dispel4py: A Python Framework for Data-Intensive Scientific Computing

Rosa Filguiera*, Iraklis Klampanos*, Amrey Krause†, Mario David‡, Alexander Moreno§ and Malcolm Atkinson*

* University of Edinburgh, School of Informatics, Edinburgh EH8 9AB, UK
† University of Edinburgh, EPCC, Edinburgh EH9 3JZ, UK
§ Georgia Institute of Technology, School of Computer Science, Atlanta, Georgia 30332

Abstract—This paper presents dispel4py, a new Python framework for describing abstract stream-based workflows for distributed data-intensive applications. The main aim of dispel4py is to enable scientists to focus on their computation instead of being distracted by details of the computing infrastructure they use. Therefore, special care has been taken to provide dispel4py with the ability to map abstract workflows to different enactment platforms dynamically, at run time. In this work we present four dispel4py mappings: Apache Storm, MPI, multi-threading and sequential. The results show that dispel4py is successful in enacting on different platforms, while also providing scalable performance.

Keywords—Data-intensive computing, e-Infrastructures, data streaming, scientific workflows, programming frameworks, Python.

I. INTRODUCTION

During recent years there has been a widespread and increasing requirement for data-intensive computation in science, engineering, medicine, government and many other fields [1]. In this context, the term “data-intensive” is used to characterise computation that either requires or generates large volumes of data, or has complex data access patterns due to algorithmic or infrastructural reasons.

In most of these examples, the research proceeded by bringing all of the required data into one administrative context and then exploration was conducted by teams that included the domain scientists, data-analysis specialists and computer scientists. Similar arrangements are typical in many other contexts, such as those reported in [2], [3], [4].

Specific to the conduct of modern science is the use of different hardware, software and middleware, often located on machines outside the researcher’s administration vicinity. While this provides researchers with access to more powerful machines, it requires users to use machine-specific software, often use only once. Scientific workflow platforms, such as Taverna [5], MyExperiment [6], KNIME† and ERFlow‡ provide users with tools to organise their experiments graphically, also arranging submission of parts of a solution to remote computing resources. Scientific workflow systems such as the above are described as being coarse-grained, as they are mainly mechanisms to organise a computation regardless of the language it was written in or the resource it is being submitted to. The building blocks of these workflows are therefore seen by the workflow engine as black boxes, with users still having to take care of a number of technical details, such as targeting specific platforms, specifying stores and files, etc.

An alternative, often complementary, approach to file-based coarse-grained workflows is to use stream-based fine-grained workflows. Dispel [7] represents this paradigm, allowing users to define abstract, machine-agnostic, fine-grained data-intensive workflows. Dispel is an imperative language, so a Dispel script (which uses a notation similar to Java) describes how to construct a workflow rather than specifying the workflow itself directly. Dispel workflows are compositions of processing elements representing knowledge discovery activities (e.g. batch database querying, data aggregation) through which significant volumes of data can be streamed in order to manufacture a useful knowledge artefact. Such processing elements may themselves be defined by compositions of other, more fundamental computational elements, in essence having their own internal workflows.

In this work, we present the dispel4py³ library, a Python implementation of the Dispel language. In contrast to Dispel, dispel4py not only constructs and describes abstract workflows and compositions, but it contains executables as Python objects. Using dispel4py, scientists can describe the dataflow and implement processing elements in Python, which is widely supported on many platforms and is popular in many scientific domains, such as in geosciences, biology, astronomy and others.

Furthermore, scientific communities nowadays have the possibility to access a large variety of computing resources for different requirements. For testing scientific workflows sequential and local resources (e.g. a laptop or desktop) are usually selected, however for larger applications, distributed resources (e.g. Apache Storm clusters⁴, MPI [8] powered clusters, and shared-memory multi-core machines) are the most frequent choices. Workflows should support enactment/execu- tion on various computing resources while remaining unchanged. Therefore, dispel4py has been designed to map the workflow descriptions onto arbitrary computational resources dynamically at run time, setting it apart from other stream-based frameworks.

