dispel4py: A Python Framework for Data-Intensive Scientific Computing

Rosa Filgueira, Iraklis Klampanos, Amrey Krause, Mario David, Alexander Moreno and Malcolm Atkinson
University of Edinburgh, Institute of Physique du Globle de Paris, Georgia Institute of Technology
Roadmap

- Introduction
- dispel4py Features
- dispel4py Concepts
- Objective- Portable workflows
- dispel4py mappings
- Evaluation
- Conclusions and future work
Introduction

+ Scientific workflows → necessary tool for many scientific communities:
  + Enabling easy composition and execution of applications in computing resources
  + Scientists can focus on their research and not computation management

+ Scientific communities have access large variety of computing resources: local resources and parallel resources → different enactment systems
  + Workflows should support execution on various resources without changing them.
  + dispel4py designed to map abstract workflows onto different enactment systems dynamically
Introduction- Workflow life-cycle

+ **Workflow composition** — construct a high level workflow known as an *abstract workflow*. Templates or *graphs*

+ **Resource mapping** — maps from the *abstract workflow* to the underlying resources. As a result → *concrete workflow*

+ **Execution & monitoring** — enacts the mapped workflow in the execution environment and monitors the performance of workflow execution

+ **Provenance** — records the history of the data or workflow creation
Introduction-Workflow life-cycle

Focus

Build time
- Workflow Composition
  - abstract workflow
  - mapping
  - optimising (resource allocation)

Run time
- Execution & Monitoring
  - concrete workflow
  - scheduling
  - optimising (execution)

Workflow libraries
Component libraries
Data catalogue
Provenance catalogue
Composition provenance
Provenance record
Mapping provenance
Execution provenance

Jobs
Execution status
Results

Compute
Data
Web services

adapted from Deelman et al. 2009
dispel4py Features

+ **Stream-based**
  + Tasks are connected by streams and not by writing and reading from intermediate files
  + Multiple streams in & out
  + Optimization based on avoiding IO

+ **Data-flow**: Data are moving and transformed between tasks. Dependency among the tasks represents the flows of data.

+ **Fine-grained**: Users write abstract workflow. dispel4py also provides mappings to create different concrete workflows

+ **Python** language for describing tasks and connections
dispel4Py concepts - Processing element

+ PEs represent the basic computational blocks of any dispel4Py workflow: algorithm, service, data transformation

+ Shared: Storing them into the registry

+ Reusable: To be recombined in many different applications

+ General PE features
  + Consumes any number and types of input streams
  + Produce any number and types of output streams
  + It does some processing → computational activity
dispel4Py concepts - Instance and graph

+ **Graph** (*Abstract* level)
  + Topology of the workflow: connections among PES

+ **Instance** (*Concrete* level)
  + Executable copy of a PE that runs in a process.
  + Each PE is translated into one or more instances in run-time

+ Example of graphs
  - Pipeline
  - Split & Merge
  - Tree
dispel4py concepts – Composite PE and partition

+ Composite PE (Abstract level)
  + Sub-workflow in a PE
  + Hides the complexity of an underlying process
  + Treated like any other PE

+ Partition (Abstract level)
  + PEs wrapped together
  + Run several PEs in a single process

+ Example of Composite PE
  ![Composite PE Diagram]

+ Example of Partition
  ![Partition Diagram]
**dispel4py concepts - Groupings**

- Specifies the communication pattern among PEs instances at *Concrete* level
- **Types**
  - **Group_by**: instances that satisfy same feature are sent to the same PE instance (map-reduce)
  - **One_to_all**: all PE instances send data to all the connected PE instances
  - **Global**: all PE instances send data to 1 connected PE instance
dispelp4y concept- Split & Merge example

