THE UNIVERSITY *of* EDINBURGH

Optimisation of the enactment of fine-grained distributed data-intensive workflows

Chee Sun Liew

Doctor of Philosophy

Centre for Intelligent Systems and their Applications
School of Informatics
University of Edinburgh

2012

Abstract

The emergence of data-intensive science as the fourth science paradigm has posed a data deluge challenge for enacting scientific workflows. The scientific community is facing an imminent flood of data from the next generation of experiments and simulations, besides dealing with the heterogeneity and complexity of data, applications and execution environments. New scientific workflows involve execution on distributed and heterogeneous computing resources across organisational and geographical boundaries, processing gigabytes of live data streams and petabytes of archived and simulation data, in various formats and from multiple sources. Managing the enactment of such workflows not only requires larger storage space and faster machines, but the capability to support scalability and diversity of the users, applications, data, computing resources and the enactment technologies.

We argue that the enactment process can be made efficient using optimisation techniques in an appropriate architecture. This architecture should support the creation of diversified applications and their enactment on diversified execution environments, with a standard interface, i.e. a workflow language. The workflow language should be both human readable and suitable for communication between the enactment environments. The data-streaming model central to this architecture provides a scalable approach to large-scale data exploitation. Data-flow between computational elements in the scientific workflow is implemented as streams. To cope with the exploratory nature of scientific workflows, the architecture should support fast workflow prototyping, and the re-use of workflows and workflow components. Above all, the enactment process should be easily repeated and automated.

In this thesis, we present a candidate *data-intensive architecture* that includes an intermediate workflow language, named DISPEL. We create a new fine-grained measurement framework to capture performance-related data during enactments, and design a performance database to organise them systematically. We propose a new enactment strategy to demonstrate that optimisation of data-streaming workflows can be automated by exploiting performance data gathered during previous enactments.

Acknowledgements

I would like to express my sincere gratitude to my supervisors: Malcolm Atkinson, Murray Cole and Jano van Hemert, for their immense efforts spent in guiding this research. This thesis would not have been produced without their invaluable advice, excellent knowledge, unceasing support and enormous patience.

I have been fortunate to work with a group of remarkable people in the Edinburgh Data-Intensive Research Group: David , Michelle, Paul, Rosa, Donald, Gary, Ole, Fan, Paolo, Alessandro, Luca, Adam, and those who have left: Liangxiu, Gagarine, Jos, Rob, Yin, and Kostas. Thank you for sharing your thoughts, experiences and pints. You are wonderful people and I hope that we can continue to work and booze together.

I would like to acknowledge the collaborators of the ADMIRE project: Amrey Krause, Alastair Hume, Radosław Ostrowski, Adrain Mouat, Mark Parsons and Rob Baxter (EPCC), Dave Snelling (Fujitsu Laboratories of Europe), Oscar Corcho and Carlos Buil Aranda (Universidad Politécnica de Madrid), Peter Brezany and his team from University of Vienna, Ladislav Hluchý and his team from Slovenská Akadémia VIED, and Maciej Jarka (Comarch SA, Poland). We have done an outstanding work!

In the course of the research, Malcolm has used his widely-spread human network to helped me linking up with the world-class leading researchers; having the opportunity to visit and learn from them is an honour. Ewa Deelman, Ann Chervenak, Robert Grossman, Yunhong Gu, Daniel S. Katz, Michael Wilde, Bernie Ács, Xavier Llorá, Loretta Auvil: thank you for the hospitality and stimulating discussions. I'm looking forward to collaborate with you in the near future. I would also like to acknowledge Martin Kersten, Shantenu Jha, Mirron Livny, and the instructors and friends from the International Grid Computing Summer School 2009, for the motivating and inspiring moments talking to you all.

I would not forget to thank my funding agencies, Malaysia Ministry of Higher Education and University of Malaya, for their generosity; School of Informatics for providing such a great research environment; and my family and friends, for their moral support.

Last but not least, to an amazing lady who has accompanied me throughout this gruelling journey, shared every single laughter and tears, left me no worry so that I can have full concentration on my work, thank you my love, Lai Pin.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Chee Sun Liew*)

Publications

Some of the work in this thesis has been published in journals, conference proceedings and as technical reports. Below are the articles published during the course of this research, my contributions in producing them and where they are being used in this thesis (shown in parenthesis):

- Chee Sun Liew, Amrey Krause, and David Snelling. *Dispel enactment*, in Malcolm P. Atkinson *et al.*, *The DATA Bonanza – Improving Knowledge Discovery for Science, Engineering and Business*, Wiley, to be published in 2012.
  Contributions: Involved in the design of the architecture and language, and the design the measurement framework, the performance database and the mapping algorithms (Chapter 1, 3 & 4).

- Malcolm P. Atkinson, Chee Sun Liew, Michelle Galea, Paul Martin, Amrey Krause, Adrian Mouat, Oscar Corcho, and David Snelling. *Data-intensive architecture for scientific knowledge discovery*, Distributed and Parallel Databases, 30(5), 2012, pp. 307–324.
  Contributions: Involved in the design of the architecture and language, and the architecture evaluation (Chapter 1, 3, & 4).

- Chee Sun Liew, Malcolm P. Atkinson, Radoslaw Ostrowski, Murray Cole, Jano I. van Hemert and Liangxiu Han. *Performance database: capturing data for optimizing distributed streaming workflows*, Philosophical Transactions of the Royal Society A, 369 (1949), 2011, pp. 3268–3284.
  Contributions: Main author (Chapter 3, 4 & 5).

- Liangxiu Han, Chee Sun Liew, Malcolm P. Atkinson, and Jano I. van Hemert. *A generic parallel processing model for facilitating data mining and integration*, Parallel Computing, 37 (3), 2011, pp. 157–171.
  Contributions: Co-designed the pipeline streaming model and the parallel processing model, implemented the prototype, and conducted the evaluation (Chapter 3, 4 & 5).

- Gagarine Yaikhom, Chee Sun Liew, Liangxiu Han, Jano van Hemert, Malcolm Atkinson, and Amy Krause. *Federated enactment of workflow patterns*, in Euro-Par 2010 - Parallel Processing, P. D'Ambra, M. Guarracino, and D. Talia, eds., vol. 6271 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2010, pp. 317–328.
  Contributions: Involved in the design of the language, built the prototype and conducted the evaluation (Chapter 5).

- Chee Sun Liew, Malcolm P. Atkinson, Jano I. van Hemert, and Liangxiu Han. *Towards optimising distributed data streaming graphs using parallel streams*, in HPDC '10: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, S. Hariri and K. Keahey, eds., New York, NY, USA, 2010, ACM, pp. 725–736.
  Contributions: Main author (Chapter 3, 4 & 5).

- Malcolm P. Atkinson, Jano I. van Hemert, Liangxiu Han, Ally Hume, and Chee Sun Liew. *A distributed architecture for data mining and integration*, in DADC '09: Proceedings of the second international workshop on Data-aware distributed computing, ACM, 2009, pp. 11–20.
  Contributions: Built prototype and conducted the evaluation (Chapter 1 & 4).

To my beloved parents
and
my beautiful wife.

Table of Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Modern scientific collaborations have opened up the opportunity of solving complex problems that involve multidisciplinary expertise and large-scale computational experiments. These experiments comprise a sequence of processing steps, or referred as tasks, that need to be executed on selected computing platforms. Each task takes input data, either from preceding tasks or data sources, performs predefined computations, and produces output data for the succeeding tasks or to be delivered to data storage. Tasks are usually separate instances of executable programs that need to be run in a predefined order. A common strategy to make the experiments more manageable is to model the processing steps as *workflows*, and use a workflow management system (WMS) to organise the enactment [80]. A wide range of WMSs have been developed over the decades. Good reviews regarding these systems can be found in [43, 51, 184].

Workflows can be expressed in *standard workflow languages*, e.g. BPEL [155] and YAWL [170] (and the extended version, newYAWL [147]), *systems specific workflow languages*, e.g. SCUFL (Taverna) [138], ZigZag (Meandre) [123] and Swift parallel language (Swift) [178], *graph-based representations*: a general Directed Acyclic Graph (DAG) with the tasks as the nodes (vertices) and data dependencies as the edges (arcs), or *simple text scripts* that define the execution sequence of tasks, as used in DAGman [42]. Some WMSs provide workbenches for composing workflows, while others may take a workflow script and compile it into an internal representation for further processing. This workflow representation defines the logical sequence in which the tasks should be executed (their dependencies) and it is usually referred as an *abstract workflow*. The WMSs then find the appropriate resources and organise the enactment, which includes mapping the tasks onto the selected resources, deploying the program instances, and finally triggering the execution, based on an execution plan named *concrete workflow*.

The analysis conducted in [186] on the status and challenges of scientific WMSs has highlighted the issue of the data deluge challenge in modern science experiments. The scientific community is facing an imminent flood of data from the next generation of experiments and simulations, as recognised in "*The Fourth Paradigm*" [99]. There are many reports identifying requirements for data-intensive computation [4, 19, 23, 98, 103, 160]. Enhancement of workflow technologies is crucial to survive the data deluge.

This dissertation focuses on optimising the mapping phase during enactment of data-intensive workflows and investigates the potential for reducing the overall workflow execution time by making the best use of data streaming. Workflows are modelled as directed graphs. Tasks are handled by software components named processing elements (PEs) and represented as nodes in the graph. The nodes are connected in a pipelined streaming manner, which allows the overlap of PEs' executions—a source PE continues data production while a consuming PE consumes its output, permitting some of the PEs to be executed simultaneously on different portions of a data stream. The streaming technology can process workflows with large-scale data by an efficient implementation of buffering in main memory, and the processing speeds of memory access outperforms disk by a factor of $> 10^5$ [104].

1.1 Motivation

The arrangement of the enactment is a challenge for WMSs due to *a*) the *complexity and diversity of the applications* run in scientific experiments, *b*) the *heterogeneity of the computing platforms* used for the enactment, and *c*) the *volume of data* involved in these experiments. The WMSs need to supports various applications from different domains, e.g. astronomy [21], physics [33], biomedicine [112] and meteorology [143]. For instance, Bharathi *et al.* [24] characterise five scientific workflows from five diverse scientific applications, which involve different computation steps, but are equally complex. On the other hand, the advancement in computing over the last decades has fostered the development of a huge variety of computing technologies, in system architectures (e.g. cluster computing, grid computing and cloud computing), computing models (e.g. map-reduce and data streaming) and data-storage mechanisms (e.g. databases and distributed filesystems).

On top of that, the digital revolution is transforming the way research is conducted, where data-intensive computing is recognised as the "*the fourth paradigm of science*" by Jim Gray [85]. We are facing the challenge of handling the deluge of data generated by sensors and modern instruments that are widely used in all domains. Callaghan *et*

al. [35] describe the increase of the complexity of the workflow system following the growth of data size, in managing the execution of an earthquake science application, CyberShake [84] on the TeraGrid[1], using Pegasus [54] and DAGman. The examples below highlight some of the projects from various scientific domains that are dealing with large scale and distributed data:

- *Optical astronomy* – The Pan-STARRS project[2] for detecting potentially hazardous objects in the Solar System, is equipped with four 1.4 Gigapixel resolution digital cameras, that will capture more than 1 PB of raw data and generate 100 TB data into the cataogue database each year. Everyday, a *Load Workflow* creates about 700 new Load databases storing nightly detected object, and once a week, a *Merge Workflow* merges 50,000 Load databases with existing 12 offline Cold databases, using Trident [151].

