# dispel4py: An Agile Framework for Data-Intensive eScience

Rosa Filgueira[*], Amrey Krause[†],
Malcolm Atkinson[*], Iraklis Klampanos[*],
Alessandro Spinuso[‡], Susana Sanchez-Exposito[§]
[*] *School of Informatics, University of Edinburgh, Edinburgh EH9 3JZ, UK*
[†] *EPCC, University of Edinburgh, Edinburgh EH9 3JZ, UK*
[‡]*KNMI, Koninklijk Nederlands Meteorologisch Instituut, De Bilt 3731 GA, NL*
[§]*Instituto de Astrofísica de Andalucía (CSIC) , Granada E-1800, Spain*

***Abstract*—We present dispel4py a versatile data-intensive kit presented as a standard Python library. It empowers scientists to experiment and test ideas using their familiar rapid-prototyping environment. It delivers mappings to diverse computing infrastructures, including cloud technologies, HPC architectures and specialised data-intensive machines, to move seamlessly into production with large-scale data loads. The mappings are fully automated, so that the encoded data analyses and data handling are completely unchanged. The underpinning model is lightweight composition of fine-grained operations on data, coupled together by data streams that use the lowest cost technology available. These fine-grained workflows are locally interpreted during development and mapped to multiple nodes and systems such as MPI and Storm for production.**

**We explain why such an approach is becoming more essential in order that data-driven research can innovate rapidly and exploit the growing wealth of data while adapting to current technical trends. We show how *provenance* management is provided to improve understanding and reproducibility, and how a *registry* supports consistency and sharing. Three application domains are reported and measurements on multiple infrastructures show the optimisations achieved. Finally we present the next steps to achieve scalability and performance.**

***Keywords*-eSciences workflows, data intensive application, run time analysis, distributed systems, python frameworks.**

## I. Introduction

Recent years have seen a spectacular growth in scientific data, that must be shared, processed and managed on different distributed computational infrastructures (DCI). Major contributors to this phenomenal data deluge include new avenues of research and experiments facilitated by e-Science, which enables global inter-disciplinary collaborations. These share resources to solve the new problems of science, engineering and humanities. Such e-Science data emanates from different areas, such as sensors, satellites, high-performance computer simulations and already exceeds tens of petabytes per year. A rate that will increase significantly over the next decade. Performance of applications using these data is sensitive to data movement operations between distributed resources, such as large transfers data between the storage and compute systems, communication among workflow components executed on different DCI, intensive internal communication in parallel applications, or data streaming between locations. On the other hand, scientific communities (*e.g.* seismologists or astronomers) already access a wide variety of computing resources, and have computational problems that need HPC architectures. However, success with these technologies depends on additional mechanisms that are not straightforward for non-experts: for example MPI or OpenMP) should be used depending on the memory architecture. This technical detail distracts from the domain goals and limits progress, *e.g.* by requiring code changes for each target DCI .

To empower domain scientists to invent and improve their data-science methods they need to be able to work independently of the details of target infrastructures, to experiment freely in their development environment and to move to production scales fluently. They need to be able to collaborate, sharing both data and scientific methods with each other. Providing scientific communities with easy-to-use tools to support such activities is vital for eScience.

This paper presents a new tool kit for scientists, called dispel4py, to enable them to rapidly prototype their distributed data-intensive applications. It provides an enactment engine that maps and deploys abstract workflows onto multiple parallel platforms, including Apache Storm, MPI and shared-memory multi-core architectures. dispel4py comes with basic data *provenance* functionality allowing for monitoring, as well as with an information *registry* that is accessed transparently to achieve collaboration between users and consistency between runs.

e-Science applications which depend on data-intensive computing already benefit from dispel4py, when consistency between runs, execution on multiple platforms, as well as sharing, collaboration and replicability of results are required. However, to increase the scalability of applications we are designing and prototyping a run-time adaptive compression mechanism. dispel4py has been primarily used in e-Science contexts, most notably in *Seismology*. Recently it has been used in another two domains: *Astrophysics* and *Social Computing*.

This paper is structured as follows. Section II presents background. Section III defines the dispel4py concepts.

Section IV discusses dispel4py features. Section V presents three eScience domains, *Seismology*, *Astrophysics* and *Social Computing*, with dispel4py workflows. Section VI introduces the design for the dispel4py compression strategy. We conclude with a summary of achievements and outline some future work.

## II. Background and related work

There are many scientific workflow systems, including Taverna, Kepler [1], Pegasus [2], Triana [3], Swift [4], Trident [5], Meandre [6], and Bobolang [7]. 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-* or *file-based*, and whether or not they use an automatic mapping onto different enactment platforms. In most cases, they reflect underlying technical details; consequently it is less easy for data-scientists to take full responsibility for their methods.

Among the most popular of these are Taverna, Kepler, and Pegasus. Taverna offers graphical interfaces, well-developed catalogues of composable services particularly for the life sciences. It is open source and is supported by a large community. Workflow sharing is arranged via myExperiment [8]. 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.

Kepler is a streaming GUI-based system, developed primarily for geosciences, environmental sciences and bioinformatics workflows. It separates the computation model from the workflow structure so that different computation models can fit the same workflow graph.

Pegasus uses Chimera's virtual data language [9] 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. Even with its Wings front end[1], a semantically rich workflow system used to create and validate workflows, it does not support immediate experiment or users working undistracted by technical detail.

Bobolang, a relative new workflow system based on data streaming, is linguistically based on C++ and focuses on automatic parallelisation [7]. It supports multiple inputs and outputs, meaning that a single node can have as many inputs or outputs as a user requires. Currently, it does not support automatic mapping to different DCIs.

Mechanisms to improve sharing and reuse of workflows have proved important in the task-oriented context, *e.g.* myExperiment [8] and Wf4Ever [10]. It is unclear whether these will need extension to accommodate data-streaming workflows [11], as the scale of data handled precludes simple bundling of data with workflows to ensure reproducibility as in computationally intensive workflows [12].