In this work, four mappings are presented: Apache Storm,

DOI 10.1109/DISCS.2014.12

978-1-4799-7038-4/14 $31.00 © 2014 IEEE

1https://www.knime.org/
2http://www.erflow.eu/
3https://github.com/akrause2014/dispel4py
4https://storm.incubator.apache.org/
**MPI, Multiprocessing** (a multi-threading Python package) and *Sequential*. In the future, more mappings will be added to dispel4py, making dispel4py more flexible and broadly applicable.

The rest of the paper is structured as follows: In Section II we discuss related work. In Section III we present core dispel4py concepts. In Section IV we present and discuss the four dispel4py mappings currently available. In Section V we present a realistic workflow borrowed from the field of seismology. Using this workflow as a case-study, we evaluate the dispel4py mappings developed on different platforms. Finally, in Section VI we present our conclusions and provide pointers for future work.

### II. BACKGROUND

There are many scientific workflow systems, including but not limited to Taverna, Kepler [9], Pegasus [10], Triana [11], Swift [12], Trident [13], Meandre [14], and Bobolang [15]. They focus primarily on managing dataflow and computations, and generally use a bottom-up approach to describe experiments. Important defining features include whether they have a GUI, are stream-based or file-based, and whether or not they use an automatic mapping onto different enactment platforms.

Among the most popular of these are Taverna, Kepler, and Pegasus. Taverna is GUI based and mainly used in the field of bioinformatics. It is open source, it is supported by a large community, and many nodes and graphs are freely available to use through the MyExperiment website. The first Taverna version was file-based, while streaming support was added in subsequent versions. It has the advantage that it can be operated via a range of environments, including the Taverna workbench, the command line, a remote execution server, and the online workflow designer OnlineHPC. However, it does not provide automatic mappings to computing mechanisms.

Kepler is a streaming GUI-based system, developed primarily for geosciences, environmental sciences and bioinformatics workflows. Its unique feature is separating the computation model from the workflow structure so that different computation models can fit the same workflow graph. It also does not have automatic mapping.

Pegasus uses Chimera’s virtual data language [16] and is not GUI-based. Instead, it is a file-based framework for mapping complex scientific workflows onto remote systems. Pegasus supports an input DAX workflow description, which can be generated using a Python or Java API, or a script. However, this framework does not translate workflows to different computing platforms automatically.

A newer workflow system is Bobolang[15]. This is implemented as a new language based on C++. Bobolang focuses on automatic parallelisation and it is stream-based, containing many solutions to streaming parallel application problems. It has a useful feature of multiplication of inputs and outputs, meaning that a single node can have as many inputs or outputs as the user requires. Currently, it does not support distributed computing, and it does not provide automatic mapping to different computing platforms.

### III. DISPeler4PY CONCEPTS

As we introduced in Section I, dispel4py is a Python implementation of the Dispel language that allows users to describe their data-intensive applications on an abstract level. Below, we present a summary of the main dispel4py concepts and terminology:

- **A processing element** (PE) is a computational activity that encapsulates an algorithm, a service or another data transformation process. PEs represent the basic computational blocks of any dispel4py workflow, at an abstract level – they are the nodes in a workflow graph.
- **An instance** is the executable copy of a PE that runs in a process. During execution time, each PE is translated into one or more instances.
- **A connection** streams data between PE instances through one of their corresponding input and output interfaces. The rate of data consumption and production depends on the behaviour of the PE and its neighbouring elements.
- **A composite processing element** is a PE that wraps a dispel4py sub-workflow. Composite processing elements allow for synthesis of increasingly complex PEs, based on previously defined constituent parts.
- **A partition** is a number of PEs wrapped together and executed within the same process. This a new feature of dispel4py, which is not included in its previous version, dispel.
- **A graph** defines the ways through which PEs are connected and data can be streamed, i.e. the topology of the workflow. There are no limitations on the type of graphs that can be designed with dispel4py. Figure 1 is an example graph involving four PEs. PE-1 produces random words (output-1) and numbers (output-2) as outputs, and sends each output to a different PE. PE-2 and PE-3 cumulative the number of words, and PE-3 calculate the average of the numbers. The output of those PEs are then merged in PE4, which prints out the number of words and the final average. More graph examples can be found in the dispel4py documentation.
- **A grouping** specifies the communication pattern between PEs. There are different groupings available: For example, group by streams the data that satisfy the same feature to the same instance of a PE; one to all means that all PE instances send copies of their output data to all the connected instances.
- **dispel4py** is dataflow-oriented rather than control-oriented system. As a result, no specification of how data should be produced or consumed is required; instead, data is pushed out of or pulled into PEs based on the balance of their respective implemented behaviours, regulated by the enactment platform.