- *Radio astronomy* – The LOFAR[3] for observing the universe using next generation very low frequencies radio telescopes, is producing high-quality interferometric data on baselines ranging from 100 m up to more than 1000 km, from 24 core stations (within 2 km radius in The Netherlands), 16 Remote Stations (within 100 km), and 8 International Stations (includes France, Germany, Sweden and the UK) [96]. The data processing pipeline involves correlating and reducing data from all of the stations connected through a WAN using IBM Blue Gene/P supercomputer, real-time analysis and model tuning using a general purpose cluster, storing the raw data in temporary storage, and archiving the final data products for further use. This is both data-intensive and complex.

- *Seismology* – The VERCE project[4] aims to deliver an e-Science environment to the seismological community to exploit the increasingly large volume of seismological data. It will provide a comprehensive architecture and framework to support diversified data-intensive applications in data mining and modelling and the integration of the community data infrastructure with the computational infrastructure. This is a framework for executing heterogenous tasks that process large volume of data (e.g. 100 TB of raw data and 5 PB of modelling data), from geographical distributed and diversified data sources, on Grid, Cloud and HPC computing resources.

[1] TeraGrid: `http://www.teragrid.org`
[2] Pan-STARRS: `http://pan-starrs.ifa.hawaii.edu`
[3] Low Frequency Array: `http://www.lofar.org/`
[4] Virtual Earthquake and seismology Research Community e-science environment in Europe: `http://www.verce.eu/`

- *Meteorology* – The study of mesoscale weather phenomena such as storms and tornados involves running the forecasting model application on large amount of data input from distributed mobile radars and satellites using large-scale computational resources. The CASA and LEAD projects [143] demonstrate a cyber infrastructure for real-time weather prediction that operates on distributed computing resources deployed on the TeraGrid.

- *Environmental science* – The study of the pattern of bird species occurrence to understand their links with environmental issues [107] is a data-intensive research that involve synthesising data from different organisations (e.g. NASA[5], USGS[6], NOAA[7], AKN[8]), analysing these data using a high-performance computing infrastructure, and exploring the complex model and a large volumes of data through visualisations, using VisTrails [36].

- *Experimental biology* – The OME[9] provides flexible data management and interoperability tools for biological light microscopy, that deals with over 90 microscopy file formats and distributed image processing. Its OMERO software project [5] provides tools for organising, analysing, and visualising microscopy images and metadata. OMERO is built from a series of databases (i.e. binary image repositories, relational databases and HDF5[10] tabular data), middleware, and client applications (i.e. processing script written in Java, C and Python, and web browser), and has demonstrated the diversity and complexity of scientific research.

The projects above demonstrate the challenges of data creation, exploration and exploitation in the scientific community. However, such challenges also arise in commerce, industry, government and society as well. For instance, the Integrated Public Use Microdata Series (IPUMS) provides researchers and educators access to data from more than 111 censuses in 35 countries. Such global-scale micro-data is useful for the study of economic development, urbanisation, social science, etc [103]. Exploration of the rapidly growing and diverse data opens many new opportunities in business, research, design, policy formulation and decision making, if and only if we can improve our knowledge-discovery apparatus entering the data-intensive era.

[5]National Aeronautics and Space Administration (NASA): `http://www.nasa.gov`
[6]United States Geological Survey (USGS): `http://www.usgs.gov`
[7]National Oceanic and Atmospheric Administration (NOAA): `http://www.noaa.gov`
[8]Avian Knowledge Network (AKN): `http://www.avianknowledge.net`
[9]Open Microscopy Environment: `http://www.openmicroscopy.org/`
[10]HDF5: `http://www.hdfgroup.org/HDF5/`

Together, these factors have forced the computer scientists to rethink the design of workflow management systems considering the issues below:

1. *Workflow languages need a higher level of abstraction to support the diversity of both applications and computing platforms.* We are in a transition between a period where researchers encoded their entire computational process in an application to one where they encode specific processing steps that *are designed* to be used in a workflow. Because of this transition, the early workflow languages are designed to re-use pre-existing code. Hence they have to "*make do*" with whatever application programming interfaces that pre-existing code exhibits. Lack of well-defined interfaces between the abstract workflow and the implementation mechanisms causes a high dependency between the experiments and the execution platforms. As a consequence, workflows need to be rewritten each time one of their components or data sources has its implementation changed. The goal of scientific workflows is to save human effort by enabling scientists to focus on their scientific work, instead of dealing with complex computing issues [126]. What the community needs is a workflow language that provides separation of concerns, which supports the creativity of both workflows creation and implementation through a standard and robust mapping interface to enactment platforms.

2. *The mapping process needs to be automated.* Workflow optimisation is not something new. Optimisation occurs in some existing workflow systems. For instance, Pegasus optimises the mapping of workflows by exploring the opportunity for clustering small jobs on execution engines in order to reduce the deployment and data movement costs. By manual optimisation of an Astronomical application, Montage, Singh *et al.* were able to reduce the total execution time by a factor of ten [154]. However, looking at the variety of applications and the fast changes in technologies, it is nearly impossible to provide hand-crafted solutions for every workflow. Thus, the mapping process needs to be automated.

3. *Knowledge extracted from the data can improve future enactments.* It is very common that scientific users repeat similar requests over similar data as they iterate their understanding or process various samples in the exploration of variants and experimentation settings [71]. Moreover, there are fundamental computing tasks that have been used across domains, e.g. image processing and cross-correlation. The behaviour of the computing tasks can be better understood by analysing the performance collected during the enactments, which is very useful to improve future mapping decisions.

4. *The traditional batch processing model that involves staging of files is becoming very expensive following the growth of data.* These enactments involve large data movement between the data sources and execution engines, especially the staging of intermediate results. Besides, the cost of scheduling, dispatching and instantiating the PEs is relatively high in large-scale scientific workflows. Reusing PEs to process a data stream seems to be a key to data-intensive computing. Each PE works like an *actor* in actor-oriented workflow [28], or *operator* in database query evaluators, and performs computation on the data stream flowing through it.

1.2 Thesis

The hypothesis we address is:

> Data-intensive research can be made scalable by using data-streaming workflows. We argue that the enactment process can be made efficient using optimisation techniques in an appropriate architecture. We demonstrate that optimisation of data-streaming workflows can be automated by exploiting performance data gathered during previous enactments.

We support this hypothesis via the following steps:

1. We present a candidate data-intensive architecture that includes an intermediate workflow language.
2. We create a fine-grained measurement framework to capture performance-related data during enactments, and design a performance database to organise them systematically.
3. We analyse a new enactment strategy that makes use of gathered performance data.

1.3 Contribution

Methods of data-streaming workflow optimisation have been pioneered to address an emerging need. The requirements for data-streaming workflow optimisation have been identified. An architecture for addressing these has been developed and a set of experiments has been conducted. The contributions of this thesis can be summarised as follow:

1. *A fine-grained measurement framework to collect performance-related data from the enactment of data-streaming workflows.* The measurement framework is built within the Open Grid Services Architecture Data Access and Integration (OGSA-DAI)[11] framework. OGSA-DAI is a framework for building distributed data access and management systems [55] developed since 2002. The earlier prototype of the measurement framework was built for a preliminary experiment in studying the streaming behaviour of data-intensive workflows [117]. An enhanced version was then presented in [116], which has a fine and precise measurement of the timing for all of the data items' events, by observing the pipeline connecting each of the process elements. A hand-crafted optimisation based on the data collected from this measurement framework is discussed in [90].

2. *A systematic way to organise and manipulate performance-related data with a performance database (PDB)*[12]*.* This thesis proposes to improve the mapping process through performance data collected from previous enactments. To organise such a large volume of data for further analysis, we designed a schema for the PDB. We then built a prototype and used it in the ADMIRE[13] project—Advanced Data Mining and Integration Research for Europe [11, 12] for gathering performance related data. We refined a set of queries to extract information used for workflow optimisation. We used this PDB in the project for organising performance data gathered from experiments on real-world applications [116]. This validated our PDB design. We then used it for all of the measurements in Chapter 5.

3. *A novel three-stage mapping algorithm that demonstrate the mapping of PEs of data-streaming workflows onto execution engines using the performance data.* The experiments conducted based on real-world applications (presented in [12, 90, 116, 183]) provide precious information in understanding the behaviour of PEs during the enactments: *a*) some PE instances (PEIs) have to be assigned at a particular location, e.g. because they access a local data source, or have a constrained license, or hardware dependency, *b*) there is a significant variation in the PEIs unit processing cost, i.e. some PEIs do not impose significant workload, and *c*) load balancing between the execution engines is crucial. This thesis proposes to partition the PEIs into three subsets: *anchored PEIs*, *heavy PEIs* and *light PEIs*, and map them in stages. The mapping process is based on the knowledge discovered from studying the previous enactment data, which are automatically collected by the measurement framework above. Thus, this process can be automated.

[11] OGSA-DAI: `http://www.ogsadai.org.uk`
[12] Not to be confused with the Worldwide Protein Data Bank.
[13] ADMIRE: `http://www.admire-project.eu`

4. *Influence on the design of a new language for describing data-intensive workflows.* The *Data-Intensive Systems, Process-Engineering Language,* DISPEL [10] has been developed as one of the main deliverables of the ADMIRE project. Workflows written as DISPEL requests are processed to generate graphs for enactment. The finding from this thesis has been incorporated in the language design, especially related to DISPEL enactment, which includes: *a*) introducing `locator` modifier to indicate anchored PEIs, and *b*) defining the `rate` modifier to describe the relationships between the input and output rates of PEs.

5. *Influence on the design and implementation of a new data-intensive architecture.* The data-intensive architecture supports different types of users involved in knowledge-discovery process to run a wide range of applications across heterogeneous platforms. One of the key goals is the enactment of workflow requests written in DISPEL onto the data-intensive platform. This thesis has contributed in *a*) identifying the information services required to support the enactment process, *b*) exploring the opportunity of optimising the enactment, and *c*) evaluating the system prototype developed in the ADMIRE project.

1.4 Preview of architecture and application

In general, there are three types of participants involved in running scientific workflows: domain experts, data-analysis experts and data-intensive engineers. *Domain experts* are scientists who are interested in scientific discovery, and want to use various tools to manage and exploit data from their experiments. *Data-analysis experts* are knowledge-discovery workers who are expert in extracting information from data. They know the data-analysis methods, data-mining techniques and statistical methods. They help domain experts to understand and exploit their data. They have the skills to design data-analysis algorithms, but may not be familiar with handling distributed and large scale computation. Thus, they rely on the computer scientists, software engineers and systems engineers who are knowledgable in distributed-computing infrastructure to manage the data and computations. This last type of participant forms the category of *data-intensive engineers.*

All of the three participant groups work perfectly well in their own context, but may or may not be able to do each other's tasks. Domain experts know what data are needed for flood forecasting, but may not know how to retrieve and integrate data from all of the distributed monitoring stations. The data-intensive engineers can execute the forecasting workflows in an optimised environment, provided the data-analysis experts

created the prediction modules as workflow components. The successful story of the Sloan Digital Sky Survey[14] is a tremendous combination of efforts of astronomers and database engineers, to design the data handling mechanisms of the large-scale database built up over the years. Moreover, as mentioned in earlier sections, workflow systems need to support the enactment of many applications onto a wide range of enactment platforms. Time and money are spent rewriting existing applications to run on new and emerging computing platforms.

