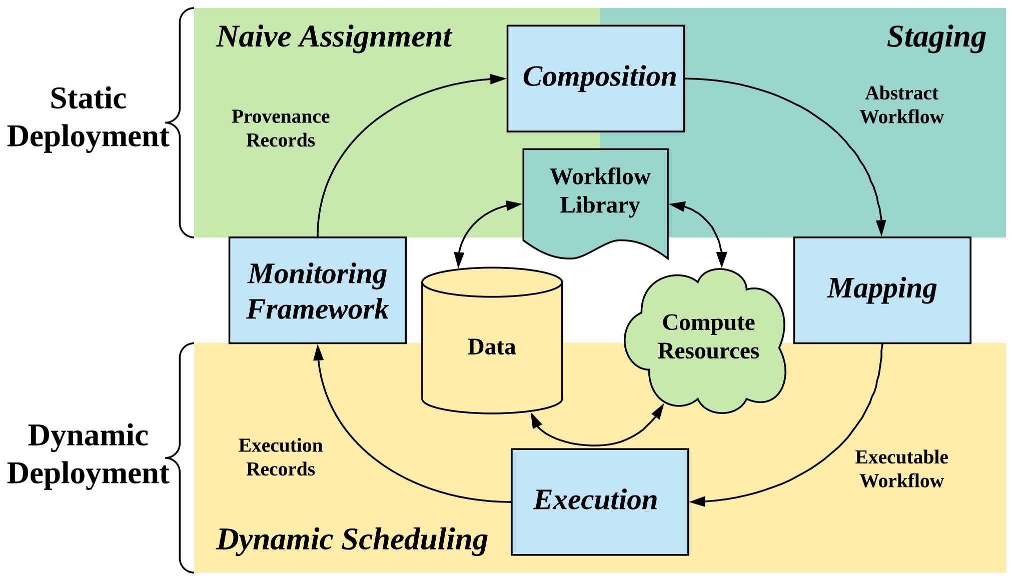
Graphical Abstract

Scalable Adaptive Optimizations for Stream-based Workflows in Multi-HPC-Clusters and Cloud Infrastructures

Liang Liang, Rosa Filgueira, Yan Yan, Thomas Heinis



Lifecycle of dispel4py stream-based workflow with adaptive optimization techniques

Highlights

Scalable Adaptive Optimizations for Stream-based Workflows in Multi-HPC-Clusters and Cloud Infrastructures

Liang Liang, Rosa Filgueira, Yan Yan, Thomas Heinis

- Three adaptive optimization techniques are proposed to improve the scalability of stream-based workflows.
- Two techniques are for the static deployment of workflows, in which resources are pre-assigned to processing elements before starting their executions. Both focus on finding the best resource allocation based on workflows features.
- The third technique enables workflows to adapt to fluctuations in the data-rate and workloads by assigning resources dynamically to processing elements without stopping their executions.
- Evaluations were performed across several platforms and applications to test their effectiveness and adaptivity.

Scalable Adaptive Optimizations for Stream-based Workflows in Multi-HPC-Clusters and Cloud Infrastructures

Liang Liang[a,*], Rosa Filgueira[b,*], Yan Yan[a], Thomas Heinis[a]

[a] *Imperial College London, Exhibition Rd, South Kensington, London, SW7 2BX, England, United Kingdom*
[b] *MACS, Heriot-Watt University , Campus The Avenue, Edinburgh, EH14 4AS,, Scotland, United Kingdom*

Abstract

This work presents three new adaptive optimization techniques to maximize the performance of `dispel4py` workflows. `dispel4py` is a parallel Python-based stream-oriented dataflow framework that acts as a bridge to existing parallel programming frameworks like MPI or Python multiprocessing. When a user runs a dispel4py workflow, the original framework performs a fixed workload distribution among the processes available for the run. This allocation does not take into account the features of the workflows, which can cause scalability issues, especially for data-intensive scientific workflows. Our aim, therefore, is to improve the performance of `dispel4py` workflows by testing different workload strategies that automatically adapt to workflows at runtime. For achieving this objective, we have implemented three new techniques, called `Naive Assignment`, `Staging` and `Dynamic Scheduling`. We have evaluated our proposal with several workflows from different domains and across different computing resources. The results show that our proposed techniques have significantly (up to 10X) improved the performance of the original `dispel4py` framework.

Keywords: Scientific workflow, Stream-based workflow, Workflow optimization, dispel4py
2000 MSC: 68W40, 68W10, 68N01, 68M14, 68M20

1. Introduction

Many scientific fields have become highly data-driven with recent advances in the computational sciences [1]. Areas such as health, seismology, and social computing have come to rely on data-intensive scientific discovery as large volumes of data of various kinds are becoming available. A commonality between all these disciplines is that they generate an enormous complex dataset that requires automated analysis, which has now become a key part of the scientific method, yet remains a highly demanding data- and compute-intensive process.

Scientific communities nowadays have the possibility to access a variety of computing resources and often have computational problems that are best addressed using parallel computing technology. However, successful use of these technologies requires much additional machinery whose use is not straightforward for non-experts. Consequently, various scientific workflow systems [2] designed for bridging the gap between scientific problems and technologies by automatically handling low-level data processing have recently emerged [3].

Among them, stream-based workflow systems have been attracting growing attention from both industry and academia by virtue of their abilities to process unlimited data flows as well as providing lower latency compared to batch-based systems [4]. Therefore, many stream-based workflow systems have been implemented for solving diverse objectives, including `dispel4py` [5]. `dispel4py` is a python library for distributed stream processing which has been well-developed and gained recognition of many scientists from different disciplines varying from seismology to astronomy [6, 7]. It offers mappings to several enactment engines, such as MPI [8], Storm [9], or multiprocessing[1], provides smooth transitions from local development to scalable executions, and reduces the I/O activity by avoiding disk reads and writes.

For constructing `dispel4py` workflows, users have to design, compose and connect different processing elements (PE). PEs represent the basic computational blocks of any `dispel4py` workflow. So users connect PEs as they desire in graphs, also named abstract workflows.

A data-streaming system typically passes small data units along its streams compared with the volume of each data unit (file) passed between stages in task-oriented workflows.

Then, dispel4py automatically maps those abstract workflows to concrete ones, depending on the selected enactment engine. Since abstract workflows are independent of the underlying communication mechanism, these workflows are portable among different computing resources.

However, `dispel4py` performs a very basic and rigid workload allocation by mapping PEs to a collection of processes, as it is shown in Figure 1. Depending on the number of targeted processes (e.g. 10 and 7 processes in Figure 1), which the user specifies when executing a `dispel4py` workflow, multiple instances of each PE are created to make use of all available processes. Note that each PE instance runs in a process.

*Both authors contribute equally to this paper, corresponding author: Rosa Filgueira
Email address: `r.filgueira@hw.ac.uk` (Rosa Filgueira)

[1] `https://docs.python.org/3/library/multiprocessing.html`

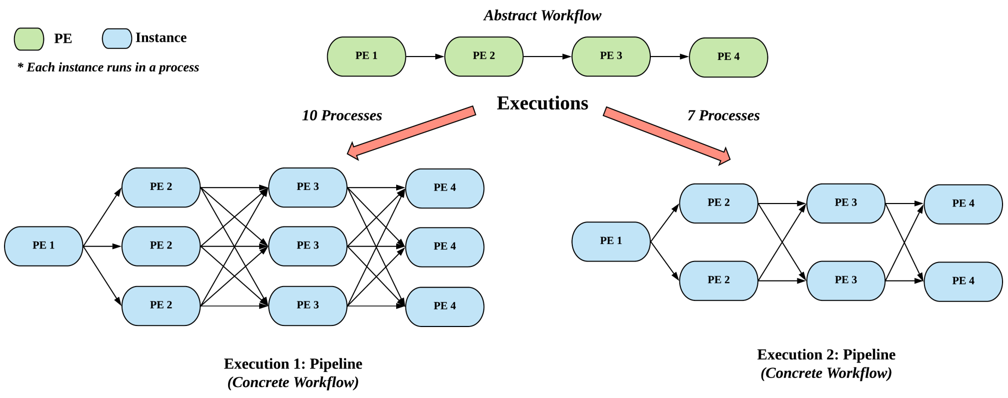


Figure 1: An abstract `dispel4py` workflow and its default workload allocation using 10 (left) and 7 processes (right). `dispel4py` evenly distributes the execution of a concrete workflow on the available processes using a round-robin strategy. Each time a data unit is produced by the PE1 instance, it is sent/streamed to one of the connected PE2 instances, using a round-robin strategy. Identically, for each of PE2 instances, every time a new data unit is produced, it is sent/streamed to one of the connected P3 instances, and so on. If we use more processes for executing a `dispel4py` workflow, the data is more distributed, allowing more calculations in parallel at the same time, and prevents CPU overloading.

The default workload allocation is performed by dividing the number of processes by the number of PEs, with the exception of the first PE, which is always assigned to one process to prevent the generation of duplicate data blocks. When we execute the abstract workflow shown in Figure 1, with 10 processes (*Execution-1*), `dispel4py` assigns three processes (PE instances) to each PE, whereas if we execute it with 7 processes (*Execution-2*) `dispel4py` assigns two processes to each PE, with the exception (for both executions) of the first PE. Notice that in `dispel4py` workflows, the data is being produced and processed in real-time while the workflows are running. Therefore, for our example in Figure 1, only when the workflow is executed, the PE1 instance starts emitting data and distributing to the rest of the topology of the concrete workflow.

This default allocation neither takes into account the data-rate consumed and produced per PE, the execution time per PE, the number of times that a PE is executed, nor the connections between PEs, which could lead to a PE needing to be mapped to more or fewer processes. Furthermore, `dispel4py` adopts the static deployment, which means that once a PE is assigned to a process, we can not do anything about it apart from manually intervening to stop the current execution and re-assign it.

In this work, we have created two adaptive optimization techniques for the static deployment of `dispel4py`: 1) `Naive Assignment`; 2) `Staging`. Both employ different workload allocation strategies taking into account different workflow's features (e.g. data-rate consumed/produced per PE, PEs execution time or PEs connectivity).

Another aspect of our work is to enable dynamic deployment of `dispel4py` workflows, in which PEs are assigned to processes dynamically. We have therefore developed the `Dynamic Scheduling` technique to enable `dispel4py` to allocate resources dynamically while a workflow is running. All techniques have been compared with the default workload allocation and evaluated in two computer infrastructures: An HPC Cluster (Cirrus), HPC at Imperial College London and a Laptop. Two different mappings, *multi* and *MPI* are used for the experiments. We have also conducted an experiment to evaluate the effectiveness of our techniques under unexpected deviation of workload distribution.

The rest of the paper is structured as follows. Section 2 presents the relevant background and formulates the research question. Section 3 presents three workflows: *Seismic Cross-correlation*, *Internal Extinction of Galaxies* and *Window Join*. Section 4 presents the different optimization techniques. Using the previous `dispel4py` workflows as case-studies, we evaluate in Section 5 the optimization techniques on different platforms. Then, we have extended our evaluations in Section 6 to run our static optimizations using a more complex and recent seismological application. We conclude in Section 7 with a summary of achievements and outline some future work.

