	915	6
Heterogeneity-Aware	85	1.5	15.509	2

Figure 6 shows the performance results of the application on the CTE-Power cluster. Similar to the results of the previous use case, the heterogeneity-aware experiments achieve a significant performance improvement that reaches up to 5x I/O performance speedup and up to 48% total time improvement when using the backfilling experiment with flushing.

Unlike the results of the previous use case, it can be noted that the priority and backfilling experiments achieve better I/O performance improvement compared to the FCFS experiments. This can be explained because this application has heterogeneous I/O workloads. The priority and backfilling experiments prioritize the scheduling of critical I/O tasks, i.e, tasks that write more data and require more bandwidth, to higher storage layers. Because higher storage layers offer more bandwidth, they can host the execution of more concurrent critical tasks as compared to bottom storage layers. On the contrary, the FCFS experiments schedule the tasks in order of their arrival to the scheduler. Therefore, less critical I/O tasks fill the bandwidth and consume the capacity of the higher storage layers.

A closer look at Figure 6 shows that the backfilling experiment is achieving better performance than the priority experiment. This can be explained by the behaviour of each scheduler. The priority scheduler stops scheduling when there are no enough resources to schedule the task under consideration. Whereas using the backfilling scheduler, if a task cannot be scheduled because there are no enough resources, subsequent tasks are scheduled if they do not delay launching the waiting task. Consequently, the backfilling scheduler does more work by backfilling the less critical tasks and maximizes resource utilization.

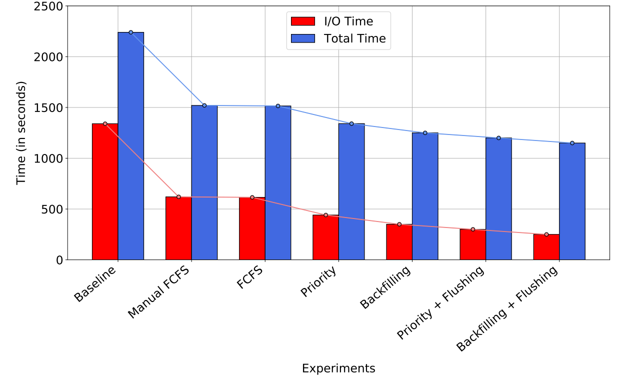

Fig. 6: Performance Results Of The Multi-References Sequence Alignment Application On The CTE-Power Cluster

Furthermore, continuing with Figure 6, it can be noted that using the flushing mechanism in the priority and backfilling experiments achieves better I/O performance and total performance. The flushing mechanism continuously frees up the capacity of the storage layers. Therefore, critical I/O tasks can be scheduled to higher storage layers and task parallelism is increased.

7.4.1 Synthetic Heterogeneous I/O Workload

In order to better understand the difference in performance between the priority and backfilling schedulers, we launched multiple experiments with a synthetic application. This synthetic application mimics the pattern of the Multi-References Sequence Alignment application as illustrated in Figure 5, i.e., interchanging iterations of computations and I/O. The application launches 2400 compute tasks followed by the same number of I/O tasks. The compute tasks generate a certain amount of data, whereas the I/O tasks write the data that has been generated to a separate file. Half of the I/O tasks writes 800 MB of data (the same amount of data written by *write_res2* in use case 7.4), while the other half of I/O task writes 500 MB (bigger size than the data written by task *write_res1* in use case 7.4).

Figure 7 shows the I/O time and total time of the experiments that use the heterogeneous I/O schedulers (i.e., priority and backfilling) on the CTE-Power cluster. Unlike the previous use case (7.4), the backfilling experiment has a worse I/O and total performance than the priority experiment. This happens due to the increase of data sizes written by the less critical task (i.e, *write_res1*). On the one hand, the backfilled I/O tasks in the backfilling experiment consume up the capacities of the higher storage layers. Once the capacities of the higher storage layers become full, more critical tasks (i.e., *write_res1*) are scheduled to lower storage layers. Therefore, task parallelism decreases and

performance degrades. On the other hand, in the priority experiment, the priority scheduler schedules tasks strictly according to their bandwidth, the capacities of higher storage layers are persevered for the execution of more critical tasks.

Continuing with Figure 7, it can be also noted that using the flushing mechanism, the backfilling experiment returns to outperform the priority experiment. As data are periodically flushed and the capacities of higher layers are continuously freed, the backfilling scheduler can maintain a high degree of task parallelism, while doing more work by backfilling less critical tasks.