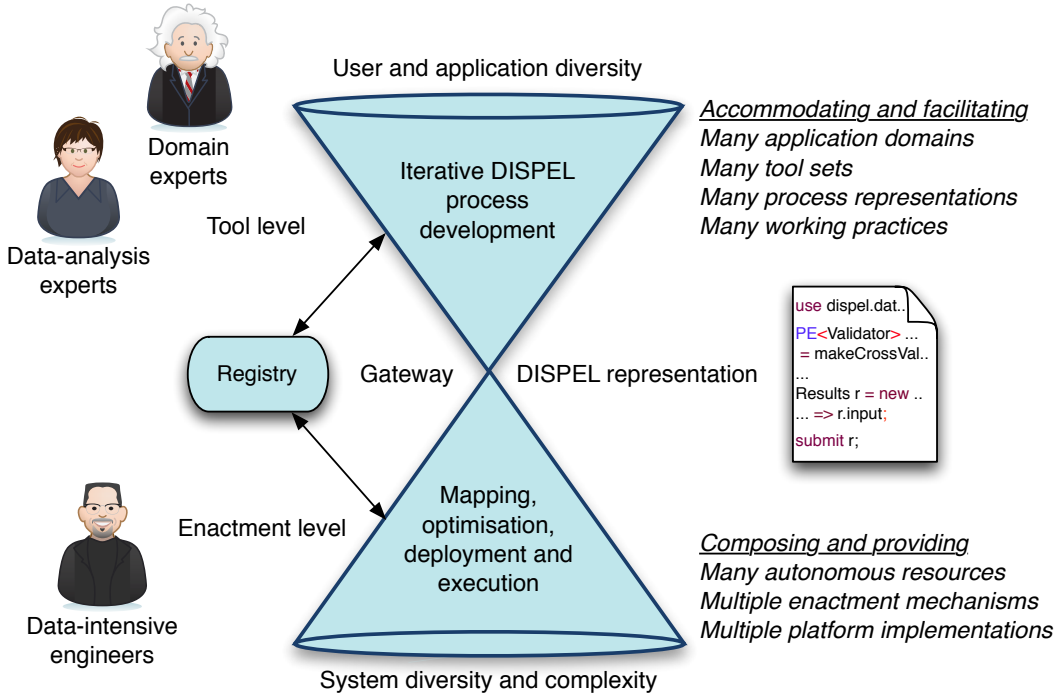


Figure 1.1: Data-intensive architecture.

The data-intensive architecture (⇝*4.1*)[15] addresses this problem by using a standard interface to support the diversity of both applications and platforms, as illustrated with an hourglass model shown in Figure 1.1. The top bulb is the tool level, where domain experts usually organise their experiments through a portal, and data-analysis experts design the analysis components using a workbench. The bottom bulb is the enactment level, where data-intensive engineers focus on the improving the enactment techniques using latest off-the-shelf technologies without knowing about the application domains. The interface of the two bulbs is a language, named DISPEL (⇝*4.2*). DISPEL provides the abstraction to separate the workflows design from their enactments. Data-analysis experts can concentrate on designing the analysis workflows at the tool level, and

[14] SDSS: `http://www.sdss.org/`

[15] We use a squiggly arrow followed by a section number and enclosed within parenthesises to mark a cross reference in the thesis. For instance, (⇝*1.4*) is read as (see Section 1.4).

pass the responsibility to the enactment engine to manage the evolution of enactment technologies. DISPEL is designed to be comprehensible to expert humans. It is a medium for dialogue between experts and an ideal notation for discussing, publishing, teaching and implementing data-intensive methods.

The enactment level is intended to support an enactment environment that is constructed from many autonomous resources, enactment mechanisms and implementations and is named the *data-intensive platform* (⇝*4.1.2*). The data-intensive platform comprises: *a*) a *gateway* — the entry point of enactment which accepts DISPEL requests, *b*) a DISPEL *language processor* — which compiles the DISPEL request into graph representation, *c*) an *enactment engine* — which optimises, maps resources, controls deployment, and initiates and monitors execution, and *d*) *execution engines* — which deploy and execute workflows.

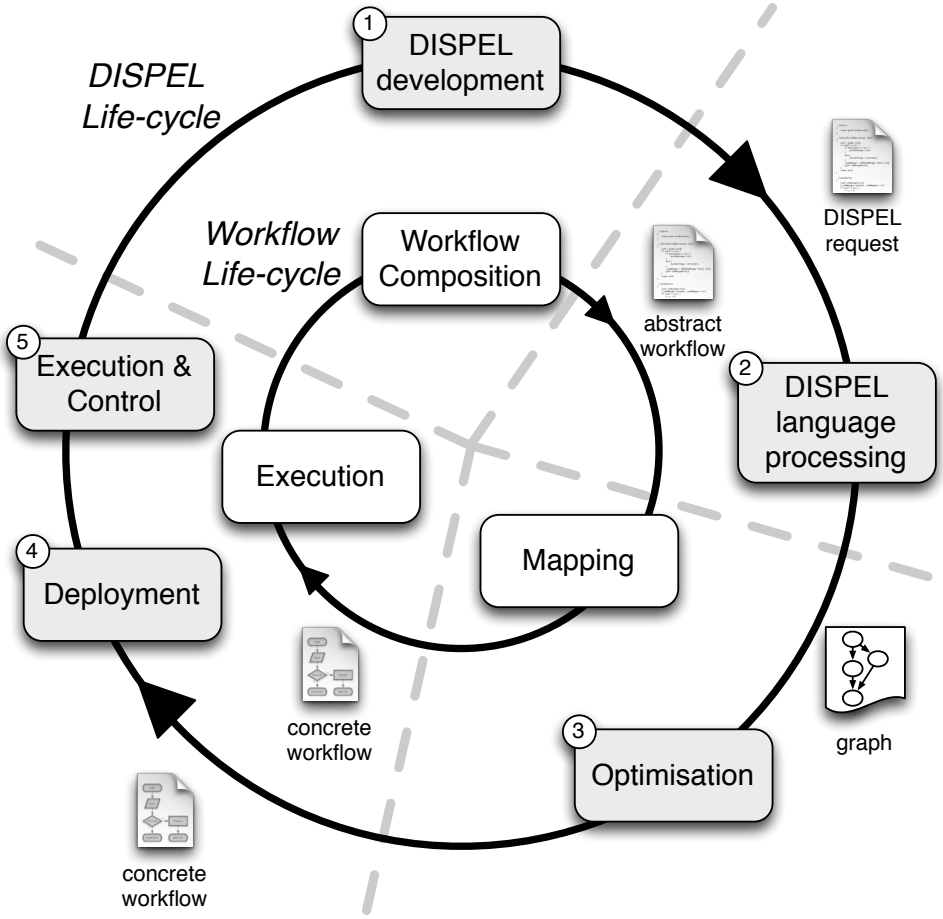


Figure 1.2: DISPEL life-cycle in the context of workflow life-cycle.

Figure 1.2 shows the life-cycle of DISPEL (outer cycle), as compared to a common workflow life-cycle (⇝*2.2.1*). The DISPEL development is undertaken at the tool level. DISPEL requests submitted to the gateway for enactment are validated and compiled into a graph representation. The optimisation stage consists of two processes: *graph transformation* (e.g. sub-graph substitution and parallelisation) and *resource mapping*. The latter finds the appropriate execution engines for each and every PEI in the graph. The output of an optimisation is a set of concrete workflows that are sent to execution engines for deployment. This is followed by the execution in a controlled environment.

There are two important sources of information used in the DISPEL life-cycle: the registry and the PDB (⇝*4.4*). The *registry* provides persistent storage for the definitions and descriptions of the data-intensive components (e.g. PEs, functions, libraries and data sources). The registry provides semantic information needed in the first three phases of the DISPEL life-cycle, and serves as a catalogue for users to register newly developed or deployed DISPEL components. The PDB organises PE-level performance-related data gathered from previous enactments (⇝*4.4*) and is used during the optimisation process.

The scope of this thesis is the optimisation stage. The remaining stages are outside the scope and implemented by other collaborators in the ADMIRE project. This thesis provides farther descriptions of those stages only as far as is necessary to explain the optimisation process. Readers who wish to view their full description are referred to [10].

The experiments conducted in this thesis (discussed in Chapter 5) are performed using a prototype of the data-intensive architecture developed in the ADMIRE project.

1.5 Terminology

A number of terms are used in this dissertation (introduced or imported) and are summarised in Table 1.1. A more detailed definition is given when each concept is first used in the text.

Term	Explanation
data-intensive	an adjectival phrase that denotes that the item to which it is applied requires attention to the properties of data and to the ways in which data are handled
data-intensive architecture	an architecture to organise data-intensive business, processes and systems
data-intensive platform	the invariably distributed infrastructure of hardware, software, services and operational procedures that provides a context for data-intensive computation
data-intensive virtual machine	an abstraction for the computational environment (i.e. the layers of software and hardware) in which a processing element instance runs during enactment
data-streaming workflow	a workflow where dependencies between the tasks represent flow of data
enactment	the execution of a workflow on data-intensive platform
gateway	a service that accepts and processes a workflow request
performance database	database designed to gather performance-related information at the level of processing element instances
processing elements	a software component that encapsulates a particular functionality to execute a task, and that can be used to construct a workflow
registry	a persistent store of definitions and descriptions of data-intensive components and their relationships to facilitate sharing and consistent use
streaming workflow model	a data-processing model where data arrives in continuous data streams and flows between processing elements
task	computational step in a business, scientific or engineering process
workflow	sequences of tasks structured based on their control and data dependancy to manage a computational activity

Table 1.1: Terminology.

1.6 Dissertation outline

Chapter 2 sets up a framework for thinking about the problems posed by this thesis. We discuss two major topics addressed in this thesis: streaming technology and workflow management systems. This chapter presents the background knowledge of the data-streaming model and a review of existing workflow systems, which gives a big picture of the workflow life-cycle, and how these systems handle it. The discussion is then focused on workflow optimisation and the relevant optimisations that happen in other problem domains. This chapter identifies related work that influences the thesis and shows how the work reported in this thesis relates to that larger framework.

Chapter 3 presents our approach to the problems posed by the thesis. This chapter starts by clarifying our streaming workflow model and defining the optimisation problem: graph transformation and resource mapping. The goals, requirements and context of the optimisation are presented, followed by the conceptual model for our optimisation approach. This chapter describes how the PEIs are partitioned into three categories, and mapped in stages. A three-stage mapping algorithm is proposed to demonstrate our approach of optimising the mapping of data-streaming workflows.

Chapter 4 describes the architecture that supports the proposed optimisation model. The data-intensive architecture is explained in detail to show how we achieve the separation of concerns in the *hourglass model*. This chapter brings the discussion of the architecture a step further by looking at the data-intensive platform, DISPEL language and the enactment process, and the information services that support the optimisation: measurement framework and PDB. This chapter discusses what are the performance-related data and how they are captured with an affordable fine-grained measurement framework. The last section of the chapter describes the life-cycle of the performance data and suggests a systematic way to organise and manipulate them with the PDB.