## III. dispel4py concepts

The dispel4py data-streaming workflow library is part of a greater vision for the future of formalising and automating both data-intensive and computation-intensive scientific methods. We posit that workflow tools and languages will play increasingly significant roles due to their inherent modularity, user-friendly properties, accommodation of the full range from rapid prototyping to optimised production, and intuitive visualisation properties – graphs appear to be a natural way to visualise logical compositions of processing steps. We are just experiencing a critical transition from an era when performance issues dominated to an era when these can be dealt with automatically so that domain scientists can experiment with and take full responsibility for the encoding of their methods. We adopt the abstract model of data streaming between operations as it is versatile and easily understood. It will have a significant impact on the way scientists think and carry out their data-analysis tasks. The low overhead of the interconnections make it suitable for immediate interpretation and for composing small as will as large steps, yet data streams can be expanded to arbitrary capacity and the graphs are easily parallelised. Consequently, dispel4py allows scientists to express their requirements in abstractions closer to their needs and further from implementation and infrastructural details.

The Dispel language [13] had these goals. dispel4py builds on this but aligns more closely with the requirements of scientists and infrastructure providers.

We now present a summary of the main dispel4py concepts and terminology:

- A processing element (PE) is a computational activity, corresponding to a step in a scientific method or a data-transforming operator. They encapsulate an algorithm or a service. PEs are the basic computational blocks of any dispel4py workflow at an abstract level. They are instantiated as nodes in a workflow graph. dispel4py offers a variety of PEs: GenericPE , IterativePE, ConsumerPE, SimpleFunctionPE and create_iterative_chain. The primary differences between them, are the number of inputs that they accept and how users express their computational activities. For example, GenericPE accepts *n* number of inputs and outputs, while IterativePE declares only one input and one output and ConsumerPE has one input and no outputs. More information is available at[2].
- An instance is the executable copy of a PE that will consume data units from its input ports and emit data units from its output ports transformed by its algorithm.
- A connection streams data units from an output of a PE instance to one or more input ports of other instances.

[1]Wings: wings-workflows.org

[2]http://dispel4py.org/processing_elements.html#pe-base-classes

The rate of data consumption and production depends on the behaviour of the source and destination PEs.

- A composite processing element is a PE that wraps a dispel4py sub-workflow. Composite processing elements allow for synthesis of increasingly complex workflows by composing previously defined sub-workflows.
- A partition is a number of PEs wrapped together and executed within the same process. It is used to explicitly co-locate PEs that have relatively low CPU and RAM demands, but high data-flows between them.
- A graph defines the ways in which PEs are connected and hence the paths taken by data, *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. FilterTweet receives the stream tweet and filter_Tweet takes them as inputs. It emits the tweets' hash_tags and language as outputs, which are sent to different PEs: CounterHashTag andCounterLanguage, which count the number of different hash_tags, and languages respectively. The outputs of those PEs are then merged in StatisticsPE, which displays which are the top five most popular hash_tags and whichlanguage is most used. More examples can be found in the dispel4py documentation[3].
- A grouping specifies for an input connection the communication pattern between PEs. There are four patterns available: shuffle, group_by, one_to_all and all_to_one. Each pattern arranges that there are 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, so that an instance receives all of the data units with a particular value. 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. In the mining_tweets dispel4py workflow, CounterHashTag and CounterLanguage apply a grouping_by based upon the field hash_tag and language respectively, and Statistics applies an all_to_one grouping.

To construct dispel4py workflows, users only have to use available PEs from the dispel4py libraries and registry, or implement PEs (in Python) if they require new ones. They connect them as they desire in graphs. We show the code for creating the mining_tweets dispel4py workflow represented in Figure 1. In the code listing below, we assume that the logic within PEs is already implemented.

Listing 1. An example Mining_Tweets dispel4py graph.

```
from dispel4py.workflow_graph import WorkflowGraph

pe1 = FilterTweet()
pe2 = CounterHashTag()
pe3 = CounterLanguage()
pe4 = Statistics()

graph = WorkflowGraph()
graph.connect(pe1,'hash_tag',pe2,'input')
graph.connect(pe1,'language,pe3,'input')
graph.connect(pe2,'hash_tag_count',pe4,'input1')
graph.connect(pe3,'language_count',pe4,'input2')
```

[3]http://dispel4py.org/api/dispel4py.examples.graph_testing.html

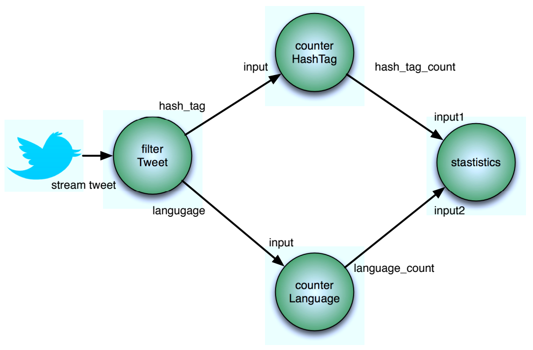


Figure 1. mining_tweets dispel4py workflow.

## IV. DISPEL4PY FEATURES

We will describe three of dispel4py's features that allow users to automatically run their workflows with different enactment systems (*mappings*), to monitor dataflow (*provenance*) at run-time, and to share and re-use their workflows via the (*registry*).

### A. Mappings

One of dispel4py's strengths is that it allows the construction of workflows without knowledge of the hardware or middleware context in which they will be executed. Users focus on designing their workflows at a logical level, describing actions, input and output streams, and how they are connected. When their Python script is run this graph is constructed, and then it is either locally interpreted or the dispel4py system maps the graph to a selected enactment platform. Since the abstract workflows are independent from the underlying communication mechanism they are portable among different computing resources without any migration cost imposed on users, i.e. they do not make any changes to run in a different context.

The dispel4py system currently implements mappings for Apache Storm, MPI and Multiprocessing DCIs, as well as a Sequential mapping for development and small applications. Descriptions of these mappings follow.

*1) Apache Storm:* The dispel4py system maps to Storm by translating its 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 has inputs (a bolt), i.e. is an internal stage, or is a data source (a spout), i.e. is a point where data flows into the graph from external sources. The data streams in the `dispel4py` graph are mapped to Storm streams. The `dispel4py` PEs may declare how a data stream is partitioned across processing instances. By default these instructions map directly to built-in Storm stream groupings. The source code of all mappings can be found at[4].

There are two execution modes for Storm: a topology can be executed locally 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 Storm package on the client machine.

*2) MPI:* MPI is a standard, portable message-passing system for parallel programming, whose goals are high performance, scalability and portably [14]. For this mapping, `dispel4py` uses mpi4py[5], which is a full-featured Python binding for MPI based on the MPI-2 standard. The `dispel4py` system 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. Input PEs, i.e. at the root of the `dispel4py` graph, only ever execute in one instance to avoid the generation of duplicate data blocks.