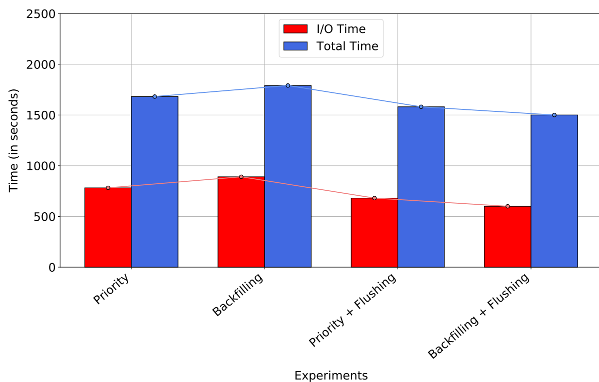

Fig. 7: Performance Results Of The Heterogeneous I/O Schedulers On The CTE-Power Cluster

7.5 Discussion

Even though the experiments presented in this section uses the same storage infrastructure, our proposal is infrastructure-agnostic, i.e., it adapts to the underlying infrastructure and maximizes its utilization. All the proposed mechanism are automatically handled by the runtime system and completely transparent to application developers. Also, we recommend using the mechanism, discussed in [24], to automatically identify the bandwidth requirements of task on a given system.

The presented experiments demonstrated the behaviour of each I/O scheduler with different workloads. It can be concluded that in some scenarios, using the Priority scheduler can make inefficient use of resources, because the scheduling is blocked until there is enough bandwidth to schedule pending I/O task. The Backfilling scheduler can solve this issue, however, if there is a high number of less critical tasks, the overhead of calculating whether to schedule these tasks can increase and negatively impact the performance.

Furthermore, the proposed flushing mechanism proved to optimize I/O intensive applications that exhibit a clear compute-I/O cycle. However, if the application does not follow such a pattern, then overhead should be expected due to the interference that may occur if the flushing has not been completed before the new I/O phase begins.

8 Conclusion

Enabling Storage-heterogeneity awareness in task-based programming models is capable of easing applications development and optimizing I/O performance for applications that follow compute-I/O patterns. In this paper, we proposed abstracting the storage system heterogeneity to unburden application programmers from managing heterogeneous storage devices for performance. In addition to that, we presented three I/O dedicated schedulers to optimize executions of different I/O workloads. Furthermore, we presented a data movement flushing mechanism that enables the reuse of the faster storage devices in the system by periodically freeing their capacity.

Our experiments on the MareNostrum CTE-Power cluster showed that our proposed prototype is able to achieve significant improvements for two real-world applications in terms of programmability and performance. The First Come First Served (FCFS) scheduler is able to optimize I/O performance when the I/O workload is constant throughout applications lifetime. Whereas the Priority and Backfilling schedulers achieve better I/O performance in applications that have variable I/O workloads.

As future work, we plan to enable an adaptive flushing mechanism to accommodate applications that do not have clear compute-I/O cycles. In addition, we plan to implement a pull mechanism to optimize read-intensive applications performance. Also, more hardware details can be specified to optimize the scheduling decisions such as concurrency lanes and wear leveling on flush storage.

Acknowledgments

This work is partially supported by the European Union through the Horizon 2020 research and innovation programme under contracts 721865 (EXPERTISE Project) by the Spanish Government (PID2019-107255GB) and the Generalitat de Catalunya (contract 2014-SGR-1051).

References

[1] *High-Performance Data Analysis: HPC Meets Big Data*. http://www.hpcuserforum.com/presentations/tuscon2013/IDCHPDABigDataHPC.pdf. Date of Last Access: 6th February, 2020.

[2] D. Ibtesham and et al, “On the Viability of Compression for Reducing the Overheads of Checkpoint/Restart-Based Fault Tolerance,” in *41st International Conference on Parallel Processing*, 2012.

[3] P. Braam, “The lustre storage architecture,” 2019.

[4] F. Schmuck and R. Haskin, “GPFS: A Shared-Disk File System for Large Computing Clusters,” in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, USA, 2002.

[5] “Scientific Grand Challenges.” [Online]. Available: https://science.osti.gov/-/media/ascr/pdf/program-documents/docs/Crosscutting_grand_challenges.pdf

[6] B. Xie and et al, “Predicting output performance of a petascale supercomputer,” in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, 2017.

[7] N. Liu and et al, “On the role of burst buffers in leadership-class storage systems,” in *IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, 2012.

[8] A. M. Caulfield and et al, "Understanding the Impact of Emerging Non-Volatile Memories on High-Performance, IO-Intensive Computing," in *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.

[9] A. Gainaru and et al, "Scheduling the I/O of HPC Applications Under Congestion," in *IEEE International Parallel and Distributed Processing Symposium*, 2015.

[10] E. Tejedor and et al, "*PyCOMPSs: Parallel computational workflows in Python*," *International Journal of High Performance Computing Applications*, 2015.

[11] Y. Babuji and et al, "Parsl: Pervasive Parallel Programming in Python," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, 2019.

