Adaptive Optimizations for Stream-based Workflows

Adaptive Optimizations for Stream-based Workflows

Liang Liang
EPCC
University of Edinburgh
Edinburgh, UK
s1980912@ed.ac.uk

Rosa Filguiera
EPCC
University of Edinburgh
Edinburgh, UK
r.filgueira@epcc.ed.ac.uk

Yan Yan
Faculty of Medicine
Imperial College London
London, UK
y.yan20@imperial.ac.uk

***Abstract*—This work presents three new adaptive optimization techniques to maximize the performance of dispel4py workflows. dispel4py is a parallel Python-based stream-orientated 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 workflows' features, which can cause scalability issues, specially for data-intensive scientific workflows. Therefore, our aim is to improve the performance of dispel4py workflows by testing different workload strategies that automatically adapt to workflows. For achieving this objective, we have implemented three new techniques, called `Naive Assignment`, `Staging` and `Dynamic Scheduling`. The evaluations show that our proposed techniques have significantly improved the performance of the original dispel4py framework.**

***Index Terms*—Scientific workflow, Stream-based workflow, Workflow optimization, dispel4py.**

I. 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 enormous complex dataset that require 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 a lot of 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 its abilities to process unlimited data flows as well as providing lower latency [4]. Therefore, many stream-based workflow systems have been implemented for solving diverse objectives, including `dispel4py` [5]. `dispel4py` is a python library for data-intensive 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], and provides smooths transitions from local development to scalable executions.

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. Then, dispel4py automatically maps those abstract workflows to concrete ones, depending on the selected enactment engine. Since the abstract workflows are independent from 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. Depending on the number of targeted processes, which user specifies when executing a `dispel4py` workflow, multiple instances of each PE are created to make use of all available processes. The default workload allocation is performed by dividing the number of processes between the number of PEs, with the exception of first PE, which is always assigned to one process to prevent the generation of duplicate data-blocks. This default allocation neither takes into account the date-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 that a PE needs 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`, called `Naive Assignment` and `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 on optimisations is to enable dynamic deployment of `dispel4py` workflow. Therefore, we have developed the `Dynamic Scheduling` technique to enable dispel4py to allocate resources dynamically while a workflow is running.

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

All three techniques have been compared with the default dispel4py workload allocation, and evaluated in two computer infrastructures: HPC Cluster (Cirrus) and Laptop.

The rest of the paper is structured as follows. Section II presents the relevant background. Section III presents three dispel4py workflows: *Seismic Cross-correlation*, *Internal Extinction of Galaxies* and *Window Join*. Section IV presents the different optimization techniques. Using the previous dispel4py workflows as case-studies, we evaluate in Section V the optimization techniques on different platforms. We conclude in Section VI with a summary of achievements and outline some future work.

II. Background

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

A. dispel4py concepts

There are some important dispel4py concepts [5] that need to be explained first:

- *Processing elements* (PE) is the computational activity for processing task or transforming data which can be considered as the node in the workflow graph. PEs in dispel4py are connected by specifying the input and output; the data will be passed among connected PEs in the manner of 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 compute process. A PE could be assigned to more than one instances.
- *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 the enactment period, 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-allocated multiple PEs into one process. Currently, the user should define partitions manually. Otherwise, each PE is allocated to one partition automatically.
- *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.

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 run-time 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, getting then the number of PE instances which will be assigned to processes. The only exception is the first PE, which will be only assigned to one process. An example of the default allocation can be seen in Figure 1, in which four PEs have been allocated to seven processes using the *mutli* mapping.

It is important to highlight that for running a dispel4py workflow using *MPI* and *multi* mappings, a user needs to indicate a greater or equal number of processes than PEs. Otherwise, dispel4py would rise an error.

In this work, we have selected the *multi* mapping to evaluate our proposed techniques. This mapping is ideal for working in shared-memory architectures.