Data units to be shipped along streams are converted into pickle-based Python objects and transferred using MPI asynchronous calls. Groupings are mapped to communication patterns, which assign the destination of a stream (*e.g.* shuffle grouping is mapped to a round-robin pattern, for group-by the hash of the data block determines the destination). The MPI mapping requires mpi4py and any MPI interface, such as mpich[6] or openmpi[7].

*3) 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` (also called multi) 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 queues 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 from a PE is distributed to the connected PEs depending on the grouping pattern that the destination PE requests. The Communication class manages distribution of data. The default is ShuffleCommunication which implements a round-robin pattern; the example below uses GroupByCommunication to group output by specified attributes. The Multiprocessing mapping allows partitioning of the graph to colocate PEs in one process. Users can specify partitions of the graph and the mapping distributes these across processes in the same way as single PEs.

*4) Sequential mode:* The sequential mode (simple) is a standalone mode that is ideal for testing workflows during development. It executes a `dispel4py` graph in sequence within a single process 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 user-specified number of iterations. All data is processed and messages are passed in-memory.

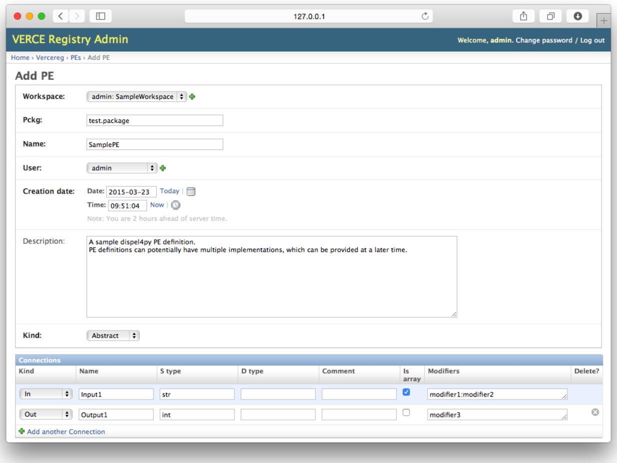
### *B. Registry*

The modularity of workflow approaches, and in particular the data-centric, fine-grained design of `dispel4py`, leads to extensive re-usability. In `dispel4py` users can store, share and reuse workflow entities, such as PEs and functions, via the `dispel4py` information registry. The registry implements a data schema able to describe `dispel4py` entities in a form usable by the `dispel4py` enactment engine. By using the registry, researchers can import their own or third-party PE specifications, functions for inclusion in their workflows.

To facilitate collaboration, the `dispel4py` registry introduces *workspaces*, essentially containers of workflow components, which are owned and managed by users and user groups, and which can be isolated or shared, selectively or wholly, at will. The registry adopts a packaging system for grouping semantically related workflow components, in line with `dispel4py`. Workflow components, such as PEs, are then uniquely identifiable by their workspace, package and name. Workspaces are designed as inheritable entities, such that a workspace can be created by inheriting or cloning another workspace, if permissions allow. Users can store new components by interacting with the registry. Once set-up, `dispel4py` is able to make use of components and code stored in the registry transparently by overriding the standard Python import keyword: the specified component is fetched from the registry, if it is not present locally.

A prototype version of the information registry has been developed and has been released as an open-source project[8]. This is implemented in Django[9], a Python-based Web framework, and has a RESTful API. Figure 2 shows the administrator UI for adding PE specifications via a web

[4]https://github.com/dispel4py/dispel4py/
[5]http://mpi4py.scipy.org/
[6]http://www.mpich.org/
[7]http://www.open-mpi.org/
[8]https://github.com/iaklampanos/dj-vercereg
[9]https://www.djangoproject.com


127.0.0.1

VERCE Registry Admin

Welcome, admin. Change password / Log out

Home › Vercereg › PEs › Add PE

Add PE

| | |
|---|---|
| Workspace: | admin: SampleWorkspace |
| Pckg: | test.package |
| Name: | SamplePE |
| User: | admin |
| Creation date: | Date: 2015-03-23 Today<br>Time: 09:51:04 Now<br>Note: You are 2 hours ahead of server time. |
| Description: | A sample dispel4py PE definition.<br>PE definitions can potentially have multiple implementations, which can be provided at a later time. |
| Kind: | Abstract |

Connections

| Kind | Name | S type | D type | Comment | Is array | Modifiers | Delete? |
|---|---|---|---|---|---|---|---|
| In | Input1 | str | | | ✓ | modifier1:modifier2 | |
| Out | Output1 | int | | | ☐ | modifier3 | |

Add another Connection

Figure 2. The administrator interface for the Django-built registry service. Users can use this to make PE specifications available in `dispel4py`.

browser, and as the registry service is implemented as a Web-based API, more user-friendly UIs can be implemented. The registry backend is provided by a relational MySQL database server. The adoption of Django and MySQL, allows for fast prototyping, while achieving a performant solution, deployable on thoroughly tested and reliable software, such as the Apache Web Server. While the use of a relational database fulfils the basic registration requirements of `dispel4py`, we envisage extensions that make better use of the intrinsic semantic relations between workflow components, types, provenance information, etc., either as part of the core implementations or as extensions that use external resources.

The current implementation allows for basic user and group management, which in turn allows for the creation and modification of workspaces. It also covers all core `dispel4py` entities, such as PEs, functions, literals and implementations. It does not yet contain resource descriptions for middleware and hardware, nor does it contain descriptions of data resources and products. Based on the requirements of our current work, we intend to look into these two extensions in the future, also taking into account provenance, described in the following section, and optimisation considerations.

## C. *Provenance Management*

As for other workflow systems, `dispel4py` provides support for the production and collection of provenance information. The provenance management consists of a comprehensive system which includes: extensible mechanisms for provenance production, a web API and a visualisation tool. The system is used in production in the field of solid-Earth sciences. The design of the provenance system considered the following requirements:

- *Flexible metadata and Selectiveness*: Users are considered as part of the archival process of their computations. They have the best knowledge of the data produced and therefore they need to be supported with tools to describe it according to community and custom terminology and interest [15].
- *Runtime Analysis*: The rapid availability of provenance data may allow for the detection of scenarios that can't be foreseen when developing a scientific workflow. For instance, errors, unexpected metadata values or excessive execution time.
- *Ease of Access*: Streaming workflows can produce huge quantities of provenance data with high I/O rates. As Pauw observed [16], reducing the risk of technical and cognitive overload is essential in order to use provenance effectively. Interactive consumption by users is the primary concern.