Chapter 5 presents the experiments conducted to evaluate the proposed optimisation model. The experiments are divided into three phases. The first phase aims to study the enactment behaviour of streaming workflows and evaluate the capability of the measurement framework in capturing the fine-grained performance data needed as the parameter for mapping optimisation. To have a better understanding of the streaming behaviour and more realistic evaluation, real-world scientific applications are used in the experiment instead of synthesised workflows. The second phase is to demonstrate by hand the potential of exploring parallelism in optimising the enactment of data-streaming workflows with the performance data. In this experiment, we measure the enactment performance of a life-science use case, and use these data to manually split

the workflow for executing in parallel. The last phase aims to evaluate the proposed three-stage mapping algorithm. The experimental use cases, experimental procedures and the results are presented, followed by our observations on their significance.

Chapter 6 draws conclusions from our research and highlights future research opportunities. The achievement section summarises what we have done, and emphasises again the impact and contributions of this thesis. The last part of this dissertation highlights the future direction of research that this work opens up.

CHAPTER 2

Related work

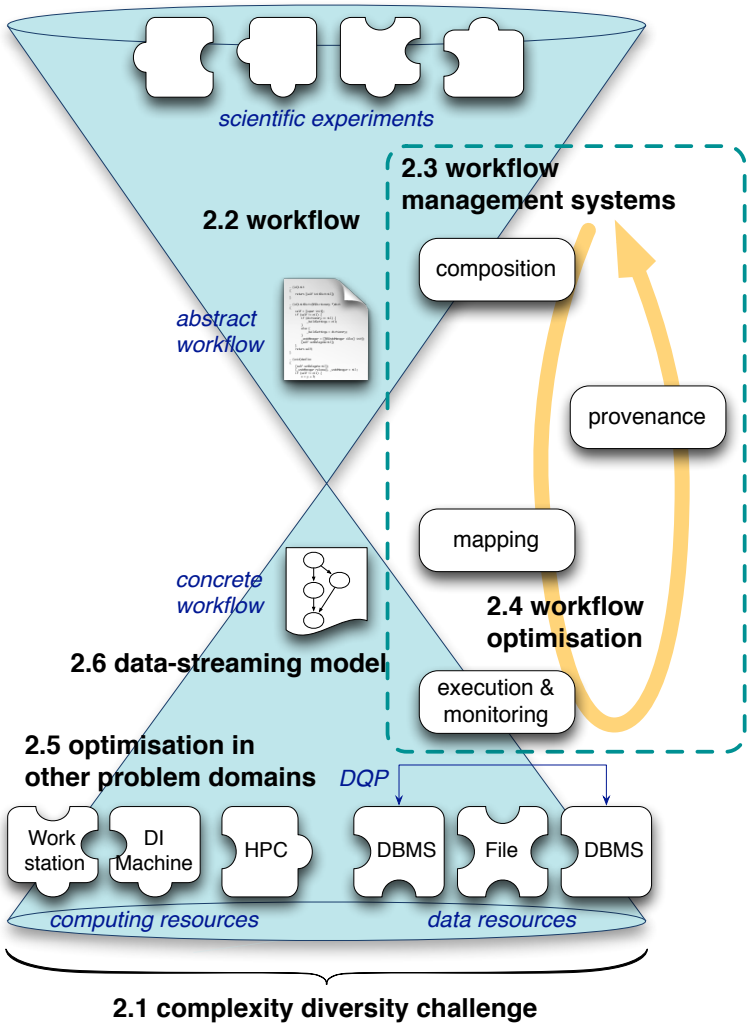


Figure 2.1: Chapter 2 overview.

This chapter sets up a framework for thinking about the problems posed by the thesis and shows how the related work shown in Figure 2.1 fits in this framework. The background is the hourglass model of the data-intensive architecture introduced in Section 1.4. We first look at the overall diversity and complexity challenge posed by applications in scientific experiments and the systems they run on in Section 2.1. Section 2.2 further examines the notion of workflows and their life-cycle, which is handled by Workflow Management Systems (WMSs). Section 2.3 describes the requirements of WMSs (Section 2.3.1) and proposes a taxonomy of WMSs (Section 2.3.2) which covers aspects that have been overlooked in related studies. The discussion is continued with a review of existing WMSs in Section 2.3.3. We then move on to the discussion on workflow optimisation in Section 2.4, as well as optimisation work that happens in other domains (Section 2.5). Section 2.6 clarifies our streaming context and describes the data-streaming model. Lastly, we summarise our discussion on the related work in Section 2.7.

Computational science has stood alongside *experimental* and *theoretical* science in scientific discovery over the last decades. The advancement in computing technology fosters the use of simulations to perform complex analysis in theoretical modelling. These simulations generate large volumes of data through analysis and reduction, which are stored in databases and files. At the same time, the revolution in digital technologies has increased the size of observation data in experimental science with the mass use of sensors and modern instruments, e.g. digital imaging devices in astronomy and microarray DNA sequencers in genomics [103]. The scientific community is facing a date deluge challenge, i.e. gigabytes of live data stream[1] and petabytes of dataset[2]. What makes the scientific data big is the repeated observations over time and/or space [104]. A good example is the decade-long earth science program, EarthScope[3], which is a set of integrated and distributed geophysical instruments to explore the formation, structure and evolution of the North American continent. One of the components is the USArray[4], which is a dense network of permanent and portable seismometers covering the entire United State over a ten-year period, which is shown in Figure 2.2.

Entering the 21st century, a new science paradigm has emerged, known as *data-intensive science* [99]. This new model is also called data-driven science where the scientists

[1]The Square Kilometer Array (`www.skatelescope.org`) will generate about 200 GB of raw data per second and the LOFAR (`http://www.lofar.org/`) low band antennas will generate 1.6 TB raw data per second.

[2]The Euclid Imaging Consortium (`http://www.ias.u-psud.fr/imEuclid`) will generate 1 PB data per year and the Large Synoptic Survey Telescope (`www.lsst.org`) will generate several petabytes of new image and catalogue data every year.

[3]EarthScope: `http://www.earthscope.org/`

[4]USArray: `http://www.usarray.org/`

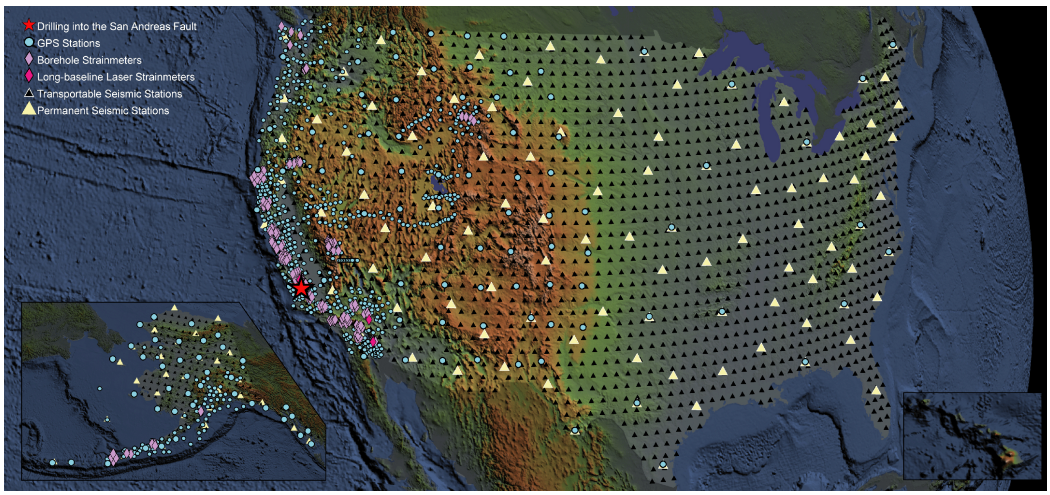


Figure 2.2: Instrument locations of the EarthScope planning map.

discover new knowledge by processing a large volume data captured in experiment or generated by simulations. As observed by Jim Gray in [85], astronomers do not actually look through the sophisticated and expensive telescopes. Instead of the raw data, they are working at the end of a data pipeline, looking at the information on their computers. Computing software and tools are used extensively in integrating and analysing these data to extract new knowledge. However, this does not imply that the data-driven science is the alternative way in scientific discovery, but as complementary to the existing paradigms, in an iterative cycle linking knowledge and observations [106].

Managing the data deluge not only requires larger storage space and more computation power, but also imposes the advancement of new technologies, e.g. scalable data-processing algorithms that can handle massive datasets, new data-management technologies for distributed and heterogeneous data sources, high-speed network for transferring large volumes of data [23, 81]. Boncz *et al.* discuss how they redesigned the database architecture in MonetDB, making use of modern technology to avoid the performance bottleneck in main-memory access [27]. Many of the datasets are stored in DBMSs which are designed for efficient transaction processing and not for scientific data. For example, three dimensional spatial time-series data in seismology need to be stored in a DBMS. Stonebraker *et al.* [160] have specified a common set of requirements for new science database systems, e.g. a new array data model and operators to process time series data.

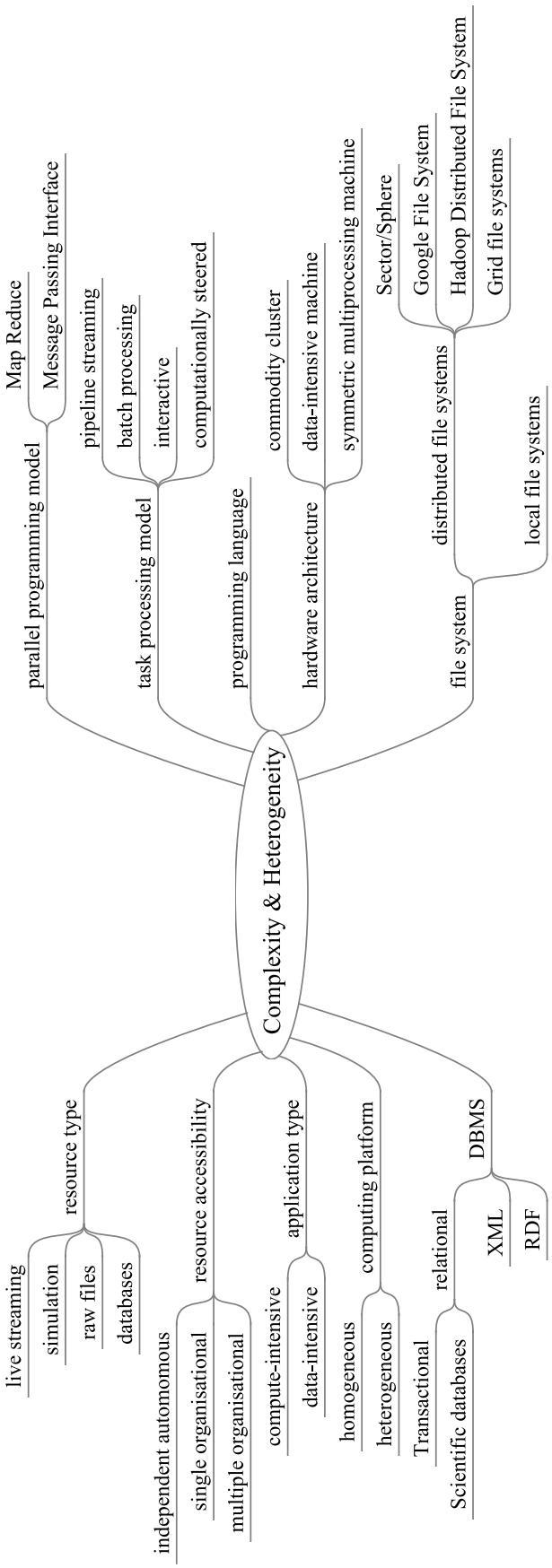


Figure 2.3: Complexity and heterogeneity challange in running scientific experiments.