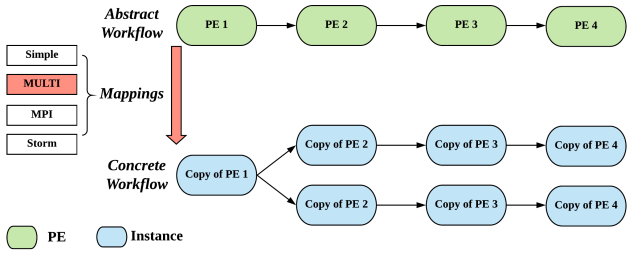

Fig. 1. Example of the default workload allocation. In this example, the user has indicated to run the workflow using the *multi* mapping with seven processes. Each PE instance runs in a different process.

B. Related Work

There are many optimization techniques proposed for improving the performance of scientific workflows [10], [11]. 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. Whereas, scheduling techniques are usually classified either static or dynamic deployment.

We have noticed that scientific workflows mostly use heuristics and meta-heuristics as the optimization method to improve their performance [12], [13], being *Dynamic Constraint Algorithm* (DCA) [14] and *Workflow Orchestrator for Distributed Systems* (WORDS) [15] two representative examples. DCA is a user-friendly method for handling issues of bi-criteria of dynamic scheduling. However, the performance of DCA may decay when some criteria require more time to meet. 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.

Furthermore, partitioning methods, such as *Multi-Constraint Graph Partitioning* (MCGP) [16], 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* [17]. *Apache Spark* applies a partitioning method [18] 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.

Regarding the scheduling techniques, most of scientific workflows apply a static deployment [12], 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 less resources, it dynamically up-scale or down-scale, without stopping the workflow execution.

In this work, we have employed two optimization methods to improve the current static deployment in `dispel4py`. The first one is the `Naive Assignment` technique for which we have implemented two new heuristics. While in the `Staging` technique, we have employed a partitioning method. Both optimization methods presented in this paper aim to reduce the overall workflow execution time by minimizing the communication time among PEs. Furthermore, we have also developed a new scheduling technique, `Dynamic Scheduling`, to enable dynamic deployment in `dispel4py`. These are introduced in Section IV.

III. Use Cases

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

A. Seismic Cross-Correlation

The workflow has been designed to monitor and analyse 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].

Figure 2 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 from any other, making this phase embarrassingly parallel. Phase Two pairs all stations and calculates the cross-correlation for each pair.

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

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

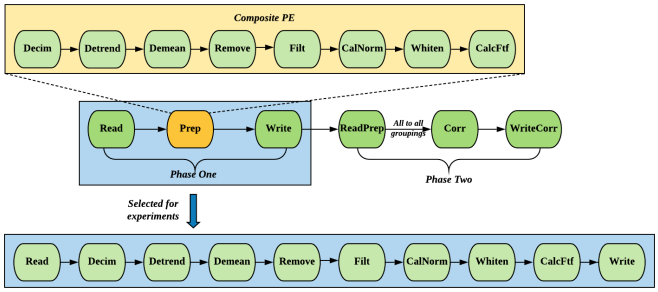

Fig. 2. A simplified abstract workflow for seismic cross-correlation.

B. Internal Extinction of Galaxies

This workflow has been implemented to calculate the extinction within the galaxies, which is a significant property in astrophysics [19]. 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 3, 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. Afterwords, `Filt` PE parses the VOTable by using astropy library and filters the parsed data by selecting needed columns. Finally, `Intext` PE calculates the internal extinction based on data from `Filt` PE.

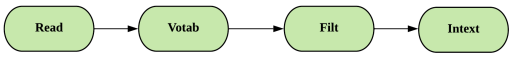

Fig. 3. Workflow for calculating the internal extinction of galaxies.

C. Synthetic Workflow - Window Join

We have developed this new synthetic workflow[6], which has a more complex topology than both previous workflows. The *Window Join* workflow aims to simulate a fundamental query operation, window join, in the streaming processing, which produces the result from unbounded streams by using concepts of the window to limit the scope of data for join [20].

As we can see in Figure 4, this workflow consists of six PEs connected via a fork-join manner. `Read` PE loads the data from Customer and Supplier TPC-H Tables[7]. Then, data is sent to `FilterCus` and `FilterSup` PEs. `FilterCus` selects

[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]https://git.ecdf.ed.ac.uk/msc-19-20/s1980912/tree/master/workflows/join

[7]http://www.tpc.org/tpch

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.