One of our challenges was to inject the functionalities needed to support the *Flexible metadata and Selectiveness* with minimal impact on the core `dispel4py` API. We consider provenance as a modular and dynamically pluggable enrichment of the bare bone system, leaving a PE developer to make the best use of it, with minimal effort when she wants to. We take into account the potentially high I/O rate of a streaming computation by having mechanisms for a selective provenance activation, which may select, for instance, a subgraph of a workflow or a section of a stream, to achieve focused validation, also in real-time, with a significantly reduced overhead [17]. Our approach makes a clear distinction between three important phases: the provenance production phase, its collection and eventually, its storage. In Figure 3, we show how the instances of a *GenericPE*, or its specialisation, organised in a `dispel4py` subgraph, can be transformed to include in their base types also the *ProvenancePE* type, which then allows them to produce provenance data. We achieve this injection by adopting dynamic polymorphism techniques, to assure that provenance operations can be activated at runtime. This extends the PE implementation at runtime, in such a way that all of the logic needed to deal with extensive metadata management and dependency tracking. The enhanced PEs are connected to a *ProvenanceRecorderPE*, which have the specific role of collecting and possibly analysing provenance data. The implementation of the *ProvenanceRecorderPEs* may change from case to case, in order to support a variety of scenarios. For instance, a specific type of recorder could comply to the requirements of the hosting DCI, which may impose specific protocols on transfer operations to external facilities, such as an external provenance store.

The recorders can be used for the *Runtime Analysis* of the incoming lineage traces. This could have immediate effects on the life-cycle of the workflow, besides triggering specific actions on the intermediate data products of a PE. For instance, rapid stage-out operations towards external systems which might provide dedicated hardware, could better serve tasks for data management, visualisation, etc. Usually, such

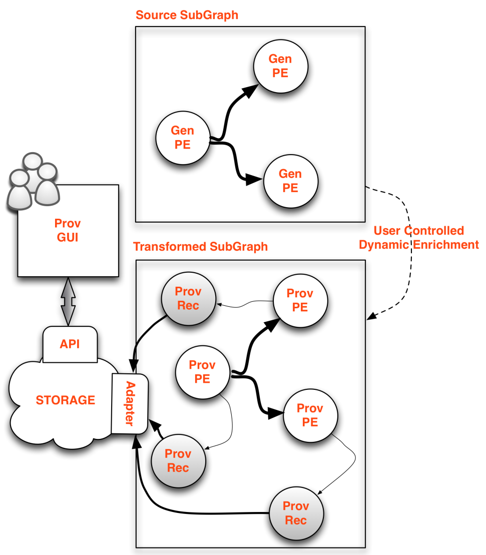


Figure 3. Provenance Injection: The *GenericPE* of a `dispel4py` subgraph, are extended at runtime assuming the *ProvenancePE* type. These are connected to instances of *ProvenanceRecorderPE* which reads, analyse and activates the transfer of the provenance traces towards a centralised catalogue, external to the DCI.

scenarios would require the implementation of *ad hoc* PEs. Instead we include a number of common behaviours and practices within our provenance data model to ensure these are properly handled by the system automatically.

Substantial effort has been dedicated to *Ease of access*. In Figure 4 we show a screenshot of the *Provenance Explorer* GUI that allows different operations including: workflow progress monitoring, searches over metadata, data dependency navigation, data preview and download. The latter can be achieved in interactive mode, via download links, and in batch mode, thanks to the automatic generation of bulk download scripts. This batch download feature is actively used to combine search results with large download operations, which are most likely to be performed on institutional or campus clusters, which are better sized for massive post processing tasks, rather then a user's laptop. The GUI is served by a web API which exposes a storage backend developed with MongoDB [10]. The choice of this NoSQL technology has been motivated by the nature of the data produced, which besides having an interconnected structure also presents a very large quantity of metadata terms which need to be flexible and efficiently indexed. After some investigation of possible solutions MongoDB proved the most reliable production quality system with support for all of the access scenarios available through our Provenance API specification. Last but not the least, we include in the concept of *Ease of access* also the notion of portability. Our API is capable of exporting the trace of a run in the W3C-PROV JSON representation[11], to facilitate interoperability with third-party tools and, its adoption by institutional archives, for long-term data curation and preservation.

[10]http://www.mongodb.org

[11]http://www.w3.org/Submission/2013/SUBM-prov-json-20130424/

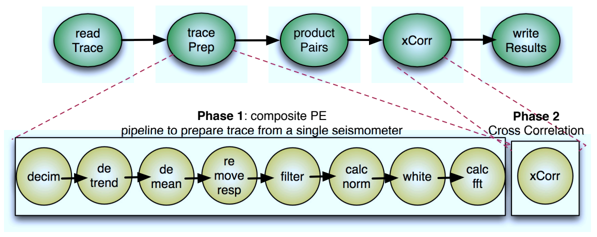


Figure 5. A simplified abstract workflow for seismic cross-correlation using `dispel4py`

## V. DISPEL4PY WORKFLOWS

The following subsections describe `dispel4py` workflows, which are examples from three domains: Seismology, Astrophysics and Social Computing. Using these we will show how `dispel4py` enables scientists to describe data-intensive applications using a familiar notation, and to execute them in a scalable manner on a variety of platforms without modifying their code.

### A. Seismology: Seismic Noise Cross Correlation

Earthquakes and volcanic eruptions are sometimes preceded or accompanied by changes in the geophysical properties of the Earth, such as wave velocities or event rates. The development of reliable risk assessment methods for these hazards requires real-time analysis of seismic data and truly prospective forecasting and testing to reduce bias. However, potential techniques, including seismic interferometry and earthquake "repeater" analysis, require a large number of waveform cross-correlations, which is computationally intensive, and is particularly challenging in real-time. With `dispel4py` we have developed the *Seismic Ambient Noise Cross-Correlation* workflow (also called the xcorr workflow) as part of the VERCE[12] project [18], which preprocesses and cross-correlates traces from several stations in real-time. The xcorr workflow consists of two main phases:

- *Phase 1– Preprocess*: Each continuous time series from a given seismic station (called a *trace*), is subject to a series of treatments. The processing of each trace is independent from other traces, making this phase "embarrassingly" parallel (complexity $O(n)$, where $n$ is the number of stations).
- *Phase 2 – Cross-Correlation*: Pairs all of the stations and calculates the cross-correlation for each pair (complexity $O(n^2)$).