2.1 Complexity and heterogeneity challenge

The data deluge is not the only challenge that the scientific community is facing. Another challenge is the complexity and the heterogeneity of the computing systems that support the experiments, the applications and the data, as shown in Figure 2.3. This figure shows some of the architectural aspects in building the systems and is not intend to give a taxonomy of the system architecture. Some efforts are undergoing in attempting cross-architectural implementation, e.g. Kepler/Pegasus [129] and Grid/Cloud[48], but the integration process is quite difficult. This is illustrated below:

- It is not unusual to run experiments that read raw data from distributed file systems, metadata from the databases, and live data streams from remote sensors. When collaborative work is involved, these resources may not be located at one site nor managed by a single organisation. The data-integration process needs to deal with different resource types and with a variety of accessibility constraints.

- Even if the experiment only involves data stored in a file system, there are different implementations available. The Sphere parallel data processing engine can efficiently perform massive parallel in-storage data processing on data stored in Sector file system (twice as fast as Hadoop MapReduce [87])[5]. However, it can not process data stored on a Gfarm file system[6].

- There is a broad spectrum of applications, from arithmetically intensive to data intensive. Each type of applications is suitable to run on certain hardware architecture. For instance, a commodity cluster provides high computing power with hundreds to hundreds of thousands of cores and usually is attached with a storage area network to store the data. This architecture is adapted to solve compute-intensive problems. However, running data-intensive applications will incur high communication costs to achieve lower performance because the disk I/O rate and the network bandwidth are the performance bottlenecks. In this case, data-intensive computing machines (e.g. GrayWulf [161] and Gordon [137]) outperform commodity clusters.

- Pegasus is a popular workflow management system used to manage the execution of experiments. It works perfectly well with DAGman and Condor[120, 165] handling batch processing, which stage in data and executable script into the high-performance computing (HPC) cluster and stage out the results after the execution. However, some of the programs are provided as web services and their

[5]Sector/Sphere: `http://sector.sourceforge.net/`
[6]Gfarm file system: `http://datafarm.apgrid.org/`

execution is orchestrated using different workflow systems such as Taverna, and the workflows systems are not interoperable [58].

In the real world, this complexity and heterogeneity challenge cannot be solved by unification of technologies as there are three main drivers encouraging its continued growth. Forcing a community to give up their *existing investments* and change to a new standard technology in managing their experiments is difficult. Money, negotiation and time are spent over many years to develop the operational practices and their associated community data interchange standards. When boundary crossing research links two such 'islands' of homogeneity, neither can afford to disrupt its community to comply with the other. Moreover, some legacy systems are hard to replace, or perhaps, too expensive even if it can be done. For instance, new workflow management systems still need to support the use of an old executable program written decades ago, either because no one can rewrite it in a new language, or it is too costly to do so.

Even if a community were to agree on a standard technology, the diversity will reappear sooner or later based on the *independent evolution* of technology that happens in each group. The Swift system is originated from GriPhyN Virtual Data System (VDS)[7] which is a multi-organisational collaboration designed to automate the analysis of the large quantities of data produced by high-energy physics experiments through a set of tools for expressing, executing and tracking the results of workflows. In the earlier stage, the VDS is using Chimera virtual data language [65] for expressing logical organisation of operations, Pegasus as the workflow planner, and Condor DAGman as the execution engine. The Swift system has grown to be a stand alone workflow system for peta-scale parallel execution [177], using its own SwiftScript to support iteration operations, and Falkon [144] for efficient task submission.

The third driving factor is the *socio-economic power of identity*. Cloud computing [8, 95] has emerged to be a new computing paradigm that provides dynamic and scalable infrastructure for application, computing and storage. The key players in the industry have shown their interests and started moving into this niche in the Internet ecosystem, e.g. Amazon[8], Google[9], Microsoft[10] and Rackspace[11]. Each of them has established their own strength and market share. Brynjolfsson *et al.* examine the cloud computing model as compared to another utility model, such as electricity, and conclude that cloud offerings will not be interchangeable across cloud providers [34]. This is a barrier for cross-platforms experiments.

[7] GriPhyN VDS: `http://www.ci.uchicago.edu/wiki/bin/view/VDS/VDSWeb/WebMain`
[8] Amazon Elastic Compute Cloud (Amazon EC2): `http://aws.amazon.com/ec2/`
[9] Google App Engine: `http://www.google.com/apps`
[10] Microsoft Windows Azure: `http://www.microsoft.com/windowsazure/`
[11] Rackspace Cloud: `http://www.rackspace.com/cloud/`

2.2 Workflows

As mentioned in earlier chapter, the emergence of computational and data-driven science as the third and fourth science paradigms boosts the use of modern computational technologies in simulations and knowledge discovery. With the help of *data-analysis experts* who master various statistical methods or data-mining techniques, domain scientists, referred to here as *domain experts*, try to discover new knowledge from their data collected from simulations and experiments, using various computational tools. This process often involves the steps below: *a*) moving data from the data sources into the computational resources, *b*) cleaning the data, *c*) constructing a model using part of the preprocessed data, *d*) validating the model with the remaining data, *e*) visualising the result, and *f*) moving the result to the storage system. The process can be modelled as a workflow.

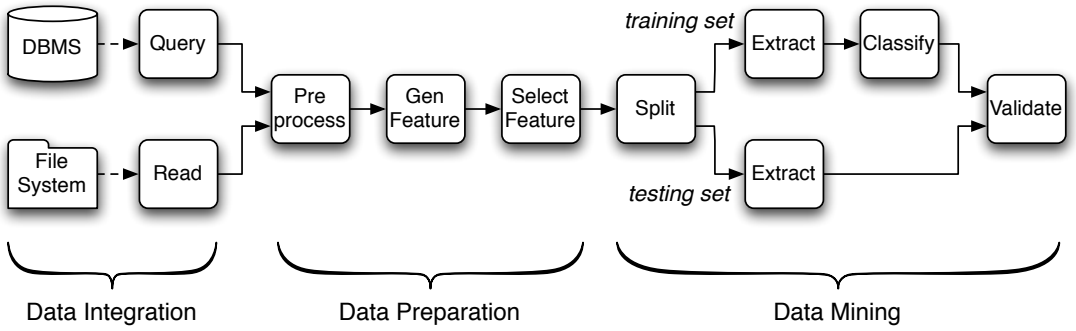


Figure 2.4: Common workflow in scientific experiments.

A workflow comprises three components: a list of tasks/operations, the dependencies between the interconnected tasks (the flow), and the data resources[12]. In a graph representation, the tasks and data resources are the vertices and the dependencies are the edges connecting the vertices, as shown in Figure 2.4. There are two types of dependency the edges can represent: control-flow and data-flow [150]. *Control-flow* graph made up of tasks and precedence constraints to control the flow. The *tasks are the operations* and the edges indicating the operations' order. The example workflow demonstrates the basic workflow patterns. The three tasks in the data-preparation stage form a *sequence*, which is executed one after another following the direction of the arrow on the edges. The sequence is then *split* into two parallel threads in the data mining stage. This gives the potential to run both threads in parallel. These threads are then *merged* back into a single sequence towards the end of the process. The graph

[12]Data resources are referred to both data sources (where data are read from) and data sinks (where results are stored).

can be directed cyclic (DCG) or directed acyclic (DAG). The main difference is that DCG supports iteration and DAG does not. Bharathi *et al.* provide a characterisation of workflows structures from different scientific domains in [24] and van der Aalst *et al.* have described twenty basic workflow patterns in [171].

In a *data-flow* graph, data is the key and the dependency represents the flows of data. In contrast to control flow, data are moving between the tasks, and are transformed by the tasks during the move. Tasks are operators and the edges indicating the movement of data from the source to destination. Assume the example workflow above is a data-flow graph, data from the Query operator (metadata) are joined with data from Read operator (raw images) and they flow into Preprocess operator. The Preprocess operator picks up every single data item (raw image), and performs a transformation (rescales the size and removes the image noise), before sending it to the succeeding operator, i.e. Gen Feature. Unlike a control-flow graph, a distinguishable feature of data-flow graph is that it allows overlapping in the operators' execution, to form a processing *pipeline*.

It is important to understand that these workflows are only logical models defining the steps in scientific experiments, known as *abstract workflows*. Abstract workflows define the tasks and their dependencies. To run the experiments on the computing platforms, these tasks need to be mapped to an instance of executable software components. This execution plan is known as a *concrete workflow*. Section 2.2.1 will describe this process in detail in the workflow life-cycle.

2.2.1 Workflow life-cycle

The workflow life-cycle has been defined in many existing works, for both business and scientific worlds [51, 80, 126]. All of these studies proposed their own phases of the life-cycle. Görlach *et al.* [80] suggested three phases in the life-cycle but Ludäscher *et al.* [126] suggested four phases instead, with a "workflow preparation" phase defined explicitly for staging data into the computing resources prior to the execution phase. Besides, only Deelman *et al.* [51] discussed the importance of the "provenance capture" phase, which provides information for workflow reproducibility. However, they all have the common observations below:

- these phases are viewed from the scientists' perspective, i.e. steps involved in creating and running workflows;
- scientists play the main role, i.e. they compose, operate and analyse workflows;

- scientific workflows are exploratory, i.e. it is common to reuse existing workflows and refine them in a trial-and-error manner;
- scientific workflows tend to be repeated, i.e. scientists re-run the same workflows with different parameters and datasets; and
- run-time monitoring is important, i.e. scientists monitor the progress and may decide to abort or suspend the execution.

These observations help to identify the requirements of WMSs, which are discussed in Section 2.3.1. We now discuss the life-cycle with the model adapted from Deelman *et al.*.

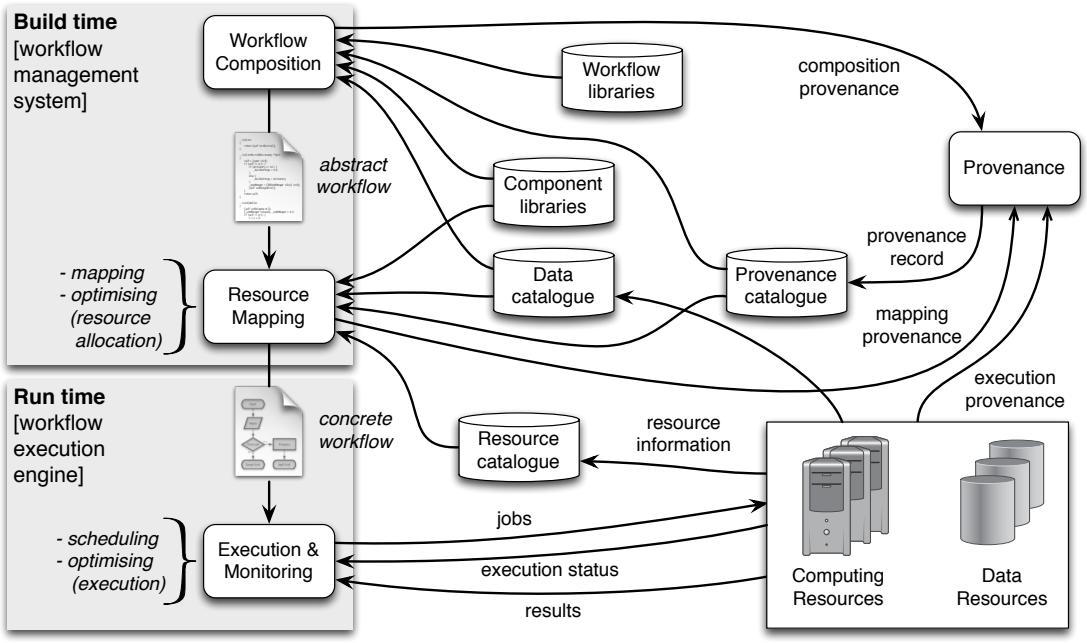


Figure 2.5: Workflow life-cycle (adapted from Deelman *et al.* [51]).