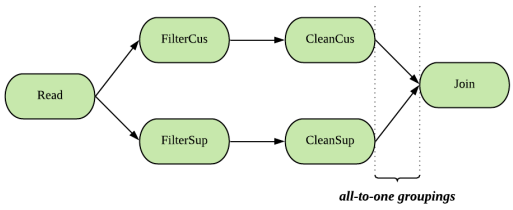

Fig. 4. Workflow for representing the *Window Join* synthetic application.

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.

IV. 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 are expected to be applicable to other stream-based workflow systems.

A. Naive Assignment Technique

`dispel4py` includes a monitoring framework, which collects the following information while a workflow is being executed: a) execution and communication times per PE b) number of PEs c) number of iterations d) data size e) mapping used.

We have developed a new technique, called `Naive Assignment`, which calculates the most efficient allocation parameters to run a workflow. Those are the number of partitions to divide the workflow and the number of processes to assign to each partition. Notice, that by default, when partitions are not indicated by the user, each PE runs in a single partition.

This technique relays on the information recorded in the monitoring framework, from a previous execution of the same workflow under the same conditions (same mapping, computing infrastructure, and number of processes). Therefore, after running a workflow with the monitoring framework activated, this technique analyses the execution times and communication times to discover which is the best workload allocation for that specific workflow under the same circumstances.

To calculate the most suitable number of partitions, we developed the Algorithm 1. This algorithm aims to to reduce the overall workflow execution time by minimizing the communication time among PEs. For achieving this goal, Algorithm 1 maximises the number of PEs assigned into the same partition. It groups into the same partition all connected PEs which their communication times are higher than their execution time. With the exception of the first PE, where the algorithm assigns a single partition to it.

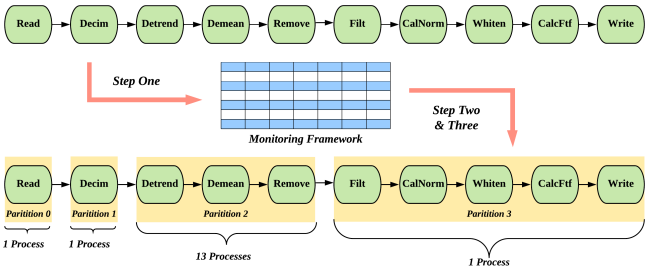

Fig. 5. Application of the `Naive assignment` technique over the *Seismic Cross-correlation* workflow using 16 processes.

Figure 5 shows an example of the `Naive Assignment` technique over the *Seismic Cross-correlation* workflow introduced in Section III-A.

Algorithm 1: Assigning Partition

Require: Workflow consisting of N PEs (PE_0, PE_1 ... PE_{N-1})
Require Execution time of each PE as $E(PE_i)$
Require: Communication time between adjacent PEs as $C(PE_i, PE_{i+1})$
for i = 0 **to** i = N-2 **do**
  if i = 0 **then**
    PE_i is assigned to single partition
  else
    if $C(PE_i, PE_{i+1})$ > MIN($E(PE_i)$, $E(PE_{i+1})$) **then**
      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
    end if
  end if
end for

The next step is to calculate the number of processes assigned to each partition previously calculated. 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 developed the 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 among 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), obtaining then the suitable number of processes per partition.

B. Staging Technique

Our next technique has been inspired by *Apache Spark Stage* method[8]. As, we introduced in Section II-B, *Apache Spark* uses a DAG to represent an the execution plan (job) of a program. In other words, this DAG represents the logical plan of operations. Once the DAG is created, Apache Spark divides