Figure 5 shows the `dispel4py` xcorr workflow, which has five PEs. Note that the tracePrep PE is a compositePE, where data preparation takes place. Each of those PEs, from decim to calc_fft, performs processing on the data stream. The xcorr workflow was initially tested on a local machine using a small number of stations as input data. Later, it was executed and evaluated on different parallel platforms

[12]http://www.verce.eu

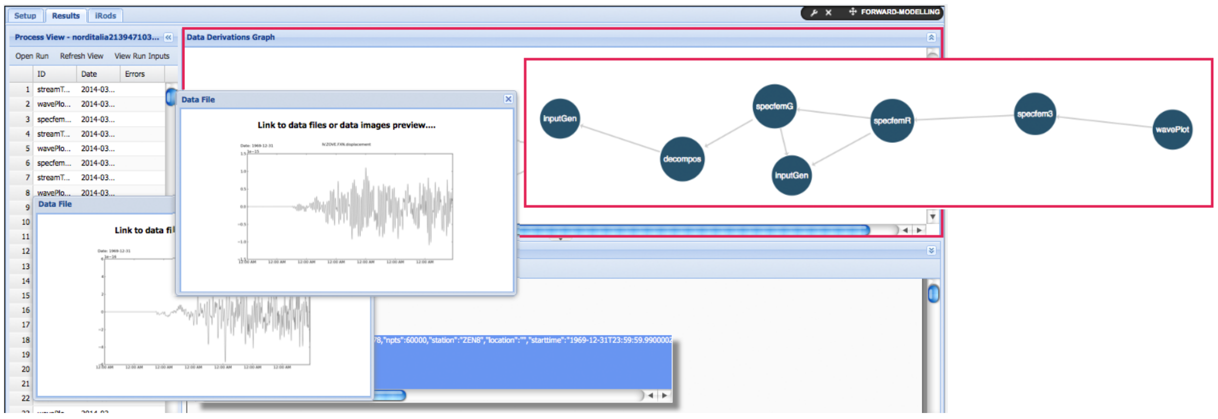


Figure 4. Provenance Explorer GUI.

(described at Section V-D and summarised in Table IV) automatically, scaling up by using using the parallel mappings of `dispel4py` to 1000 stations as input data (150 MB) performing 499,500 cross-correlations (39GB) without modifying the code.

Table I
MEASURES (SECONDS) FOR 1000 STATIONS ON FOUR DCIS WITH THE MAXIMUM NUMBER OF CORES AVAILABLE

| Mode | Terracorrelator | Amazon EC2 | EDIM1 | SuperMuc |
|---|---|---|---|---|
| MPI | 3066.22 | 16862.73 | 38656.94 | 1093.16 |
| multi | 3143.77 | | | |
| Storm | | 27898.89 | 120077.123 | |

The results (see Table I) demonstrate that `dispel4py` can be applied to diverse DCI targets and adapt to variations among them. However, the xcorr performance depends on the DCI selected. For example, the best results in terms of performance were achieved on the SuperMuc machine with MPI followed by the Terracorrelator machine with MPI and multi mappings. The Storm mapping proved to be the least suitable in this case. Yet it is the best mapping in terms of fault-tolerance for any case and DCI, as Storm delivers automatic fault recovery and reliability. It may be those features that make it the slowest mapping. See [19] for further measurements.

### B. *Astrophysics: Internal Extinction of Galaxies*

The Virtual Observatory (VO) is a network of tools and services implementing the standards published by the International Virtual Observatory Alliance (IVOA)[13] to provide transparent access to multiple archives of astronomical data. VO services are used in Astronomy for data sharing and serve as the main data access point for astronomical workflows in many cases. This is the case of the workflow presented here, which calculates the *Internal Extinction of the Galaxies* from the AMIGA catalogue[14]. This property represents the dust extinction within the galaxies and is a correction coefficient needed to calculate the optical luminosity of a galaxy. The implementation of this workflow (also called int_ext workflow) with `dispel4py` shows that it can use VO services, as a first step to supporting more complex workflows in this field.

Figure 6 shows the `dispel4py` int_ext workflow with four PEs. The first PE reads an input file (*coordinates.txt* size 19KB) that contains the right ascension (Ra) and declination (Dec) values for 1051 galaxies. The second PE queries a VO service for each galaxy in the input file using the Ra and Dec values. The results of these queries are filtered by filterColumns PE, which selects only the values that correspond to the morphological type (Mtype) and the apparent flattening (logr25) features of the galaxies. Their internal extinction is calculated by the internalExtinction PE.

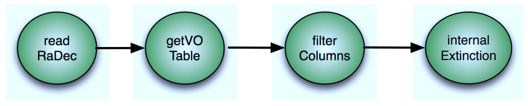


Figure 6. Workflow for calculating the internal extinction of galaxies using `dispel4py`

The int_ext workflow has also been implemented using Taverna, which, as well as the VO service query and the python script, includes two services belonging to the Astrotaverna plugin [20][15] to manage the data format of VOTables.

Initially, we compared the time-to-complete for both implementations (Taverna vs `dispel4py`), performing a set of executions using the same desktop machine, with 4 GB RAM, an Intel Core 2 Duo 2.66GHz processor, and ensuring the same network conditions, checking that the VO service is running, and executing them in the same time slot. The number of jobs executed concurrently was set to 10 in the Taverna workflow, since after several tests, we verified this was its optimal configuration. `dispel4py` was configured

[13]http://www.ivoa.es
[14]http://amiga.iaa.es
[15]http://myexperiment.org/workflows/2920.html

Table II
Elapsed times (seconds) for 1050 galaxies on two DCIs with the maximum number of cores available

| Mode | Terracorrelator | EDIM1 |
|---|---|---|
| MPI | 31.60 | 96.12 |
| multi | 14.50 | 101.2 |
| Storm | | 30.2 |

to use the multi mapping and 4 processes. The difference in elapsed time is considerable. While the Taverna execution takes approximately 9.5 minutes, dispel4py takes around 100 seconds.

Further evaluations were performed using the Terracorrelator and EDIM1 DCIs, and the results show (see Table II) that the lowest elapsed time for this workflow was achieved using the multi mapping in Terracorrelator, followed by Storm on EDIM1. For those evaluations we used the maximum number of cores available in each DCI (see Table IV). The performance differences for the MPI and multi mappings between Terracorrelator and EDIM are due to the number and features of the cores available in each DCI. We used 14 cores for the MPI mapping and 4 cores for the multi mapping on EDIM1, and 32 cores for both mappings on the Terracorrelator.