2. Background

This section explains the main `dispel4py` concepts and introduces the related work about optimization techniques for scientific workflows.

2.1. `dispel4py` concepts

The most important `dispel4py` concepts [5] in the context are as follows:

- *Processing element* (PE) is the computational activity for processing a task or transforming data which can be considered as the node in the workflow graph. PEs are connected by specifying the input and output; the data will be passed among connected PEs as a stream rather than using the file in the task-based workflow system.
- *Instance* refers to the copy of PE that can be executed by the computing process. A PE could be assigned to more than one instance. Instances allows us to scale out a `dispel4py` workflow. One of the challenges that our static techniques try to solve in this work is to calculate

the suitable number of instances per PE (or per partition) in a workflow, taking into account the processes available for workflow execution, as well the workflow's features.

- *Connection* streams data from one output of a PE instance to one or more input ports on other PE instances. The rate of data consumption and production depends on the behavior of the source and destination PEs. Consequently, a connection needs to provide adequate buffering.

- *Abstract Workflow* defines the ways in which PEs are connected and hence the paths taken by data. This is the workflow defined by the user.

- *Concrete Workflow* is the directed acyclic graph that is automatically built by `dispel4py` during enactment, based on the abstract workflow. This is the workflow executed by the compute infrastructures.

- *Partition* can be conducive to optimize the performance of the workflow, which can co-allocate multiple PEs into one process. This forces to run several PEs in a single process. Currently, the user should define partitions manually. Otherwise, each PE is allocated to one partition automatically. Figure 3 shows an example of a partition with three PEs wrapped together. The other challenge addressed by our static techniques is to automatically create the most suitable partitions to minimize the communication time among PEs, taking into account the workflow's features.

- *Grouping* specifies, for an input connection, the communication pattern between PEs. Four different groupings are available: shuffle, group-by, one-to-all, all-to-one. Each grouping arranges that there is a set of receiving PE instances. The shuffle grouping randomly distributes data units to the instances, whereas group-by ensures that each value that occurs in the specified elements of each data unit is received by the same instance. In this case, the effect is that an instance receives all of the data units with its particular value. Finally, one-to-all means that all PE instances send copies of their output data to all the connected instances, and all-to-one means that all data is received by a single instance.

- *Workflow execution time* (makespan, runtime), is the time needed to finish all PE instances in the workflow. In `dispel4py`, the *workflow execution time* varies depending on the number of processes used for running a workflow, obtaining (ideally) a lower execution time when the amount of resources is increased. The ultimate goal of all techniques presented in this work is to reduce the workflow execution time for a given number of resources.

- *Path in the abstract workflow* is a sequence of the dependent PEs, which starts at the head PE (a PE that does not have the predecessors, like PE1 in Figure 1) and finishes at any of the tail PEs (PEs that are not followed by any other tasks). The number of paths increases with each fork in the workflow.

- *Path in the concrete workflow* is a sequence of the dependent PE instances, which start at the head PE instance, and finishes at any of the tail PE instances. The results (output connections) from one PE instance are transferred to one (or more) of the next connected PE instance(s). In order to select which PE instance to send the results, `dispel4py` usually applies the round-robin technique (depending on the connected PE groupings), and therefore creates a circulating path between connected PE instances.

One of `dispel4py's` strengths is the level of abstraction that allows the creation and refinement of workflows without knowledge of the hardware or middle-ware context in which they will be executed. Users can therefore focus on designing their workflows at an abstract level, describing actions, input and output streams, and how they are connected. The `dispel4py` system then maps these descriptions to the selected enactment platforms.

Currently, `dispel4py` supports multiple mappings, such as *simple*, *MPI*, *multiprocessing*, among others. `dispel4py` creates automatically and at runtime different concrete workflows, depending on the mapping selected by users. For example, when users select to execute their workflows with the *simple* mapping, `dispel4py` executes them in sequence within a single process. On the other hand, if users select the *MPI* mapping, `dispel4py` assigns PEs to a collection of MPI processes. And if the selection is the *multiprocessing* mapping (also called *multi*), `dispel4py` creates a pool of processes and assigns each PE instance to its own process.

During enactment and prior to execution, for both *MPI* and *multi* mappings, `dispel4py` performs an equally and fixed allocation of processes to PEs, in which each PE is translated into one or more instances. The number of processes is divided by the number of PEs, obtaining the number of PE instances which will be assigned to each PE. The only exception is the head PE, which will be only assigned to one instance. An example of the default allocation can be seen in Figure 2, in which three PEs have been allocated to seven processes (two PE instances for PE2 to PE4, and one PE instance for PE1) using the *mutli* mapping. The limitation of the default allocation is that number of processors have to be greater (or at least the same) than the number of PEs. Given this fact, the hardware resources have to be always considered when designing workflow. More importantly, the default allocation can not always lead to the best performance in most circumstances, which can be formulated and proven by following equations (from Equation 1 to Equation 6).

$$DAG = G\{V, L\} \tag{1}$$

$$V = \{PE_0, PE_1 ... PE_{N-1}\} \tag{2}$$

$$L = \{(PE_0 \rightarrow PE_1), (PE_1 \rightarrow PE_2) ... (PE_{N-2} \rightarrow PE_{N-1})\} \tag{3}$$

Equation 1, 2 and 3 indicate the abstract workflow represented by Directed Acyclic Graph (*DAG*) with *N* vertexes (*V*) and *N* − 1 links (*L*). To simplify the question, we assume that relationships to PEs in the abstract workflow are one to one.

$$E(V) \propto \frac{k}{P} \text{ , where } k \text{ is constant} \tag{4}$$

$$C(L) \propto P \tag{5}$$

E(*V*) in Equation 4 represents the execution time of processing tasks in PEs, excluding the communication, which is inversely proportional to the number of partitions (*P*). Because, with the decrease of the *P* caused by co-allocating PEs into the same partition, the workload of some processes may increase, thereby increasing the overall execution time of PEs *E*(*V*). By contrast, the communication cost *C*(*L*) of the workflow can be reduced by the decrease of the *P*, because the communication of PEs in the same partition does not result in extra costs (Equation 5).

$$\begin{aligned} E(DAG) &= E(V) + C(L) \\ &\propto \frac{k}{P} + P \text{ , where } k \text{ is constant} \end{aligned} \tag{6}$$

Therefore, the total cost of *DAG* (*E*(*DAG*)) can be represented as the function with indeterminate constant *k* and parameter *P*. Both *k* and *P* are greater than 0 since the execution time should be a positive number, and there must be at least one partition. The function with these conditions has minimal, and we can thus conclude that the default configuration, which simply assigns each PE to each process, cannot always be optimal. Therefore, we propose both static and dynamic techniques for improving the performance of dispel4py, which is how to find the optimal configurations.

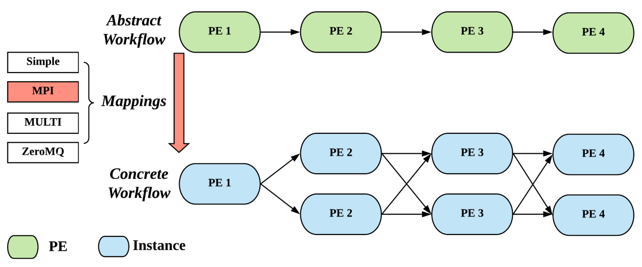


Figure 2: Example of the default workload allocation. In this example, the user has indicated to run the workflow using the *multi* mapping with seven processes. Each PE instance runs in a different process.

2.2. Related Work

Considerable research efforts have been made by previous researchers to improve the performance of scientific workflows [10, 11, 12], nevertheless, the problem persists.

We have classified those into two groups: a) optimization methods and b) scheduling techniques. Among the optimization methods, we can find: heuristic, meta-heuristic, greedy, partitioning, fuzzy and modelling. In comparison, scheduling techniques are usually classified as either static or dynamic deployment.

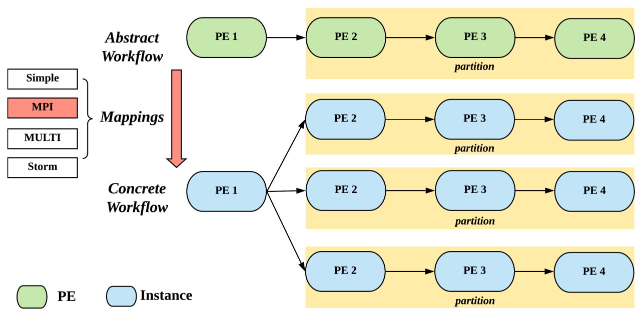


Figure 3: Example of a partition in dispel4py. In this example, the user has defined a partition wrapping together PE2, PE3 and PE4, using 4 processes to run it. Process 0 executes the PE1 instance, while the other processes (Processes 1, 2, 3) execute a copy of the partition (PE2-PE3-PE4).

We have noticed that scientific workflows mostly use heuristics and meta-heuristics as the optimization method to improve their performance [13, 14], being *Dynamic Constraint Algorithm* (DCA) [15] and *Workflow Orchestrator for Distributed Systems* (WORDS) [16] two representative examples. DCA is a user-friendly method for handling issues of bi-criteria of dynamic scheduling. However, DCA may need more number of executions to meet user-defined criteria, such as reliability or the execution requirements. As for WORDS, this approach detects the discrepant features of Cloud computing and provides an effective orchestration to achieve a moderate quality of service over different resources.

Partitioning methods, such as *Multi-Constraint Graph Partitioning* (MCGP) [17], are also very often used to minimise the communication cost of scientific workflows. Partitioning methods are not only used in Scientific workflows, as well in other Big-Data Frameworks, such as *Apache Spark* [18]. *Apache Spark* applies a partitioning method [19] for grouping a set of independent tasks into the same Spark job, where all the tasks have the same shuffle dependencies, reducing the communications across processes.

A number of meta-heuristics suitable for task scheduling on heterogeneous resources have been suggested; a comparative evaluation of twenty different heuristics can be found in [20]. Among these heuristics, the list-based heuristic algorithms, such as Critical Path on a Processor (CPOP), Heterogeneous Earliest Finish Time (HEFT) [21, 22] have become widely used for workflow task scheduling, such as the HEFT modification (GRP-HEFT) presented [23] for minimizing the makespan of a given workflow subject to a budget limit. A list-based heuristic first prioritizes the tasks according to a metric such as *Upward* and *Downward Ranking*, then assigns the tasks to the fastest processors, having the advantage of simplicity and producing generally good schedules with a short makespan. Genetic Algorithms (GA) have also been widely reckoned as useful meta-heuristics for obtaining high-quality solutions to a broad range of combinatorial optimization problems, including the workflow task scheduling problem [24].