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

Algorithm 2: Assigning Process

Require: Workflow consisting of M $PARTs$ ($PART_0$, $PART_1$ ... $PART_{M-1}$) or including N PEs (PE_0, PE_1 ... PE_{N-1})
Require: Total number of processes $TotalNumProcess$
Require: Execution time of each PE as $E(PE_i)$
Define: Execution time of each partition as $E(PART_i)$
Define: Number of processes for each partition $NumProcess(PART_i)$
Define: Total execution time $E(TOTAL)$
for i = 1 **to** i = N-1 **do**
    $E(TOTAL) = E(TOTAL) + E(PE_i)$
end for
for i = 0 **to** i = M-1 **do**
    if i = 0 **then**
        $NumProcess(PART_i) = 1$
    else
        for PE in $PART_i$ **do**
            $E(PART_i) = E(PART_i) + E(PE)$
        end for
        $NumProcess(PART_i) = (TotalNumProcess - 1) \times \frac{E(PART_i)}{E(TOTAL)}$
    end if
end for

this DAG into a number of stages. These stages are then divided into smaller tasks and all the tasks are given to the executors (processes) for execution.

Apache Spark's Resilient Distributed Datasets (RDD) are a collection of various data that are so big in size, that they cannot fit into a single node and should be partitioned across various nodes. Apache Spark automatically partitions RDDs and distributes the partitions across different nodes.

Spark aggregates into a same stage all operations which do not require shuffling the data [9].

`Staging` aims to allocate the maximum number of PEs into the same partitions to reduce the communication cost, and therefore, the total execution time of `dispel4py` workflows. But unlike the `Naive Assignment` technique, `Staging` automatically creates the number of partitions by analysing the dependencies between PEs specified in the abstract 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 (source PE only connect to a destination PE); 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.

We can see an example of this technique applied to the *Window Join* workflow (introduced in Section III-C) in Figure 6.

[9]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.

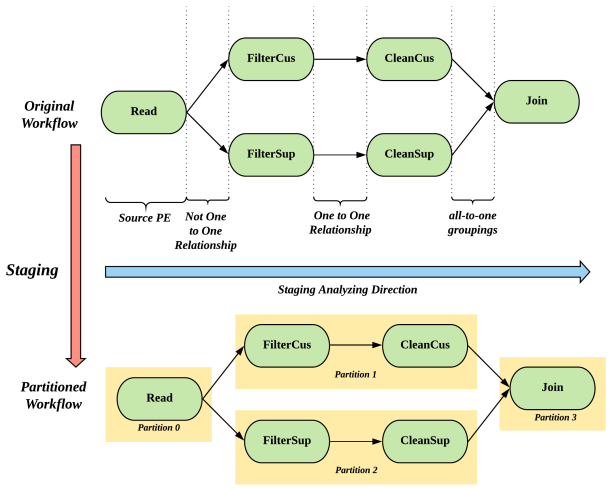

Fig. 6. Example of the `Staging` technique applied to *Window-Join* workflow.

C. Dynamic Scheduling Technique

The last technique proposed in this work is `Dynamic Scheduling`. This technique aims to enable dynamic deployment to `dispel4py`. 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 up-scale or down-scale, re-balancing automatically the graph, without stopping the workflow execution.

The implementation of this technique is based on the Python multiprocessing[10] 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 dependencies between PEs) at the beginning of the workflow execution. `Dynamic scheduling` uses a global queue to keep PEs and data coming up, which is available to all processes. The main idea is that each process as soon as is "free", goes to the global queue to *pull* the next PE to execute along with the necessary data. And then, 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. The reason is that if a user uses a grouping over a PE, all data received by this particular PE has to go to a particular PE instance running always in the same process. The current implementation of the `Dynamic Scheduling` technique, can not guarantee such behaviour.

An example of `Dynamic Scheduling` applied to the *Internal Extinction of Galaxies* workflow (introduced in Section III-B) can be seen in Figure 7.

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

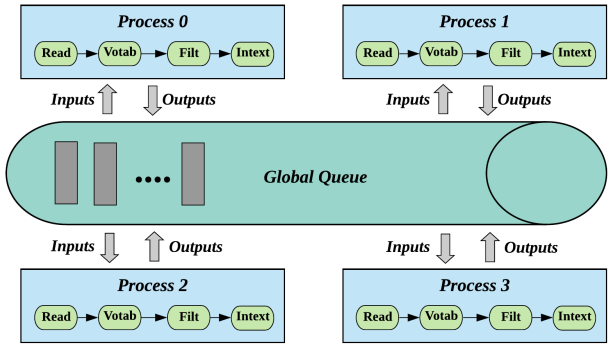

Fig. 7. Example of the `Dynamic Scheduling` technique applied to the *Internal Extinction of Galaxies* workflow.