Figure 2.5 shows a typical workflow life-cycle that comprises four phases, namely *workflow composition*, *resource mapping*, *execution & monitoring* and *provenance*. Various tools and technologies are used in handling each of the phases. In general, *workflow-management systems* provide the tools for workflow composition and resource mapping; *workflow-execution engines* take charge of executing the workflows on available resources.

Workflow composition — constructs a high-level workflow known as *abstract workflow*. Abstract workflows identify the software components and data needed for a particular a computation, without details about the physical resources used in the execution of the workflow. The information about the components and data are stored in *component*

libraries and *data catalogue*, respectively. Abstract workflows can be created from scratch by defining workflow tasks using a particular representation from a workflow system. For instance, DAGMan [42] allows users to create the workflow as a DAG and execute it sequentially. Triana [163] provides a graphical-connection tool for data-flow creation which is later represented in simple XML with no explicit support for control structure. Some workflow creation systems, such as Wings [73], provide users with an existing workflow template with which to build their complex abstract workflows. Alternatively, users can select existing workflows from the workflow libraries, such as myExperiment—a virtual research environment that supports collaboration and sharing of workflows and experiments [146].

Resource mapping — maps workflow instances into executable plans named *concrete workflows*. Concrete workflows specify the suitable resources from the execution environment to be used in a particular computation. Workflow mappers obtain resource information from a *resource catalogue*. Selection of appropriate resources may affect the overall execution of a workflow. A good mapping can increase resource usage efficiency and execution performance. Thus, various optimisation techniques have been applied to refine the executable plan. For instance, one of the workflow refinement techniques used in Pegasus is to perform workflow reduction using available data products [49]. By consulting a *replica catalogue*, i.e. catalog to gather information about the locations of data products (⇝*2.3.3.1*), Pegasus determines which intermediate data are available and removes these redundant tasks from the workflow. We will discuss other workflow optimisation approaches, e.g. task clustering and data parallelisation in Section 2.4.

Execution & monitoring — enacts the mapped workflow in the execution environment and monitors the performance of workflow execution. The workflow execution engine is responsible for scheduling the tasks on assigned resources and receiving back the execution results. Some workflow execution engines, such as Condor [120, 165], work independently and can integrate with different workflow management systems; while some are built within the same framework with the workflow management system like ASKALON [60]. Besides, a lot of optimisation strategies take place in this phase. G. Singh *et al.* examine the various factors that affect the completion of astronomy workflow and biology workflow [154]. The finding shows that job submission rate, scheduling interval and dispatch rate influence the execution performance. Based on the information collected, they have optimised these applications and reduced the completion time. Abramson *et al.* use their expertise in parametric modelling using Nimrod toolkit[13] to develop a new workflow orchestration module in Kepler (⇝*2.3.3.3*)

[13]Nimrod toolkit: `http://messagelab.monash.edu.au/Nimrod`

to dynamically spawn parallel threads to optimise the execution of massively parallel parameter sweep workflows [1]. Other run-time workflow-optimisation strategies are discussed in Section 2.4.

Provenance — records the history of the data creation. Scientific experiments are very likely to be repeated with different parameter sets and data sets, and the experiments can be improved from each iteration. Moreover, provenance information is useful in the resource mapping phase in determining the optimisation approaches and parameters. Thus, keeping the data and process provenance is important. Several workflow management systems provide services to manage provenance information, e.g. Kepler [6] and Pegasus [109]. Stevens *et al.* describe how provenance is used to manage knowledge of *in silico* experiments [159]. Davidson *et al.* discuss the provenance challenges arise in scientific workflow systems to capture the provenance of complex data and workflow evolution [46]. Many provenance frameworks tailored for scientific workflows have been developed over the years, e.g. [29, 118, 153]. A good survey of data provenance techniques in e-Science is presented in [152].

2.3 Workflow management systems

A wide range of WMSs have been developed over the last two decades[14], e.g. Pegasus [54], Kepler [125], Taverna [100], Triana [163], Swift [185], Trident [15] and Meandre [123]. Studies in [77, 80, 164] show that the strengths of WMSs in managing workflows, include: *a*) supporting the collaborative research in the scientific communities for process sharing and data analysis, *b*) allowing easy workflow construction without exposing details of workflow management and execution, *c*) providing the ability to automate workflow steps, i.e. their mapping and execution, and to repeat experiments, *d*) integrating distributed and heterogeneous enactment platforms, *e*) handling large-scale and complex computations, and *f*) improving the execution through various optimisation techniques. Section 2.3.1 describes the requirements of WMSs in managing workflows and Section 2.3.2 identifies the major characteristics of WMSs that are used in reviewing some existing WMSs in Section 2.3.3.

2.3.1 Requirements

Studies in [52, 71, 77, 186] have describes the role of WMSs in managing workflows as well as the challenges that arose. From these studies and examples from other work,

[14] WMSs appear in business world in 1990's and only adopted by scientific communities in the 21st centuary.

we derive a list of requirements that WMSs should provide in order to support the workflow life-cycle, as shown in Figure 2.6.

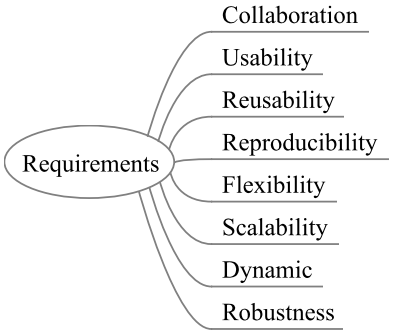


Figure 2.6: Workflow management systems' requirements.

Collaboration — Scientific research is collaborative. Scientists across organisational and geographical boundaries share data, services (i.e. codes and applications) and computing resources in running experiments. For instance, the Southern California Earthquake Center (SCEC)[15] is a community of over 600 scientists from more than 60 institutions world-wide that conducts geophysical research to develop a comprehensive understanding of earthquakes. Their research in seismic hazard analysis requires incorporating physics in their geological models and running a variety of earthquake simulation applications on Grid-based computing environments. To create a collaborative environment, the WMSs must provide tools to describe the tasks and their dependencies as workflows, support the execution of the workflow in the correct order in the distributed environment and manage the data and metadata [128].

Usability — As a tool to assist scientists managing their computational experiments and data analyses, usability is a very basic requirement to assure users' satisfaction. Scientists expect an effortless and efficient way of using WMSs to conduct their experiments, where they can focus on the scientific discovery without looking into the low-level execution of the computation tasks; on the contrary, sometimes they do like to interfere the workflow lifecycle, e.g. select preferred resources for mapping and terminate the workflow half way in the execution because the initial results are not within expectation. The Telescience project [119] demonstrate the role of portal as workflow controller, that enable users to manage data, services and collaborative tools through a simplified interface.

Reusability — Scientific workflows are exploratory in nature [14, 126]. Workflows are constructed by scientists and change frequently to incorporate their observation during each iteration of experiments. For instance, scientists add/remove experiment steps, or

[15]The Southern California Earthquake Center: `http://www.scec.org/`

try different approaches to perform the tasks, where existing sub-workflows/components are reused in constructing new workflows. During the rerun, scientists want to make use of the results from previous executions of the sub-workflows, whenever is possible. Ludäscher *et al.* introduced the idea of "smart rerun" that only executes part of the workflow that affected by the changes [125]. Besides, scientists often rerun the same workflow with different datasets. Supporting reusability requires a well structured provenance information about the workflows and the generated data.

Reproducibility — Reproducibility is the core of scientific method, where scientists repeat techniques and analysis methods done by others in validating their hypothesis [71]. The reproducibility requirement is different from reusability. Its main focus is on enabling users to rerun the workflow to obtain similar results. However, the solution to both requirements are the same, which is provenance. For modern experiments running on distributed and heterogeneous environment, reproducibility is difficult to perform. It requires the WMS to keep track on the tasks, parameters, data sources, computing resources, mapping configuration, and never the less, the data generated from the execution. WMS must keep the provenance information of the entire workflow life-cycle and data products.

Flexibility — The WMSs have to be flexible in integrating and accessing heterogeneous resources. Jones [105] highlights the important of flexibility to support biodiversity e-Science research, which challenging the limit of diversity resources that WMS can cope with. Typical biodiversity research needs to access different data and catalogues, e.g. species catalogues, geographical data and climate data, in different data standards, and analyse using distinct tools that are not interoperable. Some data are proprietary with access restriction, and some tools are accessible at particular locations, e.g. web service. Thus, the WMSs need to provide an integrated environment to manage the diversity of data and associated tasks, such as handling the communication between different tools, aggregating distributed datasets.

Scalability — The scaling requirement should be viewed from four dimensions [71]: number of tasks in workflows, number of workflows, number of resources (both computing and data resources) and number of users. The SCEC CyberShake project [35] has tested the scalability of the WMS, dealing with approximately 840,000 individual tasks. The workflow can be split into 80 sub-workflows, each with more than 10,000 tasks. A single run of the CyberShake workflow uses up to 800 processors on the TeraGrid, and generated 417,886 seismogram files (approximately 9.5GB of data). A single SCEC scientist can execute more than 2 million jobs a week. This large scale experiment has challenged the capability of the WMS in handling the execution pipelines,

staging files from/to between the sites, and optimising such large scale execution.

Dynamic — WMSs should support the exploratory nature of science and provide a dynamic execution environment which adapts to changing context and infrastructure. Scientists do not often have a fixed analysis in mind. They look at the results from the initial stage to decide the later analysis steps, thus, the execution is result-driven. Gannon *et al.* share their experience with Linked Environments for Atmospheric Discovery (LEAD)[16] project, in creating an interactive way for mesoscale weather prediction [68]. The simulation phase of the weather prediction cycle is a good example of result-driven execution. The system introducing finer computational meshes for further computation on the geographical areas that have shown interesting result during the first execution that scan across the entire landscape. The LEAD project also demonstrates resource adaptability, where the system will eliminate computation that failed to track the evolving weather, and allocate the computing resources to other simulation instances.

Robustness — Fault tolerance is an important requirement, especially in the distributed and heterogenous environment. The execution may fail due to parameter misconfiguration, missing input data, network disruptions, etc. WMSs should provide a recovery path for workflows execution. Simmhan *et al.* suggests few ways to enhance fault resilience in Trident, such as garbage collection (e.g. terminate unfinished tasks and reverse update to restore original database), data replication and rerun failed workflow from the provenance information [151].

This list of requirements above is not extensive and may not cover all of the challenges and requirements for WMSs for now or in the future, such as the automation support to perform routine data processing tasks, the instrumentation of the execution process, resource provisioning capability to improve the execution performance, and security. We intend to highlight some of the essential requirements in setting the background for our discussion in the later chapters.