Many scheduling algorithms have also been proposed to optimize throughput or resource utilization for different data streaming frameworks, such as [25, 26] for Apache Storm [27], for Apache Spark or for Apache Flink [28]. However, all these scheduling techniques have been implemented for a particular data-streaming framework in mind.

While conducting the literature review, we also noticed that most of the scientific workflows apply a static deployment [13] since this technique is usually lightweight and easy to implement. However, to re-balance the allocation performed by a *static* scheduling technique, we need to stop the current execution and re-assign the workload either manually or by applying an assignment algorithm based on previous executions.

On the contrary, dynamic deployment can re-balance the workload of scientific workflow on-the-fly, meaning that if a task needs more or fewer resources, it dynamically scale up or down without stopping the workflow execution.

In this work, we have used two optimization techniques (`Naive Assignment` and `Staging`) to improve the current static deployment in `dispel4py` by taking into account the communication patterns among PEs trying to place PEs in the same partitions and assigning the most suitable number of processes to each partition, applying in each technique a different set of heuristics. While the `Naive Assignment` uses a heuristic that shares some similarities with HEFT, the `Staging` uses a heuristic inspired by *Apache Spark* Stage method [29]. Our static optimization techniques do not aim to schedule the PEs' execution order, like previous work, because the order is fixed by the `dispel4py` static deployment. Instead, their aim is to reduce the workflow execution time and to maximize the throughput by finding find the best configuration of a) partitions (to reduce the cost of moving data through the network links); and b) the number of processes assigned to those partitions.

Furthermore, we have also developed a new scheduling technique, `Dynamic Scheduling`, to enable dynamic deployment in `dispel4py`.

Finally, we would like to highlight that since `dispel4py` offers mappings to several engines (e.g. Apache Storm, MPI, etc.), our proposed techniques are completely independent of the framework used underneath to run `dispel4py` workflows, allowing us to apply our techniques to several enactment engines, and not to specific ones.

3. Use Cases

The following subsections describe the `dispel4py` workflows used to evaluate our optimization techniques.

3.1. Seismic Cross-Correlation

The workflow has been designed to monitor and analyze the geological waveform data from several seismic stations [2]. Its main goal is to assess and forecast the risk and probability of volcanic eruptions and earthquakes in real-time [6].

[2]`https://www.fdsn.org`

Figure 4 illustrates all components of the workflow, which can be classified into two phases, prepossessing the data collected from stations and calculating the cross-correlation. Each phase has been implemented as a `dispel4py` workflow. During *Phase One*, each continuous time series from a given seismic station (called a trace) is subject to a series of treatments (all of them included in the `Prep` composite PE). The processing of each trace is independent of any other, making this phase embarrassingly parallel. *Phase Two* pairs all stations and calculates the cross-correlation for each pair.

In this work, we have selected the *Phase One* of this application and decomposed the `Prep` Composite PE into several PEs.

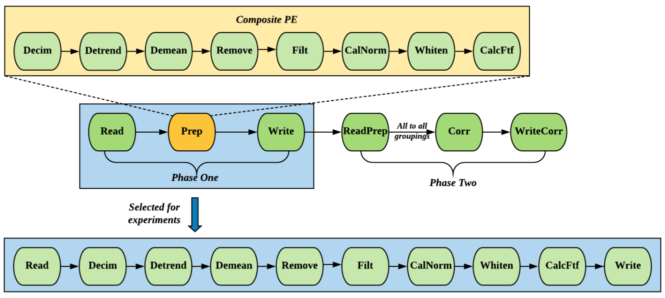


Figure 4: A simplified abstract workflow for seismic cross-correlation.

3.2. Internal Extinction of Galaxies

This workflow has been implemented to calculate the extinction within the galaxies, which is a significant property in astrophysics [30]. This property reflects the dust extinction of the internal galaxies and is used for measuring the optical luminosity[3]. This workflow is reusable since it can be regarded as a prior step for other complex tasks which require this property.

As we can see in Figure 5, this workflow has four PEs. `Read` PE loads the input file which stores the coordinates data of interest. Then, `Votab` downloads the corresponding VOTable [4] from Virtual Observatory website [5] based on those coordinates. Afterwards, `Filt` PE parses the VOTable by using astropy library[6]and filters the parsed data by selecting needed columns. Finally, `Intext` PE calculates the internal extinction based on data from `Filt` PE.

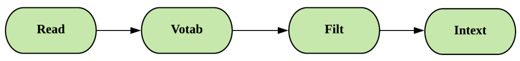


Figure 5: Workflow for calculating the internal extinction of galaxies.

[3]`http://amiga.iaa.es/p/1-homepage.html`
[4]`http://www.ivoa.net/documents/PR/VOTable/VOTable-20040322.html#ToC9`
[5]`http://ivoa.net`
[6]Python library designed for use in astronomy: `https://www.astropy.org/`

3.3. *Synthetic Workflow - Window Join*

We have developed this new synthetic workflow[7], which has a more complex topology than both previous workflows. The *Window Join* workflow aims to simulate a fundamental query operation, window join, in streaming processing, which produces the result from unbounded streams by using concepts of the window to limit the scope of data for join [31].

As we can see in Figure 6, this workflow consists of six PEs connected via a fork-join manner. `Read` PE loads the data from Customer and Supplier TPC-H Tables[8]. Then, data is sent to `FilterCus` and `FilterSup` PEs. `FilterCus` selects the data from Customer table, whereas `FilterSup` retains the data from Supplier table. `CleanCus` and `CleanSup` clean their corresponding data received from `FilterCus` and `FilterSup` respectively. Finally, the data are joined by `Join`, which exploits tuple-slide window to restrict the range of the unbounded data to perform the join.

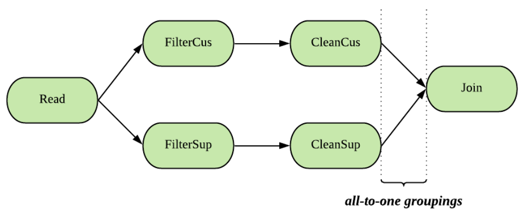


Figure 6: Workflow representing the *Window Join* synthetic application.

This workflow uses the `all-to-one` grouping for joining the data from previous PEs. This means, that all instances of `CleanCus` and `CleanSup` send their data to one instance `Join` PE. Although `Join` PE can be assigned to more than one process, data are only sent to one instance.

4. Optimizations

This section presents three new adaptive optimization techniques developed to improve the performance of `dispel4py` workflows. Although we have developed them for `dispel4py`, they can easily be applied to other stream-based workflow systems.

Naive Assignment and Staging are in the range of static deployment, while Dynamic scheduling can definitely be regarded as a dynamic deployment. Figure 7 shows which part of the life-cycle of dispel4py on which those optimization methods based and focus.

4.1. *Naive Assignment Technique*

This technique aims to reduce the overall workflow execution time by identifying *hot connections* among PEs and grouping them together into the same partition. It works *offline* by analysing

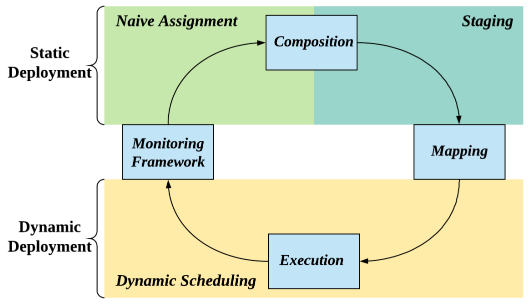


Figure 7: Overview of the adaptive optimization techniques proposed for `dispel4py`.

the previously recorded data and adapting the workload allocation to it. Notice that by default, when partitions are not indicated by the user, each PE runs in a single partition.

This technique relies on the `dispel4py` monitoring framework, which collects (while a workflow is running): a) execution and communication times per PE; b) number of PEs; c) the number of iterations; d) data size, and e) mapping used. The `Naive Assignment Technique` analyses *offline* these data to discover which is the best workload allocation (partitions and number of processes assigned to each partition) for that specific workflow under the same circumstances.

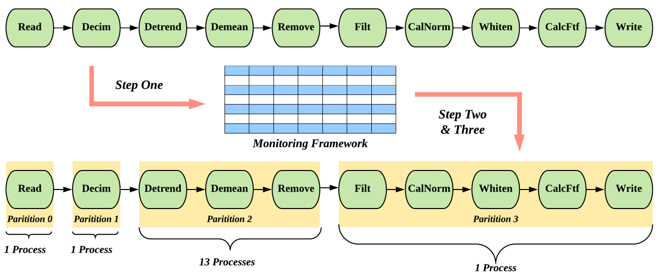


Figure 8: Application of the `Naive assignment` technique over the *Seismic Cross-correlation* workflow using 16 processes.

To calculate the most suitable partitions, the technique groups together all connected PEs that have communication times higher than their execution time (*hot connection*), with the exception of the first PE, which is always assigned to a single partition. This criterion has been implemented using the Algorithm 1. In this algorithm, there is an important prerequisite to combine two PEs into the same partition: the connected PEs should be one-to-one relationship; otherwise, the partition could result in incorrectness since PEs without a one-to-one relationship cannot receive or send data from or to PEs in other partitions.

Figure 8 shows an example of the `Naive Assignment` technique over the *Seismic Cross-correlation* workflow introduced in Section 3.1.

The next step is to calculate the number of processes assigned to each partition. As the default `dispel4py` allocation, the first partition (for the first PE), only one process is assigned to it. For calculating the remaining processes, we have devel-

[7]`https://git.ecdf.ed.ac.uk/msc-19-20/s1980912/tree/master/workflows/join`
[8]`http://www.tpc.org/tpch`

Algorithm 1: Assigning Partition

```
1: Require: Workflow consisting of $N$ $PEs$ ($PE_0$, $PE_1$ ... $PE_{N-1}$), Execution time of each $PE$ ($E(PE_i)$), Communication time between adjacent $PEs$ ($C(PE_i, PE_{i+1})$)
2: for $i = 0$ to $i$ = N-2 do
3:     if $i = 0$ then
4:         $PE_i$ is assigned to single partition
5:     else
6:         if $C(PE_i, PE_{i+1}) >$ MIN($E(PE_i)$, $E(PE_{i+1})$) and $PE_i$ and $PE_{i+1}$ is one-to-one relationship then
7:             $PE_i$ and $PE_{i+1}$ are assigned to the same partition or $PE_{i+1}$ is added into the existing partition which $PE_i$ is in
8:         end if
9:     end if
10: end for
```

oped Algorithm 2. This algorithm calculates the execution time of each partition (adding the execution time of all PEs included in each partition) and divides it by the total execution time of all partitions (except the first partition). This result is multiplied by the number of processes available (minus one, which is assigned to the first PE), then obtaining the suitable number of processes per partition.