[12] M. Rocklin, "*Dask: Parallel Computation with Blocked algorithms and Task Scheduling*," in *Proceedings of the 14th Python in Science Conference*, 2015.

[13] *Luigi source code on Github*. Github at https://github.com/spotify/luigi. Date of Last Access: 6th February, 2020.

[14] A. Kougkas and et al, "Hermes: A Heterogeneous-Aware Multi-Tiered Distributed I/O Buffering System," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, 2018.

[15] T. Wang and et al, "UniviStor: Integrated Hierarchical and Distributed Storage for HPC," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018.

[16] D. Henseler and et al, "Architecture and Design of Cray Datawarp." Cray User Group CUG, 2016.

[17] B. Dong and et al, "Data elevator: Low-contention data movement in hierarchical storage system," in *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, 2016.

[18] A. Kougkas and et al, "Harmonia: An interference-aware dynamic i/o scheduler for shared non-volatile burst buffers," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*.

[19] T. Wang and et al, "TRIO: Burst Buffer Based I/O Orchestration," in *2015 IEEE International Conference on Cluster Computing*, 2015.

[20] P. F. Corbett and et al, "MPI-IO: A Parallel File I/O Interface for MPI," 1995.

[21] The HDF Group. (1997) Hierarchical Data Format, version 5. Https://www.hdfgroup.org/HDF5/.

[22] J. Lofstead and et al, "Flexible IO and Integration for Scientific Codes through the Adaptable IO System (ADIOS)," in *Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments*, 2008.

[23] H. Elshazly and et al, "Accelerated Execution Via Eager-release Of Dependencies In Task-based Workflows," *The International Journal of High Performance Computing Applications*.

[24] H. Elshazly, J. Ejarque, F. Lordan, and R. M. Badia, "Towards enabling I/O awareness in task-based programming models," *Future Generation Computer Systems*, 2021.

[25] *MareNostrum CTE-Power Architecture*. Web Page at https://www.bsc.es/support/POWER_CTE-ug.pdf. Date of Last Access: 13th May, 2021.

[26] *EMBL-EBI FTP Server*. Web Page at http:ftp://ftp.ebi.ac.uk/pub/. Date of Last Access: 26th June, 2021.

[27] *Randon Code Metrics.* https://radon.readthedocs.io/en/latest/. Date of Last Access: 28th December, 2021.

Hatem Elshazly is a marie-curie Ph.D. student in the Computer Architecture department at the Technical University of Catalonia (DAC-UPC). Since 2017, he has been working as a research engineer at the Barcelona Supercomputing Center (BSC) optimizing task-based programming models for I/O and memory critical applications. He holds a MSc. degree in the optimization of data intensive applications on distributed infrastructures from Nile University, Egypt (2016). He is currently collaborating in the EXPERTISE multidisciplinary European project under the Horizon 2020 research and innovation programme targeting the optimization of I/O intensive applications. He also collaborated in a multidisciplinary international project with the Harvard Medical School targeting the performance and cost optimization of personalized medicine workflows for clinical use. His current research interests include parallel programming models, high performance computing, mitigating the I/O and memory bottlenecks and the management of heterogeneous distributed infrastructure.

Jorque Ejarque holds a PhD on Computer Science (2015) from the UPC. From 2005 to 2008 he worked as research support engineer at the UPC, and joined BSC at the end of 2008. He has contributed in the design and development of different tools and programming models for complex distributed computing platforms. He has published over 30 research papers in conferences and journals and he has been involved in several National and European R&D projects (FP6, FP7 and H2020). He is member of a program committee of several international conferences, reviewer of journal articles and he was member of the Spanish National Grid Initiative panel. His current research interests are focused on parallel programming models for heterogeneous parallel distributed computing environments and the interoperability between distributed computing platforms.

Rosa M. Badia holds a PhD on Computer Science (1994) from the Technical University of Catalonia (UPC). She is the manager of the Workflows and Distributed Computing research group at the Barcelona Supercomputing Center (BSC). She is also a lecturer at the Technical University of Catalonia. Her current research interest are programming models for complex platforms (from edge, fog, to Clouds and large HPC systems) and its convergence with big data analytics and artificial intelligence. The group led by Dr. Badia focuses its efforts in PyCOMPSs/COMPSs, an instance of the programming model for distributed computing, and its application to the development of large heterogeneous workflows that combine HPC, Big Data and Machine Learning. Dr Badia has published near 200 papers in international conferences and journals in the topics of her research. She has been very active in projects funded by the European Commission, participating in around 20 projects and in contracts with industry (Fujitsu, IBM and Intel). She has been actively contributing to the BDEC international initiative and is a member of HiPEAC Network of Excellence.