V. Experimental Evaluation

This section presents the experimental evaluations for the proposed techniques in Section IV. For each experiment, we have run one of the workflows introduced previously using the *multi* mapping, modifying the number of processes, and selecting one of the follow 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. 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 to run 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, 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. 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 currently is not compatible with groupings, 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.

A. 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 Linus as the OS[11]. Since, we have selected the *multi* mapping, our experiments only just use a Cirrus node. Cirrus standard compute nodes each contain two 2.1 GHz, 18-core Intel Xeon E5-2695 (Broadwell) series processors. Each of the cores in these processors supports 2 hardware threads (Hyperthreads), which are enabled by default. The standard compute nodes on Cirrus have 256 GB of memory shared between the two processors. This means that we can use up to 72 processes for our experiments in Cirrus.
- Laptop (Local): it uses macOS Catalina as the OS, and it has a 2.3 GHz 8-Core i9 processor with 16 GB memory. This means we can use up to 16 processes for our experiments using this laptop.

B. Analysis Based on the Seismic Cross-correlation Workflow

Figure 8 shows the execution times (Run-time) of the different experiments conducted using the *Seismic Cross-correlation* workflow in Cirrus (Figure 8.A and Figure 8.B), and in the laptop (Figure 8.C and Figure 8.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[12].

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

C. Analysis Based on Internal Extinction of Galaxies

Figure 9 shows the execution times (Run-time) of the different experiments conducted using the *Internal Extinction of Galaxies* workflow in Cirrus (Figure 9.A and Figure 9.B), and in the laptop (Figure 9.C and Figure 9.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 in the most suitable allocation of resources (number of partitions and number of processes assigned to each of them). Figure 9.A and 9.C show that this allocation performs better

[11] https://www.cirrus.ac.uk/about/

[12] The minimum number of processes required by this workflow is 10.

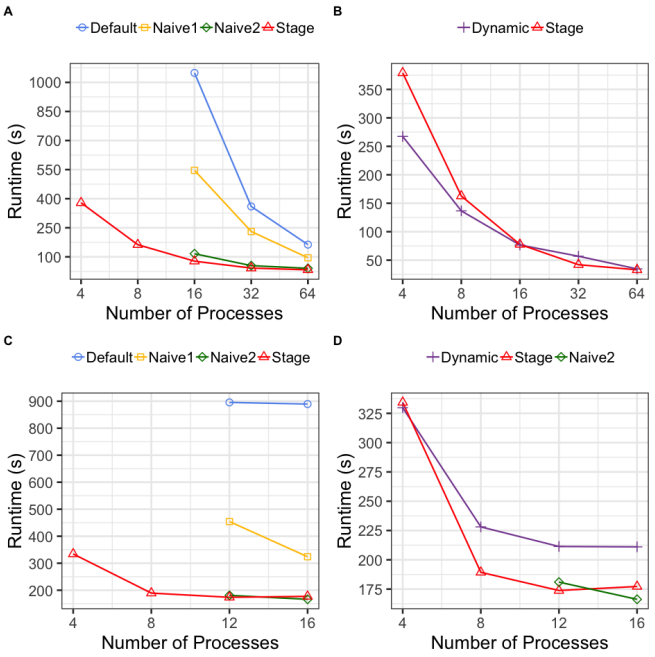

Fig. 8. Evaluations with *Seismic Cross-correlation*. A: Cirrus execution times employing all static techniques; B: Cirrus execution times employing the dynamic technique and best static deployment(s). C: Laptop execution times employing all static techniques; D: Laptop execution times using the dynamic technique and best static deployment(s).