2.3.2 Characterisations

Studies related to classification of workflows and WMSs have been published over the years. These studies have provided a thorough characterisation and proposed several taxonomies which are widely cited and used. This section does not intend to describe all of these taxonomies. We first provide a quick summary on the existing studies, and then discuss some of the missing aspects from these works.

[16]LEAD Portal: `http://portal.leadproject.org/`

Workflow technologies have existed for a long time and become popular in the business world in the automation of business process since the 1990s. Becker *et al.* [18] have classified three types of major business processes: workflow-supported organisational processes (with a high degree of human involvement), workflow-driven software processes (automated and handled by applications) and hybrid processes (a combination of both). They further describe the functionalities of workflow technologies in supporting the three process types from the organisational dimension, i.e. inter and intra-organisation level. Grefen *et al.* [86] describe the transactional workflow model based on workflow and transaction concepts, and discuss the transactional workflow support from both conceptual (i.e. specification language) and system point of view (i.e. workflow architecture). On the other hand, van der Aalst *et al.* [171] describe a collection of workflow patterns for identifying comprehensive workflow functionality, which is the basis for an in-depth comparison of commercial WMSs.

Over the last two decades, workflow technologies have been adopted by the scientific community in automating their computation experiments that run on distributed and high-performance computing infrastructures, e.g. Grid, and accessing data resources scattered across geographical locations. Yu *et al.* [184] characterise and classify various approaches for building and executing workflows on Grids. They review thirteen existing WMSs, discuss their similarities and differences in design and engineering, and suggest further research directions. Deelman *et al.* [51] extract a general taxonomy of features for WMSs from the scientists' perspective, which describe the workflow life-cycle in detail and compare the functionalities of a vast range of WMSs to support each phase in the life-cycle.

Beside the studies on the general taxonomy, a number of works that are targeting a *specific aspect* of the WMS have been published as well. Examples of these aspects include:

- *scheduling* — Wieczorek *et al.* [176] analyse in detail five different facets of the workflow scheduling problem: workflow model, scheduling criteria, scheduling process, resource model, and task model. For each facet, they describe extensively the taxonomy of different classes in each aspect, with an in-depth survey of the existing related WMSs.

- *verification and validation* — The correctness of Grid workflow specification and execution relies on verification and validation. Chen *et al.* [40] propose a taxonomy that describes the elements of Grid workflow *verification*, e.g. structure verification, performance verification and resource verification, and the *validation* which holds the key to the consistency between the processes and specifications.

- *workflow faults* — Lackovic *et al.* [113] conduct a systematic analysis of faults in scientific workflows, which helps detect, identify and correct potential faults that may arise during their execution. The taxonomy is divided into two parts: fault detection & identification and fault recovery. The first part identifies the type of faults which can be measured and detected, and the second part suggests associated recovery actions to overcome and correct them.

- *adaptive workflow management* — Han *et al.* [93] highlight the need for adaptive workflow management and classify various types of workflow adaptation at different abstraction levels, from the domain level down to the infrastructure level. They further discuss potential mechanisms for achieving adaptive workflow management.

- *provenance* — Provenance data captured by WMSs are tightly coupled with the WMSs and difficult to integrate across different systems. Da Cruz *et al.* [45] distinguish between different perspectives of provenance (i.e. capture, access, subject and storage), and provide a taxonomy of provenance characteristics that yields a better understanding of provenance data. They have surveyed eleven existing provenance systems, including some of the popular WMSs such as Pegasus, Taverna, Kepler and Swift.

These studies help to build up the understanding of WMSs from different aspects. However, as far as we are concerned, there are a few characteristics that are lacking from these taxonomies, which are important to distinguish our research. We illustrate these characteristics in Figure 2.7.

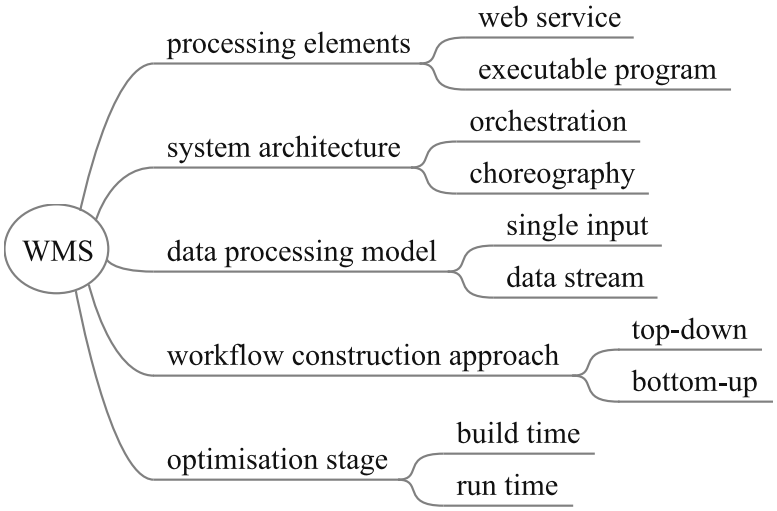


Figure 2.7: Characterisations of WMSs.

Processing elements (PEs), the building block of workflows, are software components that encapsulate a particular functionality to execute a task. Gannon distinguishes two type of workflows based on the PEs' implementation: component-based and service-based [67]. In *component-based* workflow, also known as *task-based* workflow, PEs are software components, which are encapsulation of software functionality that is accessed through an interface. Software components can be written in a programming language (e.g. Java class), a scripting language (e.g. Python script), or a proprietary application (e.g. MATLAB code). The execution of component-based workflows involves deployment of executable programs onto computing resources and staging data to and from data sources. In contrast, PEs are implemented as web services[17] in *service-based* workflows. Web services, also known as web applications, are self-contained and self-describing program codes wrapped and exposed through web servers [175]. The execution involves invocation of web-service instances through messages written in a manner specified by a Web Services Description Language (WSDL)[18] document[19]. We will discuss these workflow types in detail in Section 2.4.

The coordination of the flow of the execution in both types of workflows are very different. In component-based workflows, the software components are standalone applications that receive input data, perform the task, and produce output result. Workflow is formed by connecting these components together. WMSs take charge of the "plumbing", fetching results from preceding components and supplying them as input to subsequent components. In service-based workflows, the web services are separated web instances that may be located at different points on the network and which are independent from one another. A workflow is constructed by a collection of web services communicating with each other through message passing, and the control and data flow is explicitly defined by the coordination method.

In general, we can divide the *coordination method* into two categories: orchestration and choreography. *Orchestration* describes how services interact from a single controller perspective. Orchestration is centralised because the controller overseas the execution flow and invokes services based on the blueprint of the workflow written in a particular orchestration language, such as Web Services Business Process Execution Language (WS-BPEL)[20]. Even though initially written for the business world, BPEL meets the requirements of scientific workflows and is adapted in the scientific domains [59, 155].

[17] Web Services Architecture: `http://www.w3.org/TR/ws-arch/`

[18] WSDL: `http://www.w3.org/TR/wsdl`

[19] WSDL is the standard description language for SOAP-based web service. There are no formal way to document RESTful web services, but they can be describes in WSDL 2.0 (`http://www.w3.org/TR/wsdl20/`) or Web Application Description Language (`http://www.w3.org/Submission/wadl/`)

[20] WS-BPEL: `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`

Choreography describes a collaboration of a collection of services to achieve a common goal. In contrast to orchestration, choreography tracks a sequence of messages involving multiple parties in a decentralised manner, where no one party truly owns the conversation [16]. Web Services Choreography Description Language (WS-CDL)[21] is the language used for services choreography. Service orchestration works well in the business domain, but not for scientific applications, where data and applications are scatter across the network and owned/managed by multiple organisations. However, service choreography is more complex and invokes more communication. Barker *et al.* proposed a hybrid model with centralised control flow, distributed data flow model [17], which provides robustness and reduces data movement.

The "*plumbing*" mechanism described earlier is determined by the data processing model of the PEs, which can be divided into *single input*, where PEs read a single dataset (which can contain multiple data elements), and produce single output (which can also contain multiple data elements), and *data stream*, where data arrive in multiple, continuous and time-varying data streams [13]. In the latter, PEs read a unit of data item from the data stream and produce a unit of output data. It is best to describe both in the operations/operators terminology. In the single input model, PEs are *operations*, which are instantiated to process a dataset, and terminated after the process. WMSs execute these operations in sequence, and some operations may be executed in parallel provided there are no interdependencies among them. This is called *batch processing*. In the other model, PEs are *operators* that keep on working on data items as they arrive, and will not be terminated as long as data is continuously streamed in. Operators can be connected to form a *pipeline*, and executed concurrently.

In general, there are two approaches to construct workflows: *bottom-up* and *top-down*. For most of the scientific experiments, the individual program for each task is developed in the first place. Then the users describe the execution flow using a high-level description language, which can be a specific workflow language or a normal script. For instance, in Swift, a new application is built and added to the transformation catalogue before it can be used in writing a Swift program (we will discuss the transformation catalogue and how Swift works in the next section). As opposed to the bottom-up approach, users using the top-down approach first describe the workflow, then find the right mapping to the executable programs. If no suitable program is found, then users need to develop and add into the mapping catalogue. The top-down approach gives more flexibility to workflow design where a task can have multiple mappings. Besides, it also shields the workflow from the changes in the implementation of under-

[21] WS-CDL: `http://www.w3.org/TR/ws-cdl-10/`

lying executable programs. For instance, an image processing PE can be mapped onto a MATLAB implementation, as well as on a Java program.

The last characteristic is regarding the time when the optimisation is performed in the workflow life-cycle (see Figure 2.5). We divided the life-cycle into two stages. The first stage is *build time*, which is the first two phases of the life-cycle, i.e. workflow composition and resources mapping. This stage is handled by the workflow management system, e.g. Pegasus. The second stage is *run time*, which is the execution and monitoring phase, and is handled by workflow execution engine, e.g. Condor. The optimisations that happen during run time are more dynamic then those that happen at build time. The execution engine can obtain live status of computing resources and optimise the task scheduling. In contrast, build-time optimisation mainly focuses on graph transformation, e.g. task clustering and parallelisation. We will discuss the optimisation stage in detail in Section 2.4.

This taxonomy will be used in the next section to review WMSs.

2.3.3 Review of selected existing WMSs

Review of all of the existing WMSs is not feasible due to time and capacity constraints. We have selected five of them to be discussed in this section: Pegasus, Swift, Kepler, Taverna, and Meandre. The first four are well established WMSs and are widely used across domains. Meandre may not be as popular as the rest of the WMSs, but it is a data-flow system which is fully exploiting the streaming processing model and is the closest to our work. We will briefly give an overview of the technology, discuss some of the fundamental aspects of WMSs, such as architecture, development environment and workflow language. To aid our discussion in comparing the similarities and differences of these WMSs, we sketch an overall system architecture diagram for each of WMS, e.g. see Figure 2.8, that shows the workflow composition tool (coloured in *green*), the workflow management engine (coloured in *orange*) and the workflow execution engine (coloured in *yellow*), with other relevant components in the system. We then summarise our reviews based on the five characteristics defined in previous section.