For constructing dispel4py workflows, users only have to implement their PEs (in Python) and connect them as they

---

Split and Merge is primarily implemented in Java. Once the workflow has been built, it can be executed in a number of parallel environments thanks to the mappings, which are explained in the next section. dispel4py workflows can be executed anywhere Python is available and without any adaptation by users.

Finally, if a dispel4py workflow is executed by using MPI or Multiprocess with several number of processes, the processes are equally distributed among the PEs, except for the first PE, which is assigned by default one process. If users want to change the default topology, they only need to write the following instruction in the script before connecting their PEs in the graphs.

```python
dispel4py.workflow_graph.WorkflowGraph

g1 = WorkflowGraph()
g2 = WorkflowGraph()
g3 = WorkflowGraph()
g4 = WorkflowGraph()

g1.connect(g2, 'output1', g3, 'input1')
g3.connect(g2, 'output2', g4, 'input1')
g2.connect(g3, 'output3', g4, 'input2')
```

Listing 1: An example split-and-merge dispel4py graph.

Once the dispel4py workflow has been built, it can be automatically executed in a number of parallel environments thanks to the mappings, which are explained in the next section. dispel4py workflows can be executed anywhere Python is available and without any adaptation by users.

Finally, if a dispel4py workflow is executed by using MPI or Multiprocess with several number of processes, the processes are equally distributed among the PEs, except for the first PE, which is assigned by default one process. If users want to change the default topology, they only need to write the following instruction in the script before connecting their PEs in the graphs.

```python
<name_of_PE>. numprocesses = Number
```

### IV. DISPEL4PY MAPPINGS

dispel4py’s strength is that it allows the construction of workflows without knowledge of the hardware or middleware context in which they will be executed. Users can then focus on designing their workflows at an abstract level, describing actions, input and output streams, and how they are connected. dispel4py then maps these descriptions to the selected enactment platforms. Since the abstract workflows are independent from the underlying communication mechanism these workflows are portable among different computing resources without any migration cost imposed on users.

dispel4py currently supports Apache Storm, MPI and Multiprocessing mappings for distributed resources, as well as a Sequential mapping to act as our baseline for workflow development and small applications. Descriptions of these mappings follow below:

#### A. Apache Storm

dispel4py was initially designed to translate workflows into and make use of Apache Storm. Apart from the semantic similarities between Dispel’s and Storm’s streaming models, this was also decided because Storm has proven itself in a number of online processing contexts.

Apache Storm (primarily implemented in Java) executes topologies which consume and process streams of data. A Storm topology will typically run until killed and it processes data items as and when they are produced. The system handles load balancing and recovers from failure of worker nodes by restarting them as required. Workers can be added at runtime. However, Storm is generally not used in HPC resources and it requires installation on a dedicated cluster.

dispel4py maps to Storm by translating the graph description to a Storm topology. As dispel4py allows its users to define data types for each PE in a workflow graph, types are deduced and propagated from the data sources throughout the graph when the topology is created. Each Python PE is mapped to either a Storm bolt or spout, depending on whether the PE declares inputs (a bolt) or is a data source (a spout). The data streams in the dispel4py graph are mapped to Storm streams. dispel4py PEs may declare how a data stream is partitioned among the processing instances. By default these instructions map directly to built-in Storm stream groupings. The source code of this mapping can be found at

There are two execution modes for Storm: a topology can be executed in local mode using a multi-threaded framework (development and testing), or it can be submitted to a production cluster. The user chooses the mode when executing a dispel4py graph in Storm. Both modes require the availability of the Storm package on the client machine.