## C. Social Computing: Twitter Sentiment Analysis

With over 500 million tweets per day[16] and 240 million active users who post opinions about people, events, products or services, *Twitter* has become an interesting resource for sentiment analysis [21]. In this case study, we investigate the benefits of dispel4py for analysing *Twitter* data by implementing a basic *Sentiment Analysis* workflow, called sentiment. dispel4py has been specifically designed for implementing streaming and data-intensive workflows, which fits the data stream model followed by *Twitter*. The aim of sentiment analysis (also referred to as opinion mining) is to determine the attitude of the author with respect to the subject of the text, which is typically quantified in a polarity: positive, negative or neutral.

We focus on how dispel4py manages the frequency of tweets for performing two basic sentiment analyses by using the lexical approach and by applying the AFINN and SentiWordNet lexicons [22]. AFINN [23] is a dataset of 2477 English words and phrases that are frequently used in microbbloging services and each word is associated with an integer between minus five (negative) to plus five (positive). SentiWordNet (SWN3) [24][17] is a fine-grained, exhaustive lexicon with 155,287 English words and 117,659 synsets, which are sets of synonyms that represent cognitive synonyms. It distinguishes between nouns, verbs, adjectives and adverbs, and each synset is automatically annotated according to positivity, negativity and neutrality.
The original code used for building the sentiment workflow can be found at[18]. Figure 7 shows the sentiment workflow, which first scans the tweets preprocessing the words they contain, and then classifies each tweet based on the total counts for positive and negative words. As the sentiment workflow applies two analyses, different preprocessing and classification tasks need to be performed. To classify each tweet with the AFINN lexicon (see Phase 1a in Figure 7), the sentimentAFINN PE tokenises each tweet "text" word, and then a very rudimentary sentiment score for the tweet is calculated by adding the score of each word. After determining the sentiments of all tweets, they are sent to the findState PE, which searches the US state from which the tweet originated, and discards tweets which are not sent from the US. The HappyState PE applies a grouping by based on the state and aggregates the sentiment scores of tweets from the same state, which are sent to the top3Happiest PE. This PE applies all-to-one grouping and determines which are the top three happiest states.

[16]http://www.internetlivestats.com/twitter-statistics/
[17]http://sentiwordnet.isti.cnr.it/

Table III
Measures (seconds) for 126,826 tweets on three DCIs with the maximum number of cores available

| Mode | Terracorrelator | EDIM1 | SuperMUC |
|---|---|---|---|
| MPI | 925.04 | 32396.04 | 2057.59 |
| multi | 971.04 | | 3954.63 |

The sentiment workflow also calculates tweet sentiment in parallel using the SWN3 lexicon. The tokenizationWD PE is a composite PE, where tweet tokenisation and tagging takes place (see Phase 1b in Figure 7): the tokenTweet PE splits the tweet text into tokens, the POSTagged PE produces a part-of-speech (POS) tag as an annotation based on the role of each word (*e.g.* noun, verb, adverb) and the wordnetDef PE determines the meaning of each word by selecting the synset that best represents the word in its context. After pre-processing each tweet, the second phase of the analysis is performed by the sentiment SWN3 composite PE (see Phase 2b in Figure 7): the SWN3 Interpretation PE searches the sentiment score associated with each synset in the SWN3 lexicon, the sentimentOrientation PE gets the positives, negatives and average scores of each term found in a tweet and the classifySWN3Tweet PE determines the sentiment of the tweet. After the classification, the same procedure as before is applied to each tweet, to know which are the three happiest states.

With dispel4py we could run the sentiment workflow for computing the top three "happiest" US states and the sentiment analysis results as the tweets are produced. However, to study the performance of the workflow under different DCIs and mappings, 126,826 tweets (500MB) were downloaded into a file, and used as a test set for all the experiments. For this reason, the first PE, readTweets PE, reads each tweet from the file, and streams them out.

The results shown in Table III demonstrate that

[18]https://github.com/linkTDP/BigDataAnalysis_TweetSentiment

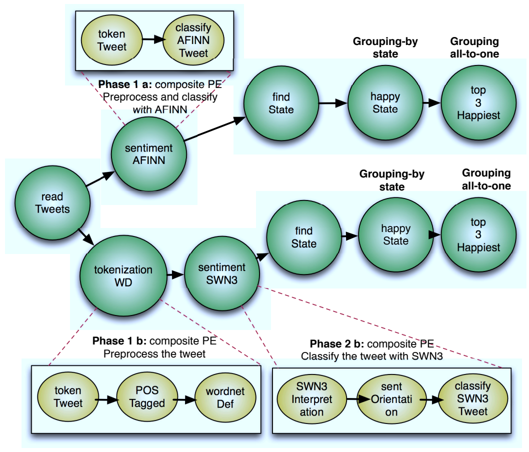


Figure 7. Workflow for calculating *Twitter* sentiment analysis and the top five happiest U.S. states using `dispel4py`.

`dispel4py` is able to map to diverse DCIs and enactment systems, adapting automatically to their variations, and show that the performance depends on the DCI selected and that the same use case can run faster or slower depending on the selection of DCI and `dispel4py` mapping made. In future work, we will automate those choices, enabling `dispel4py` to choose the best DCI and mapping.

### *D. Evaluation Platforms: DCI features*

Four platforms have been used for our experiments: Terracorrelator, the Super-MUC cluster (LRZ), Amazon EC2, and the Edinburgh Data-Intensive Machine (EDIM1). These are described below and summarised in Table IV.

The Terracorrelator[19] is configured for massive data ingest in the environmental sciences at the University of Edinburgh. The machine has four nodes, each with 32 cores. Two nodes are Dell R910 servers with 4 Intel Xeon E7-4830 8 processors, each with 2TB RAM, 12TB SAS storage and 8Gbps fibre-channel to storage arrays. We used one 32-core node for our measurements.

Super-MUC[20] is a supercomputer at the Leibniz Supercomputing Centre (LRZ) in Munich, with 155,656 processor cores in 9,400 nodes. Super-MUC is based on the Intel Xeon architecture consisting of 18 Thin Node Islands and one Fat Node Island. We used 16 Thin (Sandy Bridge) Nodes, each with 16 cores and 32 GB of memory, for the measurements.

On the Amazon EC2 the Storm deployment used an 18-worker node setup. We chose Amazon's T2.medium instances[21], provisioned with 2 vCPUs and 4GB of RAM. Amazon instances are built on Intel Xeon processors operating at 2.5GHz, with Turbo up to 3.3GHz. We used 18 VMs for our measurements.