Algorithm 2: Assigning Process

```
1: Require: Workflow consisting of $M$ $PARTs$ ($PART_0$, $PART_1$ ... $PART_{M-1}$) or including $N$ $PEs$ ($PE_0$, $PE_1$ ... $PE_{N-1}$), Total number of processes ($TotalNumProcess$), Execution time of each $PE$ ($E(PE_i)$)
2: Define: Execution time of each partition as $E(PART_i)$, Number of processes for each partition ($NumProcess(PART_i)$), Total execution time ($E(TOTAL)$)
3: for $i = 1$ to $i$ = N-1 do
4:     $E(TOTAL) = E(TOTAL) + E(PE_i)$
5: end for
6: for $i = 0$ to $i$ = M-1 do
7:     if $i = 0$ then
8:         $NumProcess(PART_i) = 1$
9:     else
10:        for $PE$ in $PART_i$ do
11:            $E(PART_i) = E(PART_i) + E(PE)$
12:        end for
13:        $NumProcess(PART_i) = (TotalNumProcess - 1) \times \frac{E(PART_i)}{E(TOTAL)}$
14:    end if
15: end for
```

The `Naive Assignment` shares some similarities with *Heterogeneous Earliest Finish Time* (or HEFT) heuristic, since both use the execution and communication times for assigning tasks (or PEs) to processes and therefore reduce the workflow execution time. HEFT takes a set of tasks, represented as a DAG, a set of processes, the times to execute each task on each process, and the times to communicate the results from each job to each of its children between each pair of processes. While HEFT assigns a task to processes after prioritizing them based on the average execution and communication times (using Earliest Finish Time (EFT)), the `Naive Assignment` uses the average execution and communication times to determine *hot connections* and therefore to calculate the best workload allocation for running the same workflow under the same conditions.

4.2. Staging Technique

Our next technique has been inspired by *Apache Spark Stage* method[9], which aggregates into the same stage all operations which do not require shuffling data [10]

Our `Staging` technique aims to reduce the overall workflow execution time by grouping PEs that do not require shuffling data. Unlike the `Naive Assignment`, `Staging` creates in *runtime* the number of partitions by analysing the dependencies between PEs specified in the workflow.

In order to allocate a PE into the previous partition, the following conditions need to be met: a) one-to-one relations between PEs; b) there is no grouping in the destination PE. c) the first PE is always assigned to its own partition.

To calculate the number of processes allocated to each partition, we applied the default method of `dispel4py`. The number of partitions is equally distributed among the processes, with the exception of the first partition, which will be allocated to just one process.

`Staging` first generates the data structure for representing the source and destination of PE (see Algorithm 3). Since the workflow is a one-way graph, Algorithm 3 finds the corresponding source PE and destination PE according to each edge of the graph; then stores all of the PEs that can be regarded as source into a dictionary, and also stores all the PEs that can be regarded as destination into another dictionary. These two dictionaries have the same structure, where the key is PE, and the corresponding value is a list composed of the direction tuple of the edge related to this PE. Moreover, for the dictionary of the source PE, the direction tuple only records the PE as the source of the edge. Similarly, for the dictionary of the destination PE, the direction tuple only records this PE as the destination of the edge.

After establishing these two dictionaries, `Staging` performs the specific partition steps (see Algorithm 4). Since the root PE of each graph can only be assigned to a single partition, Algorithm 4 finds the root PE of the graph and places it in a separate partition. Then, this algorithm traverses each key in the source PE dictionary. If the PE has not been classified into any partition, then it would find out whether this PE has a one-to-one relationship with connected PEs, and in this one-to-one relationship, the destination PE does not contain grouping parameters. If there is such a relationship, then the algorithm places the PEs involved in the relationship into one partition; however, if there is no such relationship, it places the PEs into different partitions.

There is an example of this technique applied to the *Window Join* workflow (introduced in Section 3.3) in Figure 9.

[9]`https://spark.apache.org/docs/1.2.1/api/java/org/apache/spark/scheduler/Stage.html`

[10]The process of moving the data from partition to partition in order to aggregate, join, match up, or spread out in some other way, is known as shuffling. The aggregation/reduction that takes place before data is moved across partitions is known as a map-side shuffle.

Algorithm 3: *Create_Source_dict_and_Dest_dict*

```
1:  FUNCTION: $Create\_Source\_dict\_and\_Dest\_dict$
2:  Require:   Workflow graph ($G$) consisting of $N$ $PEs$ ($PE_0$, $PE_1$ ... $PE_{N-1}$)
3:  Define:   Dictionary of source PE ($Source\_dict$), Dictionary of destination PE dictionary ($Dest\_dict$), All edges of PE ($Edges$), Source PE of edge ($Source\_PE$), Destination PE of edge ($Dest\_PE$)
4:  Return:   $Source\_dict$ and $Dest\_dict$
5:  for $PE$ in $G$ do
6:      $Edges = G$.edges($PE$)
7:      for $Edge$ in $Edges$ do
8:          get $Source\_PE$ and $Dest\_PE$ based on two direction of the $Edge$
9:          if $Source\_PE == PE$ then
10:             Update the $Source\_dict$ with key as $Source\_PE$ value as all destination PEs in form of list of tuple(s) ($Source\_PE$ and $Dest\_PE$)
11:         end if
12:         if $Dest\_PE == PE$ then
13:             Update the $Dest\_dict$ with key as $Dest\_PE$ value as all source PEs in form of list of tuple(s) ($Source\_PE$ and $Dest\_PE$)
14:         end if
15:     end for
16: end for
17: EndFunction
```

Algorithm 4: Staging Algorithm

```
1:  Require:   Workflow graph ($G$) consisting of $N$ $PEs$ ($PE_0$, $PE_1$ ... $PE_{N-1}$)
2:  Define:   Dictionary of source PE ($Source\_dict$), Dictionary of destination PE dictionary ($Dest\_dict$)
3:  $Source\_dict$, $Dest\_dict$ = $Create\_Source\_dict\_and\_Dest\_dict(G)$
4:  for $PE$ in $Source\_dict$.keys() do
5:      if $PE$ not in $Dest\_dict$.keys() then
6:          put $PE$ into a single partition
7:      else
8:          if $PE$ is not assigned to any partition and $PE_i$ and $PE_{i+1}$ is one-to-one relationship then
9:              get partition by calling $Find\_one\_to\_one()$
10:             if this partition is empty then
11:                 put $PE$ into a single partition
12:             end if
13:         end if
14:     end if
15: end for
16: for $PE$ in $dest\_dict$.keys() do
17:     if $PE$ is not assigned to any partition then
18:         put $PE$ into a single partition
19:     end if
20: end for
```

4.3. Dynamic Scheduling Technique

Workload skewness [32] and variance are common phenomena in distributed stream processing engines. When massive stream data flood into a distributed system for processing and analysis, even slight changes in the distribution of the incoming data

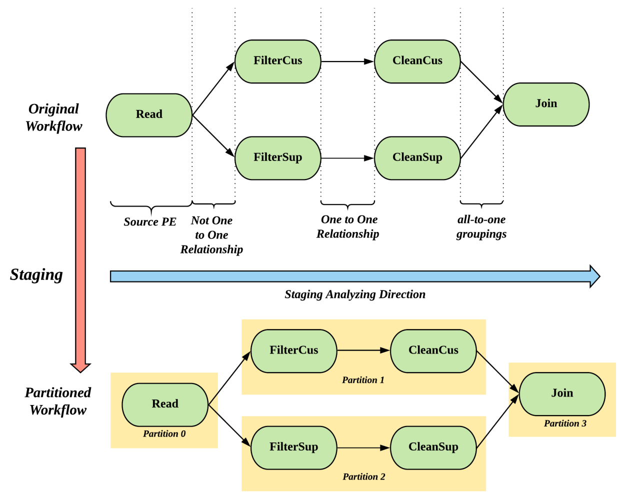


Figure 9: Example of the `Staging` technique applied to *Window-Join* workflow.

stream may significantly affect the system performance. Our previous static techniques do not take into account changes in the data stream but instead only the communication patterns to find the suitable number of partitions, potentially generating suboptimal performance when the evolving workload deviates from the expectation.

In order to tackle this problem, we propose the `Dynamic Scheduling` technique. In this case, processes are not locked to specific PEs, scheduling PE instances on-the-fly, meaning that if a PE needs more or less "resources", this technique dynamically scale up or down, re-balancing automatically graph without stopping the workflow execution. This technique allows users to design workflows without any restriction of resources such as the maximum number of threads supported by a single node, and it can deal with workload skewness.

The implementation of this technique is based on the Python multiprocessing[11] package, meaning that we can only deploy workflows dynamically on shared-memory architectures, using the `dispel4py` *multi* mapping.

When a `dispel4py` workflow is executed using the dynamic deployment, all processes receive a copy of the abstract workflow (so all are aware of the PEs dependencies) at the beginning of the workflow execution. `Dynamic scheduling` uses a global queue to store PEs and data coming up, which is available to all processes. The main idea is that each process, as soon as it is "free", goes to the global queue to *pull* the next PE to execute along with the necessary data. After finishing the execution of the PE, it returns to the global queue to *push* the output data.

This technique is currently not compatible with a workflow with groupings given that they are stateful operations. If a user uses a grouping over a PE, all data received by this particular PE has to go to a particular PE instance always running in the same process to preserve its state. The current implementation

[11] `https://docs.python.org/3/library/multiprocessing.html`

of the `Dynamic Scheduling` technique can not guarantee such behaviour.

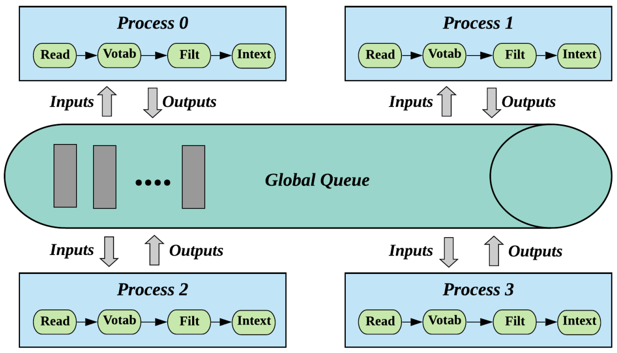
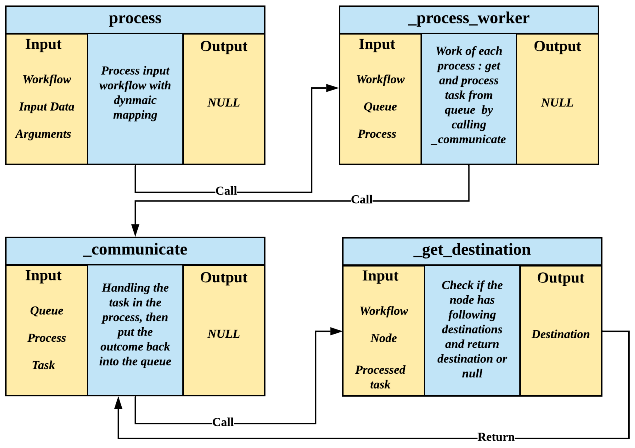


Figure 10: Example of the `Dynamic Scheduling` technique applied to the *Internal Extinction of Galaxies* workflow.



process
Input
Workflow
Input Data
Arguments
Process input workflow with dynmaic mapping
Output
NULL
_process_worker
Input
Workflow
Queue
Process
Work of each process : get and process task from queue by calling _communicate
Output
NULL
Call
Call
_communicate
Input
Queue
Process
Task
Handling the task in the process, then put the outcome back into the queue
Output
NULL
_get_destination
Input
Workflow
Node
Processed task
Check if the node has following destinations and return destination or null
Output
Destination
Call
Return

Figure 11: This diagram shows the main blocks of `Dynamic Scheduling`, and their relationships. This information is complementary to the pseudo-code shown in Algorithm 5.