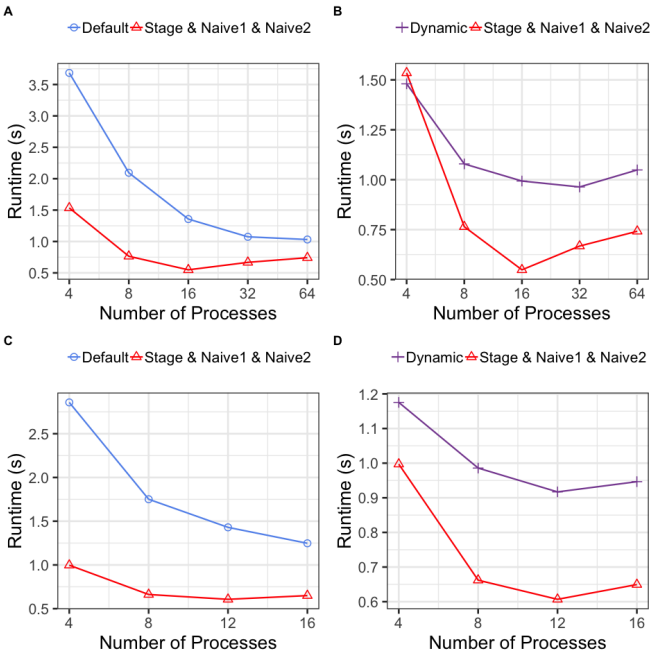

Fig. 9. Evaluations with *Internal Extinction of Galaxies*. A: Cirrus execution times employing all static techniques; B: Cirrus execution times employing the dynamic technique and best static deployment(s). C: Laptop execution times employing all static techniques; D: Laptop execution times using the dynamic technique and best static deployment(s).

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.

D. Analysis Based on Window Join Workflow

Figure 10 shows the execution times (Run-time) of the different experiments conducted using the *Window Join* workflow in Cirrus (Figure 9.A), and in the laptop (Figure 9.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[13].

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.

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 to add overhead in the execution of the workflow. Even so, our proposed 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.

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

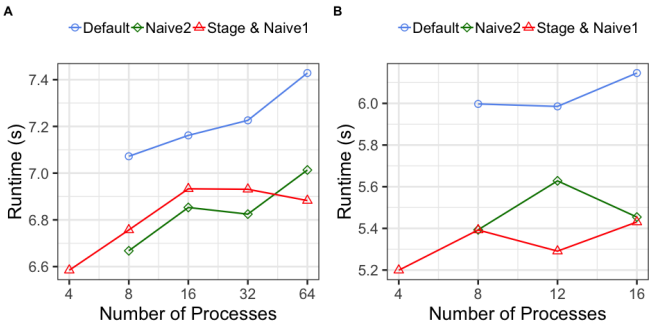

Fig. 10. Evaluations with *Window Join*. A: Cirrus execution times employing all static techniques; B: Laptop execution times employing all static techniques.

E. Observations

A summary of the different evaluations across techniques, use cases, and platforms can be seen in Figure 11.

VI. 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`. And the third technique, `Dynamic Scheduling`, enables to run `dispel4py` workflow with a dynamic deployment. The `Naive Assignment` static technique has been divided into two sub-techniques: *Naive 1*, in which we only apply the

	Seismology Cross-correlation		Internal Extinction of Galaxies		Window Join	
	Cirrus	*Laptop (Local)*	*Cirrus*	*Laptop (Local)*	*Cirrus*	*Laptop (Local)*
Is the Naive 1 algorithm Better than the Default algorithm	*Yes*	*Yes*	*Yes*	*Yes*	*Yes*	*Yes*
Is the Naive 2 algorithm Better than the Default algorithm	*Yes*	*Yes*	*Yes*	*Yes*	*Yes*	*Yes*
Is the Stage algorithm Better than the Default algorithm	*Yes*	*Yes*	*Yes*	*Yes*	*Yes*	*Yes*
Best Static Optimization Method(s)	*Stage*	*Stage, Naive 2*	*Stage, Naive 1, Naive 2*	*Stage, Naive 1, Naive 2*	*Stage, Naive 1, Naive 2*	*Stage, Naive 1, Naive 2*
Comparsion between the dynamic method with the best static algorithm	*Similar scalability*	*Stage and Naive 2 scale slightly better than Dynamic*	*Stage, Naive 1 and Naive 2 scale slightly better than Dynamic*	*Stage, Naive 1 and Naive 2 scale better than Dynamic*	/	/

Fig. 11. Overall Evaluation

Algorithm 1; and *Naive 2*, in which we apply both algorithms of this technique.