EDIM1[22] is an Open Nebula[23] linux cloud designed for data-intensive workloads. Backend nodes use mini ITX motherboards with low powered Intel Atom processors with plenty of space for hard disks. Each VM in our cluster had 4 virtual cores – using the processor's hyperthreading mode, 3GB of RAM and 2.1TB of disk space on 3 local disks. We used 14 VMs for our evaluations.

Table IV
DCI FEATURES

| Load | Terracorrelator | Super-MUC | Amazon EC2 | EDIM1 |
|---|---|---|---|---|
| DCI type | shared-memory | cluster | cloud | cloud |
| Enactment systems | MPI, multi | MPI, multi | MPI, Storm, multi | MPI, Storm, multi |
| Nodes | 1 | 16 | 18 | 14 |
| Cores per Node | 32 | 16 | 2 | 4 |
| Total Cores | 32 | 256 | 36 | 14 |
| Memory | 2TB | 32GB | 4GB | 3GB |
| Workfklows | xcorr, int_ext, sentiment | xcorr, senti-ment | xcorr | xcorr, int_ext, senti-ment |

## VI. Run-time stream compression

A workflow's performance can suffer from large data transfers between storage systems (*e.g.* databases or file systems) and/or DCI(s) (*e.g.* cloud, cluster, grid) resources. Therefore, we propose a run-time adaptive compression strategy in `dispel4py`, which we expect to reduce data-transfer bottleneck and improve the scalability of the workflows. The compression strategy will be transparent for users and it will be placed in two levels:

- between two connected PEs' instances, which are located in different nodes from the same or different DCI's resources.
- between PEs and storage systems.

When the workflow is run using multiprocessing or sequential mapping, we will not apply compression between the connected PEs, with the exception of the first and the last ones, because the data is transferred via memory and not via a network. Therefore, compression between PEs will be considered only when MPI and Storm mappings have been selected.

Instead of passing data directly to an I/O system or between two connected PE instances located in different computing resources, it will first be intercepted by the compression strategy which, if considered beneficial, losslessly compresses the data with LZ4 [25] algorithm. We have selected this algorithm, because we have performed previously [26] an exhaustive empirical study with synthetic[24] and real files[25]. The studies show that the shortest compression and decompression times are achieved by *LZ4*, *Snappy* [27], *LZO* [28], *RLE* [29], *Huffman* [30].

[19]http://gtr.rcuk.ac.uk/project/F8C52878-0385-42E1-820D-D0463968B3C0
[20]http://www.lrz.de/services/compute/supermuc/systemdescription/
[21]http://aws.amazon.com/ec2/instance-types/
[22]https://www.wiki.ed.ac.uk/display/DIRC
[23]http://opennebula.org
[24]http://effort.is.ed.ac.uk/Compression/SyntheticResults.htm
[25]http://effort.is.ed.ac.uk/Compression/CompressionFiles.pdf

Our design for compression consists of three steps:

- *Deciding when to compress*: The first $x$ iterations of the workflow will be used for transferring the data without compression (odd iterations) and with compression (even iterations). The strategy will record the execution times for each PE for each iteration and the compression ratios for the iterations when compression is applied. At the end of the $x$ iterations, the strategy will have enough information to decide which data streams should be compressed. The compression decisions, will be stored in the registry with relevant information about the workflow: DCI, mapping and parameters used.
- *Applying the compression decisions*: For the rest of the workflow's iterations the compressions chosen will be applied.
- *Re-evaluating the compression decision*: Every $n$[26] iterations of the workflow, the decisions will be re-evaluated to check whether the network conditions or data features have changed, updating the compression decision in the registry.

The compression decisions applied to each PE will be completely independent from each other. However, they will be dependent of the mapping and DCI selected for running the workflow. It could happen that different decisions are taken when the workflow is executed using the same mapping on different DCIs. The compression strategy will check at the beginning of a workflow's execution whether there are compression decisions stored in the registry for that particular workflow, mapping, DCI and input parameters. In which case, those decisions will be applied initially (avoiding the first strategy's step), and at the $n$ iteration, they will be re-evaluated, updating the decisions if needed.

## VII. 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 computational need as a fine-grained abstract workflow, taking care of the underlying mappings to suitable resources. Scientists can use it to develop their scientific methods and applications on their laptop and then run them at scale on a wide range of e-Infrastructures without making changes.

We demonstrate with three realistic scenarios borrowed from the field of seismology, astrophysics and social computation, that `dispel4py` can be used to design and formalise scientific methods (the PE compositions and their connectivity). `dispel4py` is easy to use, it requires very few lines of Python code to define a workflow, while the PEs can be re-used in a well-defined and modular way by different users, in different workflows and executed on different platforms via different mappings.

To help make data-intensive methods more reproducible and open, `dispel4py` provides a registry and provenance mechanisms. By using the registry, users store their workflows (and their components) with their relevant annotations, for further runs and/or for sharing them with others. The provenance mechanism allows users to analyse at runtime the provenance information collected and it offers combined operations to access and download data, which may be selectively stored at runtime, into dedicated data archives. Moreover it foster for a rapid diagnostic of logical errors and failures thanks to a flexible metadata structure and errors capturing.

In the near future we will add optimisation mechanisms based on a number of features, such as implementing the run-time compression strategy, developing additional diagnostic tools that can select the best target DCIs and enactment modes automatically. Additionally, we aim to add more mappings, such as for *Apache Spark*[27].

## Acknowledgment

This research was supported by the VERCE project (EU FP7 RI 283543) and the Terracorrelator project (funded by NERC NE/L012979/1).

## References

[1] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock, "Kepler: an extensible system for design and execution of scientific workflows," in *Proceedings of 16th International Conference on Scientific and Statistical Database Management*, June 2004, pp. 423–424.

[2] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. C. Laity, J. C. Jacob, and D. S. Katz, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.

[3] D. Churches and *etal.*, "Programming scientific and distributed workflow with Triana services," *Conc.&Comp.: P&E*, vol. 18, no. 10, pp. 1021–1037, 2006.

[4] M. Wilde and *etal.*, "Swift: A language for distributed parallel scripting," *Parallel Comp.*, vol. 37, no. 9, pp. 633 – 652, 2011.

[5] Y. Simmhan and *etal.*, "Building the Trident Scientific Workflow Workbench for Data Management in the Cloud," in *ADVCOMP*. IEEE, October 2009.