An overview of the `Dynamic Scheduling` technique applied to the *Internal Extinction of Galaxies* workflow (introduced in Section 3.2) can be seen in Figure 10.

To explain in more detail how `Dynamic Scheduling` works, we show the main components of this technique (and their relationship) in Figure 11. Each of them can be described as follows:

- `process`: this function is the entry point for the `Dynamic Scheduling` technique. It is responsible for creating the global queue (see Figure 10), distributing copies of an abstract workflow to processes and also orchestrating the execution of a workflow with available processes.

- `_process_worker`: this function allows processes to fetch 'work' from the global queue. More specifically, processes pull PEs (one at a time) from the queue, execute their work and return the results (and destination PEs) back to the queue.

- `_communicate`: this function handles all the communications from and to the global queue, pulling PEs (and their data) and pushing back results and destination PEs.

- `_get_destination`: this function is responsible for identifying PE's destinations. Taking the example shown in Figure 10, when a process executes *Votab* PE, this function identifies that *Filt* is its destination PE, and returns this information to `_communicate` function. Subsequently, `_communicate` pushes back to the queue the results of *Votab* PE and *Filt* PE.

Algorithm 5 shows the pseudo-code of the previous components and all the steps previously described.

5. Experimental Evaluation

This section presents the experimental evaluations for the proposed techniques in Section 4. For each experiment, we have run one of the workflows introduced previously using the *multi* and *MPI* mapping , modifying the number of processes, and selecting one of the following allocations techniques:

- *Default*: We apply the default `dispel4py` workload allocation. Each PE runs in a single partition, and the number of processes allocated to each PE (or partition) is calculated by dividing the number of processes between the number of PEs (or partitions).

- *Naive 1*: We apply just the first algorithm (Algorithm 1) of `Naive Assignment` to calculate the most suitable number of partitions by identifying *hot connections*. The number of processes assigned to each partition is calculated using the default method: dividing the number of processes between the number of partitions previously calculated by Algorithm 1. This technique implies running previously the same workflow under the same conditions collecting the necessary information using the monitoring framework of `dispel4py`.

- *Naive 2*: We apply the full `Naive Assignment` technique, which includes Algorithm 1 to calculate the number of partitions (by identifying *hot connections*), and Algorithm 2 to calculate the number of processes assigned to each partition. This method also implies to run the same workflow previously using the monitoring framework of `dispel4py`.

- *Stage*: We apply `Staging` to calculate the number of partitions by grouping PEs that do not require shuffling data. The number of processes assigned to each partition is calculated by applying the default method: dividing the number of processes between the number of partitions previously calculated by the `Staging` technique.

- *Dynamic*: We apply `Dynamic scheduling`, enabling the dynamic deployment of workflows. Since this technique

Algorithm 5: Dynamic Scheduling

```
1: FUNCTION: _get_destination()
2: Require:   Workflow graph ($G$), Process ($proc$)
3: Define:   Destinations ($dest\_list$)
4: Return:   Destination ($dest$)
5: Traverse $G$ to find destinations ($dest\_list$) of $proc$
6: if $dest\_list$ is empty then
7:     Return NULL
8: else
9:     for $dest$ in $dest\_list$ do
10:        Return $dest$
11:    end for
12: end if
13: EndFunction

14: FUNCTION: _communicate()
15: Require:   Workflow graph ($G$), Queue ($Q$), Process ($proc$),
    Task consists of $PE$ and $data$
16: Define:   Processed task ($output$)
17: $output$ = PE.process(data)
18: if the PE has connections with other PEs by calling
    _get_destination then
19:     put the output $output$ back into the $Q$
20: end if
21: EndFunction

22: FUNCTION: _process_worker()
23: Require: Workflow graph ($G$), Queue ($Q$), Process ($proc$), Task
    ($task$)
24: while TRUE do
25:     if $Q$ is not empty then
26:         get $task$ from $Q$
27:         call _communicate()
28:     else
29:         break
30:     end if
31: end while
32: EndFunction

33: FUNCTION: process()
34: Require: Workflow graph ($G$), Input data ($input$), Number of
    processes ($size$)
35: Define: Queue ($Q$), Process ($proc$), processed input
    ($processed\_input$)
36: for $proc$ = 0 to $proc$ = $size$ do
37:     $proc$ get a copy of workflow
38: end for
39: for $PE$ in $G$ do
40:     get $processed\_input$ by calling dispel4py built-in function
41:     if $processed\_input$ is not NONE then
42:         put $processed\_input$ into the $Q$
43:     end if
44: end for
45: start the Multiprocessing to execute _process_worker() for each
    process with its workflow
46: EndFunction
```

currently is not compatible with groupings (stateful operations), we have not used it for the *Window Join* workflow.

Note that *Default*, *Naive 1* and *Naive 2* techniques require to use a greater or equal number of processes as PEs.

5.1. Evaluation Platforms: Computing Infrastructures features

We have selected the following computing infrastructure:

- Cirrus: it is a state-of-the-art SGI ICE XA system with 280 compute nodes with Lustre as the file system and CentOS Linux as the OS[12]. Cirrus standard computes nodes contain two 2.1 GHz, 18-core Intel Xeon E5-2695 (Broadwell) series processors. Each core supports two hardware threads (Hyperthreads), which are enabled by default. The standard computes node on Cirrus has 256 GB of memory shared between both processors. This means that we can use up to 72 processes for our experiments in Cirrus.

- HPC supported by Imperial College London: it is HPC cluster with Intel E5-2680 v3 @ 2.50GHz running Centos 8. It provides multiple job classes for work with different workloads, which allows us to employ up to 72 computing nodes. Each node consists of 48 cores.

- Google Cloud Platform[13]: Offered by Google, it is a suite of cloud computing services (such as computers and hard disk drives, or virtual machines (VMs)) that run on the same infrastructure that Google uses internally for its end-user products. For this work, we have configured a Debian VM with 32 vCPUs and 128 GB memory.

- Laptop (Local): it uses macOS Catalina as the OS and has a 2.3 GHz hyper-threading 8-Core i9 processor with 16 GB memory. Since each core supports two hardware (Hyperthreads), we have used up to 16 processes for our experiments.

5.2. Analysis Based on the Seismic Cross-correlation Workflow

Figure 12 shows the execution times (runtime) of the different experiments conducted using the *Seismic Cross-correlation* workflow in Cirrus (Figure 12.A and Figure 12.B), and in the laptop (Figure 12.C and Figure 12.D). In this experiment we have varied the number of cores from 4 to 64 in Cirrus, and 4 to 16 in the Laptop. However, since the workflow has 10 PEs, all experiments conducted with *Default*, *Naive 1* and *Naive 2* techniques start with 16 processes[14].

First, we evaluated all static deployment techniques (*Default*, *Naive 1*, *Naive 2* and *Stage*) in both computing infrastructures. Subsequently, we selected the best static technique(s) (which is the one that has lower execution time across different

[12]`https://www.cirrus.ac.uk/about/`
[13]`https://cloud.google.com/`
[14]The minimum number of processes required by this workflow is 10.

number of processes) and compared it/them with the *Dynamic* technique.

For Cirrus, the best static technique is *Stage*. Whereas for the laptop *Naive 2* and *Stage* have found the same allocation, being this one the best one according to the results.

The *Dynamic* technique has a very similar performance to the selected static techniques for both infrastructures.

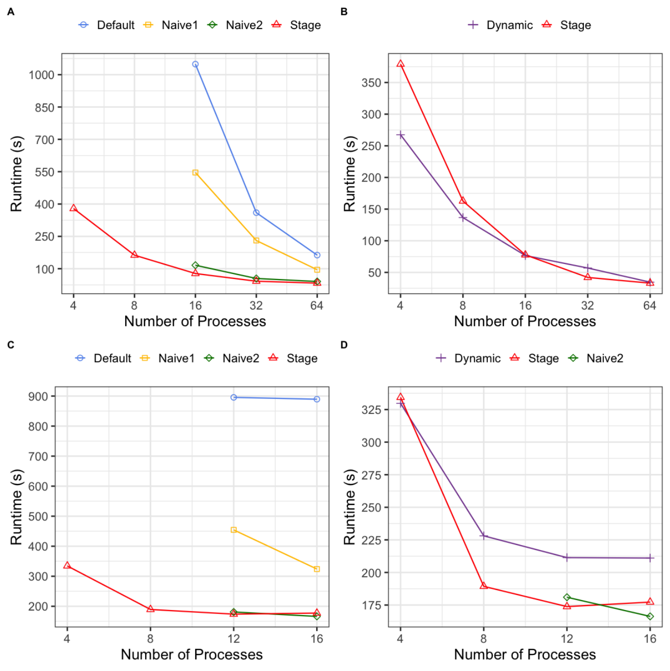


Figure 12: Evaluations of *Seismic Cross-correlation* with *multi* mapping. A: Cirrus execution times employing all static techniques; B: Cirrus execution times employing the dynamic technique and best static deployment(s). C: Laptop execution times employing all static techniques; D: Laptop execution times using the dynamic technique and best static deployment(s).

Figure 13 shows the static deployment with *MPI* mapping running on laptop and HPC resources respectively. In this experiments, we used up to 256 processes in HPC at Imperial (Figure 13 (A)) and up to 28 processes in laptop (Figure 13 (B)). Additionally, for the testing on HPC, we used 30X larger dataset to test if the performance of different optimization methods is affected by large input data size. The result shows that overall trends for *MPI* mapping is similar to *multi* mapping. The performances of *Naive 2* and *Stage* are very close, outperforming others for different platforms.

5.3. Analysis Based on Internal Extinction of Galaxies Workflow

Figure 14 shows the execution times (runtime) of the different experiments conducted using the *Internal Extinction of Galaxies* workflow in Cirrus (Figure 14.A and Figure 14.B), and in the laptop (Figure 14.C and Figure 14.D). In this experiment, we have varied the number of cores from 4 to 64 in Cirrus and 4 to 16 in the Laptop. Note that this workflow has four PEs, so all experiments across techniques start with four processes.