The following commands submit the Split and Merge dispel4py graph (Figure 1) as a Storm topology in local mode and to a remote cluster respectively.

```bash
python -m dispel4py.storm.storm_submission \split_and_merge -m local
```

```bash
python -m dispel4py.storm.storm_submission \xcorr SplitAndMergeTopology -m remote
```

#### B. MPI

MPI is a standard, portable message-passing system for parallel programming, whose goals are high performance, scalability and portability. MPI, as opposed to Storm, is very well known and widely supported in HPC environments.

For this mapping, dispel4py uses mpi4py, which is a full-featured Python bindings for MPI and is based on the standard MPI-2.

dispel4py maps PEs to a collection of MPI processes. Depending on the number of targeted processes, which the
user specifies when executing the mapping, multiple instances of each PE are created to make use of all available processes. PEs at the root of the dispel4py graph only ever execute in one instance so as to prevent the generation of duplicate data blocks.

Streams are converted into generic pickle-based Python objects and transferred by using MPI asynchronous calls. Groupings are mapped to communication patterns, which assign the destination of a stream according to the grouping (e.g., shuffle grouping is mapped to a round-robin pattern, for group-by the hash of the data block determines the destination). The source code of this mapping can be found at.

To use the MPI mapping, users need to install mpi4py, and any MPI interface, like mpich or openmpi.

To execute the Split and Merge dispel4py graph by using MPI mapping, a user would issue the following command:

```
mpiexec -n <number mpi_processes> \
python -m dispel4py.worker_mpi split_and_merge
```

C. Multiprocessing

The Python library multiprocessing is a package that supports spawning subprocesses to leverage multicore shared-memory resources. It is available as part of standard Python distributions on many platforms without further dependencies, and hence is ideal for small jobs on desktop machines.

The Multiprocessing mapping of dispel4py creates a pool of processes and assigns each PE instance to its own process. Messages are passed between PEs using multiprocessing.Queue objects.

As in the MPI mapping, dispel4py maps PEs to a collection of processes. Each PE instance reads from its own private input queue on which its input blocks arrive. Each data block triggers the execution of the process() method which may or may not produce output blocks. Output of a PE is distributed to the connected PEs depending on the grouping pattern that the destination PE has requested. The distribution of data is managed by a Communication class for each connection. The default is ShuffleCommunication which implements a round-robin pattern; the use case below also uses GroupByCommunication which groups the output by certain attributes. The source code of this mapping can be found at.

The Multiprocessing mapping also allows partitioning of the graph to support handling several PEs together in one process. Users can specify partitions of the graph and the mapping distributes these across processes in the same way as single PEs. The following shows the command to execute the Split and Merge dispel4py graph, using Multiprocessing mapping:

```
python -m dispel4py.multi_process \
```

D. Sequential Mode

The sequential mode (Simple process) is a simple standalone tool that is ideal for testing graph execution during development. It allows execution of a dispel4py graph in sequence within a single process, and hence without optimisation.

When executing a dispel4py graph in sequential mode, the dependencies of each PE are determined and the PEs in the graph are executed in a depth-first fashion starting from the roots of the graph (data sources). The source PEs process a number of iterations as specified by the user. All data is processed and messages are passed in-memory by a single process. The source code of this mapping can be found at.

```
python -m dispel4py.simple_process split_and_merge
```

V. Evaluation

dispel4py is currently being developed as part of VERCE project, which aims at developing an e-Science environment to facilitate innovative data analysis and coding methods that fully exploit the wealth of data in global seismology. Several seismic workflows haven been developed with dispel4py in VERCE. The Seismic Ambient Noise Cross-Correlation workflow has been selected as a use case for evaluating the performance of dispel4py with different mappings and computing resources because it represents a data-intensive problem and it is commonly used by seismology researchers.

A. Use Case: Seismic Ambient Noise Cross-Correlation

Seismic Ambient Noise can be used for surface wave tomography, also referred to as ambient noise surface wave tomography [17]. The first two phases described in this paper have been designed and implemented as a workflow in dispel4py. During Phase 1, each continuous time series recorded from a given seismic station (called a “trace”), is subject to a series of treatments. The processing of each trace is independent from any other, making this phase an embarrassingly parallel problem (complexity O(n), where n is the number of stations). Phase 2 consists of pairing all stations and calculating the cross-correlation for each pair (complexity O(n^2), where n is the number of stations).