Users only have to implement: PEs Connections

from dispel4py.workflow_graph import WorkflowGraph
pe1 = WordNumber()
pe2 = CountWord()
pe3 = Average()
pe4 = Reduce()
graph = WorkflowGraph()
graph.connect (pe1, 'output1', pe2, 'input')
graph.connect (pe1, 'output2', pe3, 'input')
graph.connect (pe2, 'output', pe4, 'input1')
graph.connect (pe3, 'output', pe4, 'input2')
Objective– Portable workflows

- Portable workflows among parallel resources
  - Users develop and test workflows in their local machines
  - Submit their workflows to a parallel resource.
  - dispel4py:
    - Automatic and efficient parallelization
    - Without any cost for users
Objective—Portable workflows

Build time

Users write

Selection of the mapping

abstract workflow + mapping

dispel4py workflow

dispel4py mappings

Storm, MPI, Multiprocessing, Sequential

Execution time

Execution of the workflow

concrete workflow executing in the selected resource

Mapping at real time → automatic parallelization in real time
dispel4py mappings - Overview

- **Sequential**
  - Sequential mapping for local testing
  - Ideal for local resources: Laptops and Desktops

- **Multiprocessing**
  - Python’s multiprocessing library
  - Ideal for shared memory resources

- **MPI**
  - Distributed Memory, message-passing parallel programming model
  - Ideal for HPC clusters

- **STORM**
  - Distributed Real-Time computation System
  - Fault-tolerant and scalable
About Storm
- It executes topologies which consumes and process streams
- It runs the topologies until they are killed

How we do the mapping?
- `dispel4py graph → Storm topology`
- Data types are deduced ad propagated from the source
- Each PE → spout (data source) or a bolt (consume data)
- `dispel4py streams → Storm Streams`
- `dispel4py groupings → Storm groupings`

Two executions modes for running Storm topologies:
- Local-mode (desktop) using multi-thread framework → developing and testing
- Production cluster where Storm is installed.
About MPI
- Standard, portable message-passing system for parallel programming
- MPI goals: HPC, Scalability and portably

How we do the mapping?
- dispel4py uses mpi4py - Python bindings for MPI, based in MPI-2
- Each PE → collection of MPI processes:
  - Depending the number of processes → multiple instances of each PE are created
  - The PE root → only executed in 1 instance → 1 MPI process
- dispel4py streams are
  - converted into pickle-based Python objects
  - Transferred by using MPI asynchronous calls
- dispel4py groupings → Communication pattern:
  - e.g. Shuffle grouping → round-robin pattern
dispel4py mappings – Multiprocessing

- About Python Multiprocessing library
  - Package that support spawning sub-processes in shared-memory resources
  - Available as part of standard Python distribution

- How we do the mapping?
  - Each PE → Pool of processes
    - Depending the number of processes → multiple instances of each PE are created
    - The PE root → only executed in 1 instance → 1 process
    - Each PE reads from its own private input queue
  - dispel4py streams are:
    - converted into pickle-based Python objects
    - Transferred by using multiprocessing.Queue objects
  - dispel4py groupings → Communication patterns
    - Shuffle grouping → round-robin pattern
dispel4py mappings – Sequential

+ About sequential mode – Simple_process
  + Standalone tool that is ideal for testing
  + Executes a dispel4py graph in sequence within a single process

+ How we do the mapping?
  + Graph executed in a depth-first fashion starting from the root
  + Streams are passed in-memory by a single process
Evaluation

- What we want to evaluate?
  - Mappings' performance and scalability by using different types of parallel resources
  - Since dispel4py has been used recently by seismologists (VERCE project), we have selected a seismological application as use case

- Storm was not included in those evaluations
  - We could not installed in all the platforms
  - We focus in HPC environments
**Evaluation - Use case**

- **Seismic ambient noise cross-correlation** → data intensive problem and it is commonly used in seismology research
  - Phase 1: Time series data (traces) from seismic stations and days are preprocessed in parallel
  - Phase 2: Preprocessed data are matched by time and then cross-correlated between them
  - Number of cross-correlations = $n*(n-1)/2$, where $n ==$ number of stations