Once again, we evaluated all static techniques in both platforms. For this workflow, *Stage*, *Naive 1* and *Naive 2* agreed

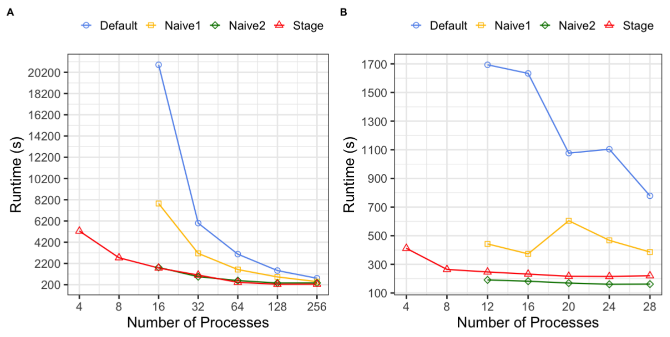


Figure 13: Evaluations of *Seismic Cross-correlation* with *MPI* mapping employing static deployments. A: HPC execution times with larger input dataset; B: laptop execution times.

in the most suitable allocation of resources (number of partitions and number of processes assigned to each of them). Figure 14.A and 14.C show that this allocation performs better than the *default* allocation across platforms and number of processes.

The *Dynamic* technique, however does not perform better than the static techniques (*Stage*, *Naive 1* and *Naive 2*), for both platforms. Therefore, static optimization methods significantly outperform the *Dynamic* technique.

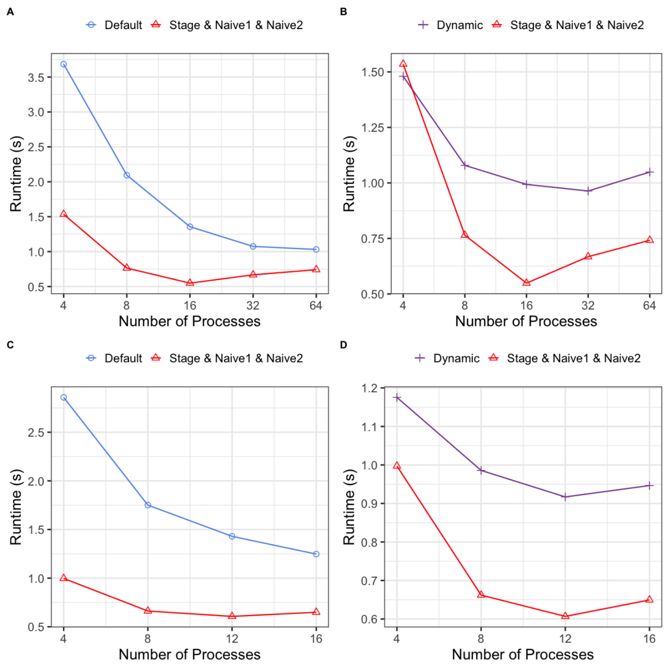


Figure 14: Evaluations of *Internal Extinction of Galaxies* with *multi* mapping. A: Cirrus execution times employing all static techniques; B: Cirrus execution times employing the dynamic technique and best static deployment(s). C: Laptop execution times employing all static techniques; D: Laptop execution times using the dynamic technique and best static deployment(s).

Figure 15 shows the overall trends of the performances of *Default* and *Stage* & *Naive 1* & *Naive 2* with *MPI* mapping for HPC and laptop. The number of processes used for the experiment is from 4 to 64 in HPC, and 4 to 20 in laptop. And the data size of HPC is 1000X of laptop. From those figures, we

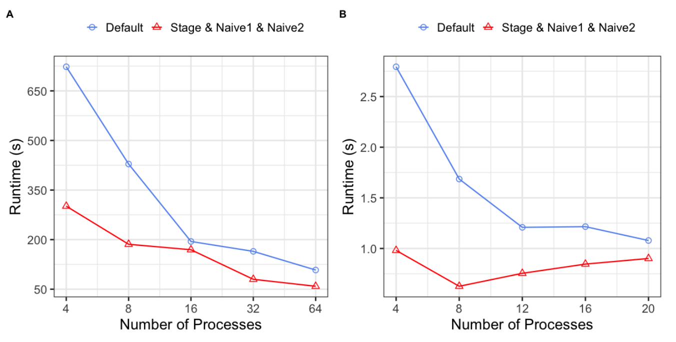


Figure 15: Evaluations with *Internal Extinction of Galaxies* with *MPI* mapping employing static deployments. A: HPC execution times with larger input dataset; B: laptop execution times.

can find that *Stage*, *Naive 1* and *Naive 2* outperform Default.

5.4. Analysis Based on Window Join Workflow

Figure 16 shows the execution times (runtime) of the different experiments conducted using the *Window Join* workflow in Cirrus (Figure 14.A), and in the laptop (Figure 14.B a). In this experiment, we have varied the number of cores from 4 to 64 in Cirrus and 4 to 16 in the Laptop. However, since this workflow has six PEs, all experiments conducted with *Default*, *Naive 1* and *Naive 2* techniques start with eight processes[15].

This workflow has a grouping in the last PE. Therefore, we have not run the *Dynamic* technique in this case since `Dynamic Scheduling` does not support groupings.

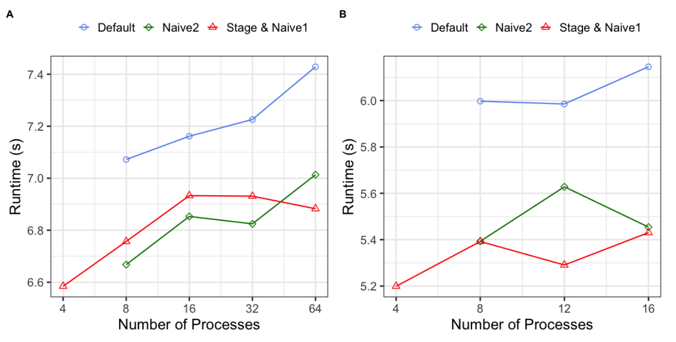


Figure 16: Evaluations of *Window Join* with *multi* mapping. A: Cirrus execution times employing all static techniques; B: Laptop execution times employing all static techniques.

Figure 17 illustrates that the performance with *MPI* mapping on HPC and laptop shares the similar performance with *multi* mapping, the same conclusion can be delivered: all static optimization methods with both *multi* and *MPI* mapping can apparently improve the default workflow, and the performance of those methods with the same mapping is close.

We can also observe that this workflow does not scale very efficiently. This is due to `Read` and `Join` PEs, which are not parallelizable. So adding more processes implies adding overhead in the execution of the workflow. Even so, our proposed

[15]The minimum number of processes required by this workflow is 6.

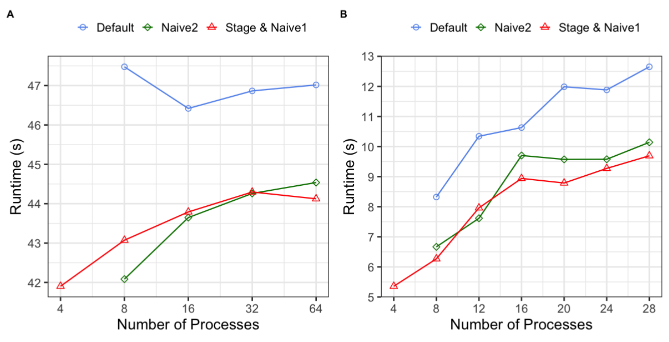


Figure 17: Evaluations with *Window Join* with *MPI* mapping employing static deployments. A: HPC execution times with larger input dataset; B: laptop execution times.

techniques outperform the *default* allocation technique employed by `dispel4py` for both computing infrastructures. Note that the *Stage* and *Naive 1* have made the same allocation of resources for this workflow.

5.5. Workload Skewness

As we mentioned earlier, one of the main features of the `Dynamic Scheduling` Technique is its availability to deal with workload skewness. However, none of our previous use cases has distribution fluctuations in their incoming data stream. Therefore, for our next experiment, we modified the *Internal Extinction of Galaxies* introduced in Section 3.2 and introduced an imbalance in the workload to show the effectiveness of `Dynamic Scheduling` regarding unexpected workload deviations.

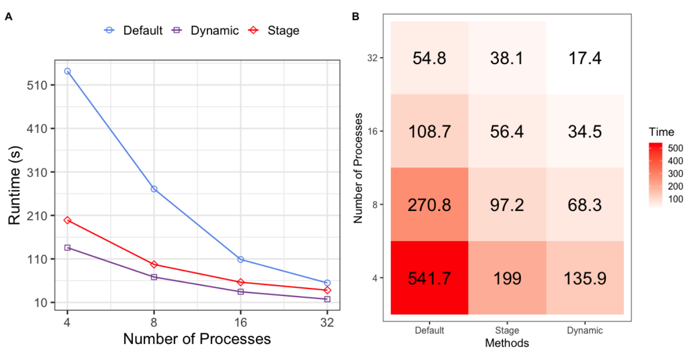


Figure 18: Evaluations of Internal Extinction of Galaxies with *multi* mapping using an unbalanced workload. The table shows the runtime (in seconds), for each technique and number of processes.

Google Cloud [16] was selected for this experiment using up to 32 processes. Due to all of our static techniques (`Naive 1`, `Naive 2`, and `Stage`) have calculated the same allocation (partitions and number of processes), as we can see in Figures 14.A and 14.C, we selected `Stage` as our static technique for this experiment. Furthermore, for `Naive Assignment` technique to become effective, the workflow has to have exactly the same

[16]By the time we performed this experiment, we did not have access to the previous HPC-Cluster platforms.

conditions as its previous recorded execution, including the workload. Since this is not the case for this experiment (we simulate workload skewness by modifying the workload distribution), we discarded this technique.

In this experiment, we included `Default` as the runtime baseline. Figure 18 illustrates the performance of the selected techniques with *MULTI* mapping, since `Dynamic` has been only implemented for this mapping. We can observe that `Dynamic` outperforms `Stage` in this type of scenario when we have an unbalanced workload since `Stage` only takes into account the communication pattern (topology of the concrete workflow) and not the incoming data stream to find the suitable partitions.

5.6. Observations

		Seismology Cross-correlation		Internal Extinction of Galaxies		Window Join	
		HPC	*Laptop*	*HPC*	*Laptop*	*HPC*	*Laptop*
Is Naive 1 algorithm Better than Default algorithm	*MULTI*	*Yes, but not significantly*	*Yes, significantly*	*Yes, significantly*	*Yes, significantly*	*Yes, significantly*	*Yes, significantly*
	MPI	*Yes, but not significantly*	*Yes, significantly*	*Yes, but not significantly*	*Yes, but not significantly*	*Yes, significantly*	*Yes, significantly*
Is Naive 2 algorithm Better than Default algorithm	*MULTI*	*Yes, but not significantly*	*Yes, significantly*	*Yes, significantly*	*Yes, significantly*	*Yes, significantly*	*Yes, significantly*
	MPI	*Yes, significantly*	*Yes, significantly*	*Yes, but not significantly*	*Yes, but not significantly*	*Yes, significantly*	*Yes, significantly*
Is Stage algorithm Better than Default algorithm	*MULTI*	*Yes, but not significantly*	*Yes, significantly*	*Yes, significantly*	*Yes, significantly*	*Yes, significantly*	*Yes, significantly*
	MPI	*Yes, but not significantly*	*Yes, significantly*	*Yes, but not significantly*	*Yes, but not significantly*	*Yes, significantly*	*Yes, significantly*
Best Static Optimization Method(s)	*MULTI*	*Stage*	*Stage, Naive 2*	*Stage, Naive 1, Naive 2*	*Stage, Naive 1, Naive 2*	*Stage, Naive 1, Naive 2*	*Stage, Naive 1, Naive 2*
	MPI	*Stage, Naive 2*	*Naive 2*	*Stage, Naive 1, Naive 2*	*Stage, Naive 1, Naive 2*	*Stage, Naive 1, Naive 2*	*Stage, Naive 1, Naive 2*
Comparsion between Dynamic method with the best static algorithm		*No significantly difference*	*Stage is better than Dynamic but not significantly, Naive 2 is signicantly better than Dynamic*	*Stage, Naive 1 and Naive 2 are better than Dynamic but not significantly*	*Stage, Naive 1 and Naive 2 are significantly better than Dynamic*	*/*	*/*

Figure 19: Evaluation summary across techniques, use cases and platforms based on execution time and T-test. This summary does not include the results obtained in Section 5.5 for the *Internal Extinction of Galaxies*, as we modified the original workflow to perform a particular experiment under workload skewness.