Raw data is recorded continuously by one or more seismic networks, over typical time periods of 5 to 10 years. Each network is composed of hundreds to thousands of stations. Each station typically records three spatial components at sampling frequencies that range from a few to hundreds of Hz. As such, raw data volumes ranging from a few TBs to hundreds TBs are not uncommon. The main attributes characterising a “trace” are the name of the network, the station, the component or channel, as well as start and end times.

Seismic data are available in several formats, such as the miniSEED [18] or SAC. These data are obtainable through.

```
```

---

public archives, (e.g. EIDA-Orfeus\(^{15}\)). Often in data representation, each file obtained contains the trace corresponding to a single day, coming from a single channel of a single station belonging to a single network. The metadata describing the origin of the trace are recorded in the header of the file.

Fig. 2 shows the complete dispel4py workflow, while Fig. 3 shows the dataflow through the main PEs. The workflow has been subdivided into four partitions. A command-line option specifies whether the graph is run based on a partition per process or, alternatively, as a PE per process.

1) Partition 1:
- PE `findFiles` produces a list of filenames according to specific input configuration parameters, such as a list of stations, time period, etc. This is shown in Fig. 3 as the output of the leftmost box: [NET/STA/CHN1, NET/STA/CHN2, NET/STA/CHN3].

2) Partition 2:
- PE `producer` takes as input a filename from the previously produced list and outputs a data stream (network/station/channel “trace”). The length is determined by the initial raw trace sampling frequency and time period (more details are given below). It extracts the header information of the “trace” and inserts it into JSON format.
- PE `composition` is a sub-workflow (Fig. 2), i.e., where data preparation (Phase 1) takes place. Each of these PEs performs a treatment on the data stream (middle boxes in Fig. 3). PEs may have input parameters given by the seismologists.

3) Partition 3:
- PE `match time window` joins two input data streams, matching seismic traces from different stations by timestamp.
- PE `xcorr` has two input streams corresponding to different stations and the same time period to calculate the cross-correlation between them. The dataflow is shown in the rightmost boxes in Fig. 3, the output result is a list of streams labelled \([n=0, n=1, \ldots]\), for each pair of stations. It is noted that following the cross-correlation computation comes the stacking phase\(^{16}\), which is not currently implemented.

4) Partition 4:
- PE `consumer` writes the result of the cross-correlation to an HDF5 file.

The output of the cross-correlation is written to a file. Each HDF5 “GROUP” contains cross-correlation calculations between a pair of stations. One main characteristic of the framework and the use of HDF5 is that it employs parallel I/O, taking advantage of underlying drivers and libraries.

It should be noted that the design and implementation of the scientific part of the workflow (the PE `composition` and the cross-correlation function) can be easily modified by the seismologists, either by changing the order, or altering/adding PEs, according to their specific analysis requirements. These scientific PEs are simply Python functions.

The scientists who write the PEs do not need to worry about how to parallelise the code or which platform it is going to run on because dispel4py performs the parallelisation automatically. The source code of the PE implementations in the example above has 350 lines while the definition of the graph is only 60 lines long. However, the modularity as well as the abstract level the PEs are typically written in, enhance reusability, allowing scientists to compose new workflow graphs using theirs or other scientists’ previously written and shared PEs.

B. Experiments

Three platforms have been used for our experiments: Terracorrelator, the Open Science Data Cloud sullivan cluster (OSDC sullivan), and the SuperMUC, described below.

Terracorrelator\(^{17}\) is a computing facility for massive real-time data assimilation in environmental sciences located at the University of Edinburgh and funded through a NERC award. The machine consists of four nodes, each having 32 cores. The first two nodes are Dell R910 rack-mounted servers with 4 Intel Xeon E7-4830 8 processors, each with 2TB RAM, 12TB near-line SAS local storage and 8Gbps fibre-channel connectivity to the additional Terracorrelator storage arrays. We used one such 32-core node for our experiments.