Input data size ~90 days 1.75GB and 3.5GB
Output data size ~ 25MB and 51MB
Evaluation - Platforms

- **Terracorrelator (TC) machine**
  - Massive real data assimilation, at the University of Edinburgh
  - 4 nodes, 32 core per node, 2TB RAM, 12TB storage, 8Gps fiber-channel
  - We used 1 node, 32 cores

- **Open Science Data Cloud sullivan cluster (OSDC)**
  - OpenStack cluster with GlusterFS
  - Each node: **8VCPUS**, 20GB VM disk and 16GB RAM.
  - We used 4 nodes, 32 cores in total

- **SuperMUC cluster**
  - Supercomputer at the Leibniz Supercomputing center in Munich
  - 9400 nodes, and 1556,656 cores
  - We used Thin Node Islands: 16 nodes, each node **16 cores** (256 cores in total) and 32 GB of Ram

Note: Multiprocessing → **32 process**

Note: Multiprocessing → **8 process**

Note: Multiprocessing → **16 process**
Evaluation – Experiments in TC & OSDC

- MPI and Multiprocessing scales well when number of cores is increased
- TC shared memory → Multiprocessing is slightly better
- Multiprocessing scales until 8 processes
  - 1 node → 8 processes
- MPI scales well
Evaluation – Experiments in SuperMUC

MPI and Multiprocessing scales well when number of cores is increased

Multiprocessing testing only until 16 proc.

1000 stations, 2 types of pre-process, 77GB as output data, 256 cores (16 nodes)

Effectiveness on scalability with MPI mapping

1 node → 16 processes
Evaluation – MPI mapping efficiency

Efficiency = \frac{\text{Time}_{32\text{ processes}}}{\text{(Time}_{4\text{ processes}} \times \frac{32}{4})}

<table>
<thead>
<tr>
<th>Machine</th>
<th>90 days</th>
<th>180 days</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terracorrelator</td>
<td>0.67</td>
<td>0.77</td>
</tr>
<tr>
<td>OSDC</td>
<td>0.43</td>
<td>0.86</td>
</tr>
<tr>
<td>SuperMUC</td>
<td>0.68</td>
<td>0.70</td>
</tr>
</tbody>
</table>

Efficiency for long datasets (180 days) is at over 70% on all platforms.
Conclusions and future Work

- **dispel4py** is a novel Python library for **streaming and data-intensive processing**

- **Novelty**
  - Users express their computational activities
  - Dispel4py takes care of the underlying mappings
  - Same workflow can be automatically executed in several parallel systems: MPI, STORM, Multiprocessing
  - Sequential mode can be used for testing workflows before starting full-scale execution
  - dispel4py is easy to use, allows users to share and re-use PEs

- **Demonstration**
  - Dispel4py achieves scalable performance in both mappings tested under parallel platforms

- **Future**
  - Support for PE failures
  - New mappings: Apache Spark
  - Dispel4py automatic monitoring performance framework
  - Dispel4py workflows onto different architectures like GPUs
Thanks & Questions

- **Documentation**
  - [http://www2.epcc.ed.ac.uk/~amrey/VERCE/Dispel4Py/](http://www2.epcc.ed.ac.uk/~amrey/VERCE/Dispel4Py/)

- **Source code**
  - [https://github.com/akrause2014/dispel4py](https://github.com/akrause2014/dispel4py)

- **Contact emails**
  - Rosa Filgueira: rosa.filgueira@ed.ac.uk
  - Amy Krause: akrause@staffmail.ed.ac.uk
  - Alexander Moreno: alexander.f.moreno@gatech.edu
VERCE

- The VERCE project provide a framework to the seismological community to exploit the increasingly large volume of seismological data (e.g. 100 TB of raw data from Tōhoku earthquake).
- Support data-intensive applications
- E-Science Gateway for submitting applications
- Applications → dispel4py workflows
- Distributed and diversified data sources
- Distributed HPC resources on Grid, Cloud and HPC clusters