A summary of the different evaluations across techniques, use cases, and platforms can be seen in Figure 19. We have also conducted statistical analysis[17]. In statistical analysis, we leverage a T-test based on the experimental data to find out whether there is a significant difference between any two methods' performances. To employ the hypothesis testing, we assume two

	Naive Assignment Algorithm (Naive 2)		**Staging Algorithm (Stage)**	**Dynamic Scheduling Algorithm (Dynamic)**
	Naive 1	**Naive 2**		
Does it require ***previous execution log***?	🟨 ***Yes***	🟨 ***Yes***	🟦 ***No***	🟦 ***No***
Does it work with ***grouping***?	🟦 ***Yes***	🟦 ***Yes***	🟦 ***Yes***	🟨 ***No***
Can it efficiently handle ***input data with unbalance workloads***?	🟨 ***No***	🟨 ***No***	🟨 ***No***	🟦 ***Yes***
Does it work with a ***distributed-memory platform***?	🟦 ***Yes***	🟦 ***Yes***	🟦 ***Yes***	🟨 ***No***
Does it cause lower ***overhead***?	🟦 ***Yes***	🟦 ***Yes***	🟦 ***Yes***	🟨 ***No***
Does it work with workflow ***automatically***?	🟨 ***No***	🟨 ***No***	🟦 ***Yes***	🟦 ***Yes***
May it work when ***given processes are less than the number of PEs***?	🟨 ***No***	🟨 ***No***	🟦 ***Yes***	🟦 ***Yes***
Does it require a ***predefined number of processes***?	🟦 ***No***	🟨 ***Yes***	🟦 ***No***	🟦 ***No***
Does it realize satisfying ***performance for more applicable scenarios***?	🟨 ***No***	🟦 ***Yes***	🟦 ***Yes***	🟦 ***Yes***

🟦 ***Advantage*** 🟨 ***Disadvantage***

Figure 20: Reference table to select which optimization method to chose.

hypothesises: null hypothesis (*no significant performance difference between both methods*) and alternative hypothesis (*significant performance difference between both methods*). The p-value calculated by the T-test represents the probability that the null hypothesis is true, which is compared with the significant level α. We adopt 0.05 for α to control the level of preciseness. Figure 19 shows that if the P-value is smaller than 0.05, then we will reject the null hypothesis and accept the alternative hypothesis, meaning that one method's performance is significantly better or worse than another [18]. Otherwise, we conclude that there is no significant difference.

In conclusion, we combine the analysis and experimental evaluation to bring users the convenience of choosing the adaptive optimization methods. There are nine aspects concerning about the feasibility of those three optimization methods that users need to consider when they plan to apply the optimization methods for improving a new workflow: *previous execution log*, *grouping*, *workload distribution of input data*, *memory architecture of platform*, *overhead*, *automation*, *given number of processes*, *number of PEs and performance*. The reference table in Figure 20 based on those aspects not only determines the adaptive capacity of proposed optimization algorithms (for example, `Staging` has the least yellow squares, which means

[17]A statistical analysis can be found in Section 5 of this document `https://drive.google.com/drive/folders/1GAUR8vhcWCk4Y5NtigTgK2z81V7Nk4Ls?usp=sharing`

[18]`https://en.wikipedia.org/wiki/One-_and_two-tailed_tests`

there is less limitation for this algorithm) but also supports users to select a suitable algorithm for their workflows.

6. Applying static optimizations to DARE workflows

DARE [33][19] project focused on empowering domain experts to invent and improve their methods and models by providing a new platform and a working environment, in which analysis tools and workflow systems are readily available. DARE worked with two scientific communities: Seismology (EPOS[20]) and Climate (IS-ENES[21]) Research Infrastructures. And several `dispel4py` workflows were generated within the project.

In this work, we wanted to evaluate our optimization techniques using one of the DARE workflows since they represent the type of applications that domain scientists are currently working on, and we also wanted to test them making use of a Cloud platform. Therefore, we selected the Rapid Ground Motion Assessment (*RA*) (see Figure 21), a seismic application developed by INGV [22].

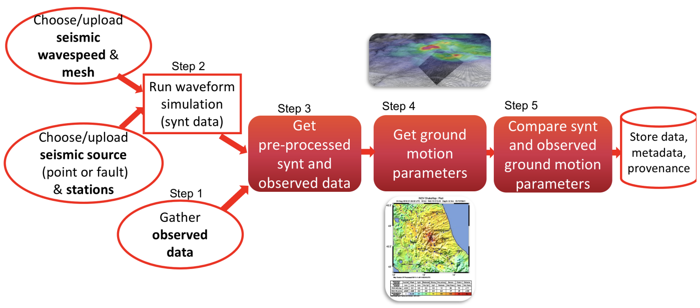

Choose/upload seismic wavespeed & mesh
Step 2
Run waveform simulation (synt data)
Choose/upload seismic source (point or fault) & stations
Step 1
Gather observed data
Step 3
Get pre-processed synt and observed data
Step 4
Get ground motion parameters
Step 5
Compare synt and observed ground motion parameters
Store data, metadata, provenance

Figure 21: Rapid Ground Motion Assessment application (*RA*).

RA aims to model the strong ground motion after large earthquakes in order to make a rapid assessment of the earthquake's impact, also in the context of emergency response. It has five main phases: (1) to select an earthquake gathering the real observed seismic wavefield, (2) to simulate synthetic seismic waveforms corresponding to the same earthquake using SPECFEM3D [34], an MPI-based parallel software; (3) to pre-process both synthetic and real data. This step is quite similar to the preprocess step included in the *Seismic Cross-correlation* workflow; (4) to calculate the ground motion parameters of displacement, velocity and acceleration for synthetic and real data; In this step, two types of normalisation are applied (mean and max), generating; as a result of two types of PGM outputs (pgm_mean and pgm_max); (5) to compare them with each other by creating shake maps.

Originally, these 5 steps were programmed as independent `dispel4py` workflows, but in this work, we have aggregated steps 3 to 5 in a single workflow (see Figure 22) to create a more complex `dispel4py` workflow.

[19]`http://project-dare.eu/`
[20]`https://www.epos-ip.org/`
[21]`https://is.enes.org/`
[22]https://www.ingv.it/

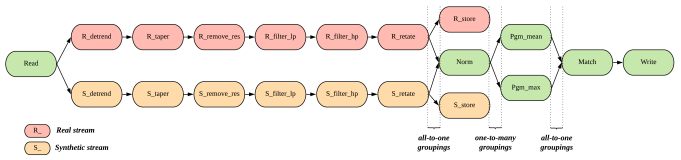


Figure 22: *RA* steps 3 to 5 as a single `dispel4py` workflow.

We run three experiments using the new `dispel4py` *RA* workflow with the following configuration: multiprocessing mapping, 30 processes, and the Google Cloud platform described in Section 5.1. The first experiment consisted of running the workflow without optimizations, whereas the others consisted in applying the `Naive Assignment` and `Staging` techniques, respectively. We did not use the `Dynamic Scheduling` because the workflow applies several groupings, and those are not compatible with this technique.

All experiments were repeated 10 times, using 31 seismic stations as input data (15MB of real and synthetic streams), generating 5MB as output data. The average execution times per experiment were: i) 4 seconds using the original `dispel4py` ii) 3.66 seconds using the full `Naive Assignment`; and iii) 3.77 seconds using `Staging`. Once again, we get lower execution times using our optimization techniques since they are able to perform a better allocation of processes to PEs than the original `dispel4py` framework, having a direct impact on the execution time of the workflow.

7. Conclusion and Future work

In this paper, we have presented three adaptive optimization techniques for improving the performance and scalability of stream-based `dispel4py` workflows. Two of them have been proposed for the static deployment of `dispel4py` workflows: `Naive Assignment` and `Staging`. Both techniques take into account the communication patterns, but they apply a different set of heuristics to group PEs together into partitions, and assign later the suitable number of processes to each partition. While `Naive Assignment` is an *offline* technique which relies on previous recorded measures of the same workflows, `Staging` is an *online* technique, which analyses the dependencies (topology) between PEs specified in the workflow. The `Naive Assignment` static technique has been divided into two sub-techniques: *Naive 1*, in which we only apply the Algorithm 1; and *Naive 2*, in which we apply both algorithms of this technique.

The third technique, `Dynamic Scheduling`, enables to run `dispel4py` workflow with a dynamic deployment. Processes in this technique are not locked to specific PEs, scheduling PE instances on-the-fly and re-balancing automatically the graph without stopping the workflow execution.

Our proposed techniques have been evaluated in four different computing infrastructures to test their effectiveness and adaptivity across platforms with different features (number of processes, hardware components, network, etc.). Furthermore, we have selected four use cases, three from real domains and one synthetic application, with different features (number of

PEs, connectivity, groupings, etc.). Given that all uses cases selected for this work do not have distribution fluctuations in their incoming data stream, we modified the *Internal Extinction of Galaxies* in Section 5.5 to introduce an imbalance in the workload to show the effectiveness of `Dynamic Scheduling` dealing with workload skewness.

As we outlined in the conclusions of our previous work [35], we have taken this opportunity to test our optimizations in the context of DARE, in which several domain scientists have developed their applications using `dispel4py`. In this work, we run part of the *Rapid Ground Motion Assessment application*, developed by the Seismology community, with and without static optimizations using a Cloud environment. Results show that we get a lower execution time when our optimizations are applied.

The evaluations shown in Sections 5 and 6, demonstrate that all our proposed techniques outperform the default allocation of resources performed by `dispel4py`. Among the static techniques, the `Staging` usually gives us the best allocation parameters, allowing workflows to scale up better. This is because data shuffling incurs a huge communication cost, being the major bottleneck in those uses cases. So, `Staging` successfully minimizes the data shuffling reducing the overall execution time. However, we have to take into account that our original use cases do not have fluctuations in their incoming data stream. When that is the case, as we demonstrated in Section 5.5, the `Dynamic Scheduling` is the best strategy to use since it is the only one capable of dealing with unexpected workload fluctuation.