[6] X. Llorá, B. Ács, L. S. Auvil, B. Capitanu, M. E. Welge, and D. E. Goldberg, "Meandre: Semantic-Driven Data-Intensive Flows in the Clouds," in *IEEE Fourth International Conference on eScience*. IEEE Press, 2008, pp. 238–245.

[7] Z. Falt and *etal.*, "Bobolang: A Language for Parallel Streaming Applications," in *Proc. HPDC*. ACM, 2014, pp. 311–314.

[26] The best values for $x$ and $n$ will be determined after implementation

[27] https://spark.apache.org/

[8] D. De Roure and *etal.*, "The Design and Realisation of the myExperiment Virtual Research Environment for Social Sharing of Workflows," *FGCS*, vol. 25, pp. 561–567, 2009.

[9] P. Fraternali and S. Paraboschi, "Chimera: A language for designing rule applications." in *Active Rules in Database Systems*, 1999, pp. 309–322. [Online]. Available: http://dblp.uni-trier.de/db/books/collections/patton99.html#FraternaliP99

[10] K. Belhajjame and *etal.*, "A Suite of Ontologies for Preserving Workflow-Centric Research Objects," *J. Web Semantics*, in press 2015.

[11] L. Lefort and *etal.*, "W3C Incubator Group Report – review of Sensor and Observation ontologies," W3C, Tech. Rep., 2010.

[12] D. Rogers and *etal.*, "Bundle and Pool Architecture for Multi-Language, Robust, Scalable Workflow Executions," *J. Grid Comput.*, vol. 11, no. 3, pp. 457–480, 2013.

[13] M. P. Atkinson, C. S. Liew, M. Galea, P. Martin, A. Krause, A. Mouat, Ó. Corcho, and D. Snelling, "Data-intensive architecture for scientific knowledge discovery," *Distributed and Parallel Databases*, vol. 30, no. 5-6, pp. 307–324, 2012. [Online]. Available: http://dx.doi.org/10.1007/s10619-012-7105-3

[14] MPI Forum, "MPI: A message-passing interface standard," *IJ of Supercomputer Applications*, vol. 8, pp. 165–414, 1994.

[15] A. Misra, A. Misra, M. Blount, M. Blount, A. Kementsietsidis, A. Kementsietsidis, D. Sow, D. Sow, M. Wang, and M. Wang, "Advances and Challenges for Scalable Provenance in Stream Processing Systems," *Provenance and Annotation of Data and Processes*, vol. 01, pp. 253–265, 2008. [Online]. Available: http://www.springerlink.com/index/w5865j066612l840.pdf

[16] W. D. Pauw, M. Letja, B. Gedik, and H. Andrade, "Visual debugging for stream processing applications," *Runtime Verification*, pp. 18–35, 2010. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-16612-9_3

[17] A. Spinuso and *etal.*, "Provenance for seismological processing pipelines in a distributed streaming workflow," in *Proc. EDBT '13*. ACM, 2013, pp. 307–312.

[18] M. Atkinson, M. C. S. Claus, R. Filgueira, and et al., "Verce delivers a productive e-science environment for seismology research," in *IEEE International eScience Conference*, 2015.

[19] R. Filguiera, I. Klampanos, A. Krause, M. David, A. Moreno, and M. Atkinson, "Dispel4py: A python framework for data-intensive scientific computing," in *Proceedings of the 2014 International Workshop on Data Intensive Scalable Computing Systems*, ser. DISCS '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 9–16. [Online]. Available: http://dx.doi.org/10.1109/DISCS.2014.12

[20] J. Ruiz, J. Garrido, J. Santander-Vela, S. Sánchez-Expósito, and L. Verdes-Montenegro, "Astrotaverna - building workflows with virtual observatory services," *Astronomy and Computing*, vol. 7–8, no. 0, pp. 3–11, 2014, special Issue on The Virtual Observatory: I.

[21] A. Pak and P. Paroubek, "Twitter as a corpus for sentiment analysis and opinion mining," in *Proceedings of the International Conference on Language Resources and Evaluation, LREC 2010, 17-23 May 2010, Valletta, Malta*, N. Calzolari, K. Choukri, B. Maegaard, J. Mariani, J. Odijk, S. Piperidis, M. Rosner, and D. Tapias, Eds. European Language Resources Association, 2010. [Online]. Available: http://www.lrec-conf.org/proceedings/lrec2010/summaries/385.html

[22] H. Cho, J. Lee, and S. Kim, "Enhancing lexicon-based review classification by merging and revising sentiment dictionaries," in *Sixth International Joint Conference on Natural Language Processing, IJCNLP 2013, Nagoya, Japan, October 14-18, 2013*. Asian Federation of Natural Language Processing / ACL, 2013, pp. 463–470. [Online]. Available: http://aclweb.org/anthology/I/I13/I13-1053.pdf

[23] F. Å. Nielsen, "Afinn," Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, mar 2011.

[24] S. Baccianella, A. Esuli, and F. Sebastiani, "Sentiwordnet 3.0: An enhanced lexical resource for sentiment analysis and opinion mining," in *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)*, N. C. C. Chair), K. Choukri, B. Maegaard, J. Mariani, J. Odijk, S. Piperidis, M. Rosner, and D. Tapias, Eds. Valletta, Malta: European Language Resources Association (ELRA), may 2010.

[25] "Real time data compress," 2012, "http://fastcompression.blogspot.co.uk/p/lz4.html".

[26] R. Filgueira, M. P. Atkinson, Y. Tanimura, and I. Kojima, "Applying selectively parallel I/O compression to parallel storage systems," in *Euro-Par 2014 Parallel Processing - 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings*, ser. Lecture Notes in Computer Science, F. M. A. Silva, I. de Castro Dutra, and V. S. Costa, Eds., vol. 8632. Springer, 2014, pp. 282–293. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-09873-9_24

[27] J. Russell and R. Cohn, *Snappy*. Book on Demand, 2012. [Online]. Available: http://books.google.co.uk/books?id=PXajMQEACAAJ

[28] Markus Franz Xaver Johannes Oberhumer, "LZO," 2002, http://www.oberhumer.com/opensource/lzo/lzodoc.php.

[29] R. Zigon, "Run length encoding," *Dr. Dobb's Journal of Software Tools*, vol. 14, no. 2, Feb. 1989.

[30] D. E. Knuth, "Dynamic huffman coding," *J. Algorithms*, vol. 6, no. 2, pp. 163–180, 1985.