\(^{16}\)Stacking is the combination of a collection of seismic traces into single trace

\(^{17}\)http://gtr.rcuk.ac.uk/project/F8C52878-0385-42E1-820D-D0463968B3C0
OSDC sullivan\textsuperscript{18} is an OpenStack cluster with GlusterFS. Each node is an m1.xlarge with 8 VCPUS, 20GB VM disk, and 16GB RAM. For our experiments, four nodes were used, summing up to 32 cores in total.

SuperMUC\textsuperscript{19} is a supercomputer at the Leibniz Supercomputing Centre in Munich, with 155,656 processor cores in 9400 compute nodes. SuperMUC is based on the Intel Xeon architecture consisting of 18 Thin Node Islands and one Fat Node Island. This work has only had access to the Thin Node Islands, which contain Sandy Bridge nodes, each having 16 cores and 32 GB of memory. The total memory is over 300 TB RAM, and it uses infiniband FDR10 interconnect. For comparison reasons, when evaluating \texttt{dispel4py} we only made use of up to two nodes (32 cores in total).

We performed our experiments by using the \texttt{dispel4py} cross-correlation workflow for 90 and 180 days worth of data, for two stations, on all platforms. The input data size of the seismic traces for 180 days is 3.5GB, and the size of the output cross correlation results is 25MB (90 days) and 51MB (180 days). For each number of days (90 or 180), we varied the number of cores from 4 to 32 with a step size of 4 cores, except for \texttt{Simple process} mapping, which uses always on core for running the \texttt{dispel4py} workflows on all platforms. As an experiment we considered a cross-correlation using a fixed number of cores and days. Each experiment was repeated 5 times.

We chose not to use the \texttt{Storm} mapping as part of the experiment presented in this paper in order to focus on HPC-compatible resources, as this is also the priority of our current users. Further, Storm could not be installed on resources such as SuperMUC and therefore comparisons would be superficial. In the future, we plan to evaluate all mappings in a suitable cluster containing all relevant middleware, with the aim of keeping the hardware as static as possible, for experimental purposes.

Figure 4 (Terracorrelator) shows that \texttt{MPI} and \texttt{Multiprocess} mappings scales when the number of cores is increased for both time lengths. As the Terracorrelator is a shared memory machine, the \texttt{Multiprocess} mapping has a slightly better performance than \texttt{MPI}.

Figure 5 (OSDC cloud) shows that the \texttt{Multiprocess} mapping scales for 8 cores, and levels off for more cores, as it is expected, as the machine has only 8 cores per node. However, for the \texttt{MPI} mapping, it scales well for the 180-day calculation, but shows some variation for 90 days, likely to be due to the increase of messages for the same input data.

Figure 6 (SuperMUC) again shows that the \texttt{MPI} and the \texttt{Multiprocess} mappings scale well. We would like to highlight that this platform is very fast, so our mappings perform extremely well. The \texttt{MPI} implementation used in this machine, is optimised to a degree that it performs as fast as the multiprocessing library within shared memory. As the number of cores per node is 16, we only tested the \texttt{Multiprocess} mapping up to that number, because as we have shown in Figure 5, this mapping does not improve performance when there are more processes than cores.

\textsuperscript{18}https://www.opensciencecloud.org/
\textsuperscript{19}http://www.lrz.de/services/compute/supermuc/systemdescription/

Fig. 4: \texttt{dispel4py} Cross-correlation experiments by using Terracorrelator machine.

Note that \texttt{Simple process} is a sequential mapping that keeps all the intermediate data in memory and does not make use of multiple cores, which explains its constant performance regardless of the number of cores allocated. SuperMUC shows better performance than the others for this mapping because of the features of the platform. Due to RAM requirements, we only have 180-day \texttt{Simple process} values on the Terracorrelator, which has 2TB of memory per node.

To demonstrate \texttt{dispel4py}'s power for automatic parallelisation, we calculated the efficiency of the \texttt{MPI} mapping by using equation 1, which is a common method in HPC for measuring the scalability of an application. As shown in Table I, for large datasets the efficiency is at over 70% on all platforms.