As future work, we plan to test our techniques using other `dispel4py` mappings, such as *MPI* [8], and also compare them with other state-of-the-art algorithms for workflow scheduling in IaaS clouds [36]. Both static techniques are applicable across mappings. Nevertheless, for the `Dynamic Scheduling` technique, we will need to modify it first to adapt it to distributed-memory architectures. This change will require applying another type of global queues, such as *Apache Kafka*, *Redis* or *ZeroMQ* messaging frameworks. We plan to work in this technique, so workflows with groupings will be able to be enacted dynamically.

Acknowledgements

This work is partially supported by the EU H2020 project DARE, No. 777413; and by Google Cloud Platform research credits program.

References

[1] F. J. Montáns, F. Chinesta, R. Gómez-Bombarelli, J. N. Kutz, Data-driven modeling and learning in science and engineering, Comptes Rendus Mécanique 347 (11) (2019) 845 – 855, data-Based Engineering Science and Technology. doi:https://doi.org/10.1016/j.crme.2019.11.009.
URL `http://www.sciencedirect.com/science/article/pii/S1631072119301809`

[2] E. Deelman, T. Peterka, I. Altintas, C. D. Carothers, K. K. van Dam, K. Moreland, M. Parashar, L. Ramakrishnan, M. Taufer, J. Vetter, The future of scientific workflows, The International Journal of High Performance Computing Applications 32 (1) (2018) 159–175. arXiv:https://doi.org/10.1177/1094342017704893, doi:10.1177/1094342017704893.
URL `https://doi.org/10.1177/1094342017704893`

[3] M. Atkinson, S. Gesing, J. Montagnat, I. Taylor, Scientific workflows: Past, present and future (2017).

[4] T. Akidau, The world beyond batch: Streaming 101, A High-Level Tour of Modern Data-Processing Concepts. Blog entry (2015).

[5] R. Filgueira, A. Krause, M. Atkinson, I. Klampanos, A. Moreno, dispel4py: A python framework for data-intensive scientific computing, International Journal of High Performance Computing Applications (IJHPCA) (2016).

[6] R. Filgueira, A. Krause, M. Atkinson, I. Klampanos, A. Spinuso, S. Sanchez-Exposito, dispel4py: An agile framework for data-intensive escience, in: 2015 IEEE 11th International Conference on e-Science, IEEE, 2015, pp. 454–464.

[7] I. A. Klampanos, F. Magnoni, E. Casarotti, C. Pagé, M. Lindner, A. Ikonomopoulos, V. Karkaletsis, A. Davvetas, A. Gemünd, M. Atkinson, A. Koukourikos, R. Filgueira, A. Krause, A. Spinuso, A. Charalambidis, Dare: A reflective platform designed to enable agile data-driven research on the cloud, 2019 15th International Conference on eScience (eScience) (2019) 578–585.

[8] Openmpi: Open source high performance computing, `https://www.open-mpi.org`.

[9] Apache storm, `http://storm.apache.org`.

[10] Q. Jiang, Y. C. Lee, M. Arenaz, L. M. Leslie, A. Y. Zomaya, Optimizing scientific workflows in the cloud: A montage example, in: 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing, 2014, pp. 517–522.

[11] H. A. Nguyen, Z. van Iperen, S. Raghunath, D. Abramson, T. Kipouros, S. Somasekharan, Multi-objective optimisation in scientific workflow, in: P. Koumoutsakos, M. Lees, V. V. Krzhizhanovskaya, J. J. Dongarra, P. M. A. Sloot (Eds.), International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland, Vol. 108 of Procedia Computer Science, Elsevier, 2017, pp. 1443–1452. doi:10.1016/j.procs.2017.05.213.
URL `https://doi.org/10.1016/j.procs.2017.05.213`

[12] A. M. Chirkin, A. S. Belloum, S. V. Kovalchuk, M. X. Makkes, M. A. Melnik, A. A. Visheratin, D. A. Nasonov, Execution time estimation for workflow scheduling, Future Generation Computer Systems 75 (2017) 376–387. doi:https://doi.org/10.1016/j.future.2017.01.011.
URL `https://www.sciencedirect.com/science/article/pii/S0167739X17300304`

[13] E. N. Alkhanak, S. P. Lee, R. Rezaei, R. M. Parizi, Cost optimization approaches for scientific workflow scheduling in cloud and grid computing: A review, classifications, and open issues, Journal of Systems and Software 113 (2016) 1–26.

[14] I. Pietri, R. Sakellariou, Scheduling data-intensive scientific workflows with reduced communication, in: D. Sacharidis, J. Gamper, M. H. Böhlen (Eds.), Proceedings of the 30th International Conference on Scientific and Statistical Database Management, SSDBM 2018, Bozen-Bolzano, Italy, July 09-11, 2018, ACM, 2018, pp. 25:1–25:4. doi:10.1145/3221269.3221298.
URL `https://doi.org/10.1145/3221269.3221298`

[15] R. Prodan, M. Wieczorek, Bi-criteria scheduling of scientific grid workflows, IEEE Transactions on Automation Science and Engineering 7 (2) (2009) 364–376.

[16] L. Ramakrishnan, J. S. Chase, D. Gannon, D. Nurmi, R. Wolski, Deadline-sensitive workflow orchestration without explicit resource control, Journal of Parallel and Distributed Computing 71 (3) (2011) 343–353.

[17] M. Tanaka, O. Tatebe, Workflow scheduling to minimize data movement using multi-constraint graph partitioning, in: 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012), IEEE, 2012, pp. 65–72.

[18] A. Spark, Apache spark, Retrieved January 17 (2018) 2018.

[19] M. Bertolucci, E. Carlini, P. Dazzi, A. Lulli, L. Ricci, Static and dynamic big data partitioning on apache spark, in: PARCO, 2015.

[20] E. H. Houssein, A. G. Gad, Y. M. Wazery, P. N. Suganthan, Task scheduling in cloud computing based on meta-heuristics: Review, taxonomy, open challenges, and future trends, Swarm and Evolutionary Computation 62 (2021) 100841. doi:https://doi.org/10.1016/j.swevo.2021.100841.
URL `https://www.sciencedirect.com/science/article/pii/S221065022100002X`

[21] H. Topcuoglu, S. Hariri, M.-Y. Wu, Performance-effective and low-

complexity task scheduling for heterogeneous computing, IEEE Transactions on Parallel and Distributed Systems 13 (3) (2002) 260–274. doi:10.1109/71.993206.

[22] X. Wu, M. Deng, R. Zhang, B. Zeng, S. Zhou, A task scheduling algorithm based on qos-driven in cloud computing, Procedia Computer Science 17 (2013) 1162–1169, first International Conference on Information Technology and Quantitative Management. doi:https://doi.org/10.1016/j.procs.2013.05.148.
URL `https://www.sciencedirect.com/science/article/pii/S1877050913002810`

[23] H. R. Faragardi, M. R. Saleh Sedghpour, S. Fazliahmadi, T. Fahringer, N. Rasouli, Grp-heft: A budget-constrained resource provisioning scheme for workflow scheduling in iaas clouds, IEEE Transactions on Parallel and Distributed Systems 31 (6) (2020) 1239–1254. doi:10.1109/TPDS.2019.2961098.

[24] D. Amalarethinam, T. Beena, Workflow scheduling for public cloud using genetic algorithm (wsga), 2016.

[25] H. Nasiri, S. Nasehi, A. Divband, M. Goudarzi, A scheduling algorithm to maximize storm throughput in heterogeneous cluster, ArXiv abs/2001.10308 (2020).

[26] L. Eskandari, Z. Huang, D. Eyers, P-scheduler: Adaptive hierarchical scheduling in apache storm, in: Proceedings of the Australasian Computer Science Week Multiconference, ACSW '16, Association for Computing Machinery, New York, NY, USA, 2016. doi:10.1145/2843043.2843056.
URL `https://doi.org/10.1145/2843043.2843056`

[27] K. Aziz, D. Zaidouni, M. Bellafkih, Leveraging resource management for efficient performance of apache spark, Journal of Big Data 6 (1) (2019) 78. doi:10.1186/s40537-019-0240-1.
URL `https://doi.org/10.1186/s40537-019-0240-1`

[28] R. Montoliu, Z. Li, J. Yu, C. Bian, Y. Pu, Y. Wang, Y. Zhang, B. Guo, Flink-er: An elastic resource-scheduling strategy for processing fluctuating mobile stream data on flink, Mobile Information Systems 2020 (2020) 5351824. doi:10.1155/2020/5351824.
URL `https://doi.org/10.1155/2020/5351824`

[29] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, I. Stoica, Apache spark: A unified engine for big data processing, Commun. ACM 59 (11) (2016) 56–65. doi:10.1145/2934664.
URL `https://doi.org/10.1145/2934664`

[30] R. Filgueira, A. Krause, A. Spinuso, I. Klampanos, P. Danecek, M. Atkinson, Dispel4py: An open-source python library for data-intensive seismology, EGUGA (2015) 6790.

[31] H. G. Kim, Y. H. Park, Y. H. Cho, M. H. Kim, Time-slide window join over data streams, Journal of Intelligent Information Systems 43 (2) (2014) 323–347.

[32] J. Fang, R. Zhang, T. Z. Fu, Z. Zhang, A. Zhou, J. Zhu, Parallel stream processing against workload skewness and variance, Association for Computing Machinery, New York, NY, USA, 2017. doi:10.1145/3078597.3078613.
URL `https://doi.org/10.1145/3078597.3078613`

[33] I. A. Klampanos, C. Themeli, A. Spinuso, R. Filgueira, M. Atkinson, A. Gemünd, V. Karkaletsis, DARE platform: a developer-friendly and self-optimising workflows-as-a-service framework for e-science on the cloud, J. Open Source Softw. 5 (54) (2020) 2664. doi:10.21105/joss.02664.
URL `https://doi.org/10.21105/joss.02664`

[34] D. Peter, D. Komatitsch, Y. Luo, R. Martin, N. L. Goff, E. Casarotti, P. L. Loher, F. Magnoni, Q. Liu, C. Blitz, T. Nissen-Meyer, P. Basini, J. Tromp, Forward and adjoint simulations of seismic wave propagation on fully unstructured hexahedral meshes, Geophys. J. Int. 186 (2011) 721–789.

[35] L. Liang, R. Filguiera, Y. Yan, Adaptive optimizations for stream-based workflows, in: 2020 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS), 2020, pp. 33–40. doi:10.1109/WORKS51914.2020.00010.

[36] M. A. Rodriguez, R. Buyya, Budget-driven scheduling of scientific workflows in iaas clouds with fine-grained billing periods, ACM Trans. Auton. Adapt. Syst. 12 (2) (May 2017). doi:10.1145/3041036.
URL `https://doi.org/10.1145/3041036`