Our proposed techniques have been evaluated in two 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 three use cases, two from real domains and another synthetic application, with different features (number of PEs, connectivity, groupings, etc.).

The evaluations shown in Section V, demonstrate that all our proposed techniques outperform the default allocation of resources performed by `dispel4py`. Among the static techniques, the `Staging` technique usually gives us the best allocation parameters, allowing workflows to scale up better.

The `Dynamic Scheduling` technique usually gives us similar execution times than the proposed static methods, still performing better than the default `dispel4py` allocation.

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 10.1145/3041036. Both static techniques are applicable across mappings. But, for the `Dynamic Scheduling` technique, we will need to modify it first to adapt it to distributed-memory architectures. This change will require to apply another type of global queue, such as *Apache Kafka*, *RabbitMQ* or *ZeroMQ* messaging frameworks. We plan to work in this technique, so workflows with groupings will be able to be enacted dynamically. Our next immediate step is to evaluate the presented optimizations using the data-intensive dispel4py workflows (from Seismology and Climate scientific communities) developed within the DARE project[14], using the DARE computing infrastructure (Cloud system).

[14]http://project-dare.eu

REFERENCES

[1] F. J. Montáns, F. Chinesta, R. Gómez-Bombarelli, and J. N. Kutz, "Data-driven modeling and learning in science and engineering," *Comptes Rendus Mécanique*, vol. 347, no. 11, pp. 845 – 855, 2019, data-Based Engineering Science and Technology. [Online]. Available: 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, and J. Vetter, "The future of scientific workflows," *The International Journal of High Performance Computing Applications*, vol. 32, no. 1, pp. 159–175, 2018. [Online]. Available: https://doi.org/10.1177/1094342017704893

[3] M. Atkinson, S. Gesing, J. Montagnat, and 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, and 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, and 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, and A. Charalambidis, "Dare: A reflective platform designed to enable agile data-driven research on the cloud," *2019 15th International Conference on eScience (eScience)*, pp. 578–585, 2019.

[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, and 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, and S. Somasekharan, "Multi-objective optimisation in scientific workflow," in *International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland*, ser. Procedia Computer Science, P. Koumoutsakos, M. Lees, V. V. Krzhizhanovskaya, J. J. Dongarra, and P. M. A. Sloot, Eds., vol. 108. Elsevier, 2017, pp. 1443–1452. [Online]. Available: https://doi.org/10.1016/j.procs.2017.05.213

[12] E. N. Alkhanak, S. P. Lee, R. Rezaei, and 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*, vol. 113, pp. 1–26, 2016.

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

[14] R. Prodan and M. Wieczorek, "Bi-criteria scheduling of scientific grid workflows," *IEEE Transactions on Automation Science and Engineering*, vol. 7, no. 2, pp. 364–376, 2009.

[15] L. Ramakrishnan, J. S. Chase, D. Gannon, D. Nurmi, and R. Wolski, "Deadline-sensitive workflow orchestration without explicit resource control," *Journal of Parallel and Distributed Computing*, vol. 71, no. 3, pp. 343–353, 2011.

[16] M. Tanaka and 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.

[17] A. Spark, "Apache spark," *Retrieved January*, vol. 17, p. 2018, 2018.

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

[19] R. Filgueira, A. Krause, A. Spinuso, I. Klampanos, P. Danecek, and M. Atkinson, "Dispel4py: An open-source python library for data-intensive seismology," *EGUGA*, p. 6790, 2015.

[20] H. G. Kim, Y. H. Park, Y. H. Cho, and M. H. Kim, "Time-slide window join over data streams," *Journal of Intelligent Information Systems*, vol. 43, no. 2, pp. 323–347, 2014.