Finally, to show the scalability of \texttt{dispel4py} with a large scale of data, and using more than 2 nodes, we performed a variation of the original cross-correlation use case by using 1000 stations (one seismic trace per station) as input data, and by applying two types of methods for pre-processing the data of each station. In this case, each pre-process data station has to be cross-correlated with all the other pre-process data stations. As a results the number of cross-correlations for this use case is \((1000*999)/2 = 499500\) per pre-process type. The total input data size is 3GB, and the size of the output
cross-correlation results is 77GB. The experiments of this use case, were performed in SuperMUC with 16, 32, 64, 128, and 256 cores (which means 1, 2, 4, 8 and 16 nodes), by using MPI mapping. And as the previous use case, each experiment was repeated 5 times. Figure 7 shows the effectiveness on scalability of those experiments, where it is appreciated that the performance of MPI mapping gains by increasing the number of cores and nodes.

\[
\frac{\text{Time}_{\text{32 processes}}}{\text{Time}_{\text{4 processes}} \times \frac{32}{4}}
\]  

(1)

<table>
<thead>
<tr>
<th>Machine</th>
<th>90 days</th>
<th>180 days</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terracorrelator</td>
<td>0.654</td>
<td>0.774</td>
</tr>
<tr>
<td>OSDC</td>
<td>0.437</td>
<td>0.860</td>
</tr>
<tr>
<td>SuperMUC</td>
<td>0.681</td>
<td>0.702</td>
</tr>
</tbody>
</table>

TABLE I: Efficiency values with MPI mapping.

From the results, we conclude that the same dispel4py workflow can be used with different mappings, and the performance of each mapping depends on the features of the machine where the application is run.

VI. CONCLUSIONS AND FUTURE WORK

In this paper we presented dispel4py, a novel Python library for streaming, data-intensive processing. The novelty of dispel4py is that it allows its users to express their compu-
tional need as a fine-grained abstract workflow, taking care of the underlying mappings to suitable resources. Currently, dispel4py is able to translate abstract workflow graphs into Apache Storm, MPI, shared-memory Multiprocessing, as well as executing in sequential mode. The sequential mode makes use of a single core and is used as a baseline and also as a means for users to test their workflows on a development machine before commencing full-scale execution.

We demonstrated the MPI, Multiprocessing and Sequential mappings on a realistic scenario borrowed from the field of seismology. dispel4py achieves scalable performance in both parallel mappings tested, showing good speedups (Section V) on all three machines, each with different architectures. The parallelisation performance achieved, at least for the scenario used, appears to be independent of the hardware architecture. More importantly, dispel4py is easy to use. dispel4py requires very few lines of Python code to define a workflow, while the PEs – also written in Python – can be reused in a well-defined and modular way by different users, in different workflows and executed on different platforms via different mappings. This is of paramount importance due to the increased complexity faced by scientists and other professionals, which is imposed by both algorithmic as well as system/hardware requirements.

Users can also use their familiar development environment, such as IPython, for all aspects of their work with dispel4py. dispel4py is still under development, and we continuously aim to improve it. In the coming months we will add optimisation mechanisms based on a number of features, such as semantic characteristics of PEs, topological features of the workflow graphs, support for PE failures, as well as data and processing properties of architectures we are mapping onto. Additionally, we aim to add more mappings, such as for Apache Spark. 20 Of particular importance will be the splitting and deployment of parts of the workflow graph onto different architectures, such as GPUs. Lastly, we aim to further deploy and evaluate dispel4py in different scientific and commercial domains. It is an open source project and we would welcome assistance with its development.

VII. ACKNOWLEDGMENT

This research has been supported by VERCE project (funded by EU FP7 RI 283543), Terracorrelator project (funded by NERC NE/L012979/1), and by grants from Gordon and Betty Moore Foundation and the National Science Foundation and major contributions from OCC members like the University of Chicago. This work also used of the Open Science Data Cloud (OSDC) which is an Open Cloud Consortium (OCC)-sponsored project.

REFERENCES


20https://spark.apache.org/