2.3.3.1 Pegasus

Pegasus[22] is a well-known WMS that is widely used across domains, e.g. earth science [128] and astronomy [22]. Together with Wings and DAGMan Condor, it provides a complete workflow solution for handling scientific experiments. Wings[23] is a semanticly rich workflow system, used to create and validate workflows, and generate metadata for data products. Workflows are created and stored in workflow libraries. At this stage, they are referred as *workflow templates*, that are logical definitions of analysis process structures, with no physical binding to data or executable programs. The metadata that describe the semantic descriptions of the components and requirements of the workflow templates are stored in the repositories, which allow them to be discovered, shared and reused by different users and experiments. Wings helps the selection of workflow templates and data (from data repositories) to create *workflow instances*, which are also known as abstract workflows. Workflow instances have the specified data to be used for the computation, but still are independent from the execution resources. Pegasus maps the workflow instance onto the execution resource to create the *executable workflow*, which has all of the execution specification: the data to be used and their location, the computing resources selected for the execution, and the required data movements. Then, Condor DAGMan takes over the responsibility and executes the workflow on the distributed environment. Figure 2.8 illustrates the overall architecture and the interaction between these systems.

Wings plays two important roles in the life-cycle: workflow composition [72] (provides an alternative semantic rich way to construct workflows) and provenance tracking [109] (records the provenance of the creation of workflow instances and adds descriptions and metadata to new data products). The second phase of the workflow life-cycle, resource mapping, is handled by Pegasus. Pegasus is a workflow planner and has no capability to execute workflows. However, it can run on various execution engines, e.g. Condor and Globus. The input to the planner is an abstract workflow written in XML format, called DAX (example shown in Figure 2.9). The planner takes the DAX and generates a concrete workflow as a Condor DAGMan file (with `.dag` extension), which is the input to Condor DAGMan, the workflow executor used by Pegasus.

The mapping process relies on three important catalogues. The first catalogue, *Site Catalogue*, describes the compute resources, known as the sites, that are used to run the workflow. A site can be a cluster, virtual machines in Clouds, or local machines. Pega-

[22] Pegasus: `http://pegasus.isi.edu/`
[23] Wings: `http://www.isi.edu/ikcap/wings/`
[24] Example taken from Pegasus 3.1 user guide at `http://pegasus.isi.edu/documentation`

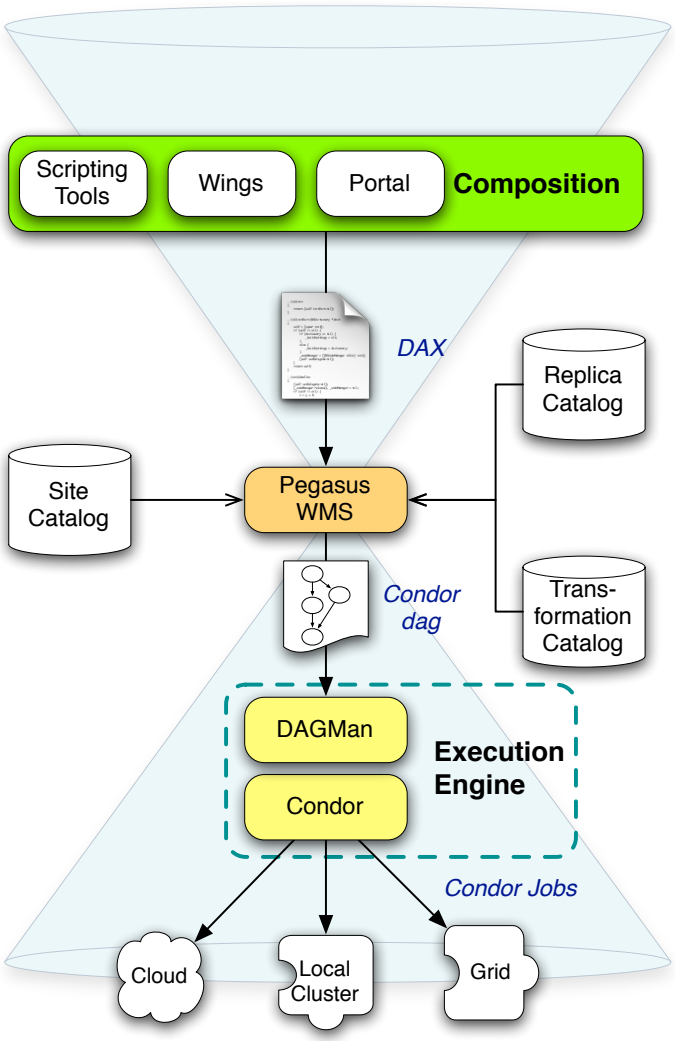


Figure 2.8: Pegasus system diagram.

sus has been proven to work on heterogenous and distributed execution environments spread across Grid and Cloud [48]. The second catalogue is used for data discovery to resolve the input/output files in the workflow. This catalogue named *Replica Catalogue* keeps mappings of logical file ids/names to physical file ids/names. In the DAX shown in Figure 2.9, the physical file for f.a, the input file of job ID0000001 is defined in the Replica catalogue. The third catalogue is the *Transformation Catalogue*, which maps logical transformations to physical executables on the system. For instance, one of the entries will be the mapping of the logical name of job preprocess to a physical path or URL, together with some system information of the site where the software component is installed. A user can define whether the component is stageable from the other sites.

During the execution, Pegasus uses Kickstart [172] to captures provenance of the job execution. Kickstart is a tool from the Virtual Data System (VDS) which is used to

```
<?xml version="1.0" encoding="UTF-8"?>
<adag xmlns="http://pegasus.isi.edu/schema/DAX"
...
      version="3.2" name="diamond" index="0" count="1">
...
  <job id="ID0000001" namespace="diamond" name="preprocess" version="4.0">
      <argument>-a preprocess -T60 -i <file name="f.a"/>
                -o <file name="f.b1"/> <file name="f.b2"/></argument>
                <uses name="f.b1" link="output" executable="false"/>
                <uses name="f.a" link="input" executable="false"/>
                <uses name="f.b2" link="output" executable="false"/>
  </job>
  <job id="ID0000002" namespace="diamond" name="findrange" version="4.0">
      <argument>-a findrange -T60 -i <file name="f.b1"/>
                -o <file name="f.c1"/></argument>
                <uses name="f.c1" link="output" executable="false"/>
                <uses name="f.b1" link="input" executable="false"/>
  </job>
...
  <child ref="ID0000002">
      <parent ref="ID0000001"/>
  </child>
...
</adag>
```

Figure 2.9: Pegasus workflow in DAX format[24].

manage the launching and capturing of execution provenance (exit status and runtime information). The provenance records are automatically stored in the Provenance Tracking Catalogue [50] and are useful for debugging. Pegasus has some features that makes it popular among the scientific community. The planner has a data management module to handles data transfer and output registration, which automatically adds *staging* jobs and *registration* jobs to the concrete workflow. It is also proven to be flexible and scalable [35]. The major advantage of Pegasus is the capability of handling workflow optimisation. For instance, in order to reduce overall execution time, users can use clustering technique to group small jobs together to reduce the overhead [41], and reuse data generated from previous runs. More strategies are discussed in Section 2.4.

The main architectural question concerning running workflows is why we need a mapper in the whole system. WMSs can support the direct composition of workflow from executable components. The argument for Pegasus is the benefit of abstraction, which separate the concern of underlying execution technology from the workflow design. Abstraction not only increases the usability of WMS, letting domain experts focus on their scientific discovery work without worrying about the low-level computing details, but also increases reusability: using the same workflow for different datasets. Abstraction enables the automation of workflow restructuring (creating different workflow instances from the same template by binding with different datasets) and execution.

2.3.3.2 Swift

Swift[25] is another technology that grew out of the GriPhyN VDS project, which was originally designed to automate the processing of large datasets from high energy physics experiments. From a simple virtual data language, Swift has matured into a powerful parallel scripting language [185] with an extensive runtime system based on CoG Karajan [173], that provides an easy, fast and efficient way to run large-scale loosely coupled computations on clusters, clouds and Grid resources across different domains, e.g. medical research [157], protein structure modelling [3] and climate modelling [179]. The Swift scripting language, *SwiftScript*, provides a set of *data-oriented language constructs* to specify how applications are connected and invoked to process collections of data files. To support the processing of file-resident datasets, Swift has the functionality for mapping file system objects into Swift variables and expressing the computations with iteration and branching capability.

Swift's main feature is the implicit parallelism and location transparency. Swift automatically parallelises the program, chooses the computing sites, handles the staging of input and output files (specified by the mappers), and invokes the execution remotely. Swift formalises and abstracts applications as functions, where the input data files become function parameters and the output data files become the function return values. Figure 2.10 shows an example of Swift program.

```
type messagefile;

app (messagefile t) greeting() {
    echo "Hello, world!" stdout=@filename(t);
}

messagefile outfile <single_file_mapper; file="hello.txt">;

outfile = greeting();
```

Figure 2.10: Example program written in SwiftScript[26].

This example script comprises four parts. First, it does type definition by declaring messagefile as a mapped type. There are three basic classes of data types: *primitive*, *mapped*, and *collection* (structures and arrays). Mapped types are used to declare data elements that are mapped to external files (e.g. messagefile in the example above is mapped to a physical file named hello.txt). Then, it defines the application procedure called greeting. The app declaration describes how the program is invoked. Similar to

[25]Swift: `http://www.ci.uchicago.edu/swift/main/`

[26]Example taken from Swift documentations at `http://www.ci.uchicago.edu/swift/docs/`

Pegasus, Swift uses a *transformation catalogue*, i.e. tc.data, to map the logical transformation used in the program (e.g. echo) to a physical executable located on the execution site. The third part defines a mapped type variable outfile, and the last part is the invocation of the greeting procedure. Swift provides a number of mappers, such as single file mapper (maps a single physical file to a dataset) and simple mapper (maps a file or a list of files into an array with specified prefix and suffix).

The Swift execution model is simple: non-collection data elements are single-assignment: only assigned to exactly one value during execution. A procedure or expression will be executed as soon as all of its input parameters have been assigned values. We illustrate this with the example below:

```
x = p(v);
y = q(w);
z = r(x);
```

Procedure p and q can be executed in parallel, but r can only executed after p. In this way, scripts are implicitly parallel. Moreover, the foreach construct that applies functions to array elements is executed in parallel for every element defined in the in clause. Thus, in the example below[27], the procedure analyse is invoked concurrently three times.

```
string fruits[] = {"apple", "pear", "orange"};
file tastiness[];
foreach fruit, index in fruits {
    tastiness[index] = analyse (fruit);
}
```

Figure 2.11 illustrates the Swift system that comprises a set of services to provide a parallel, distributed, and efficient execution environment for a Swift program. The Swift program can be constructed using any scripting tool, which is then compiled by the SwiftScript compiler into an abstract computation plan. The abstract computation plan is interpreted and dispatched to the execution sites by the execution engine. Sites are described in the *site catalogue*. Swift uses CoG Karajan as its execution engine. CoG Karajan supports remote job execution, file transfer and data management through abstract interfaces called *providers*. A *data provider* supports file transfer and data management on a wide range of protocols, e.g. GridFTP, SCP, FTP and direct copy. An *execution provider* enables the job execution from a variety of schedulers, e.g. GRAM, Condor, Sun Grid Engine (SGE) and Portable Batch System (PBS). The provider interfaces allow Swift to be easily extended to other execution environments by implementing a new execution provider. Instead of submitting tasks to the providers,

[27] Example taken from [178].

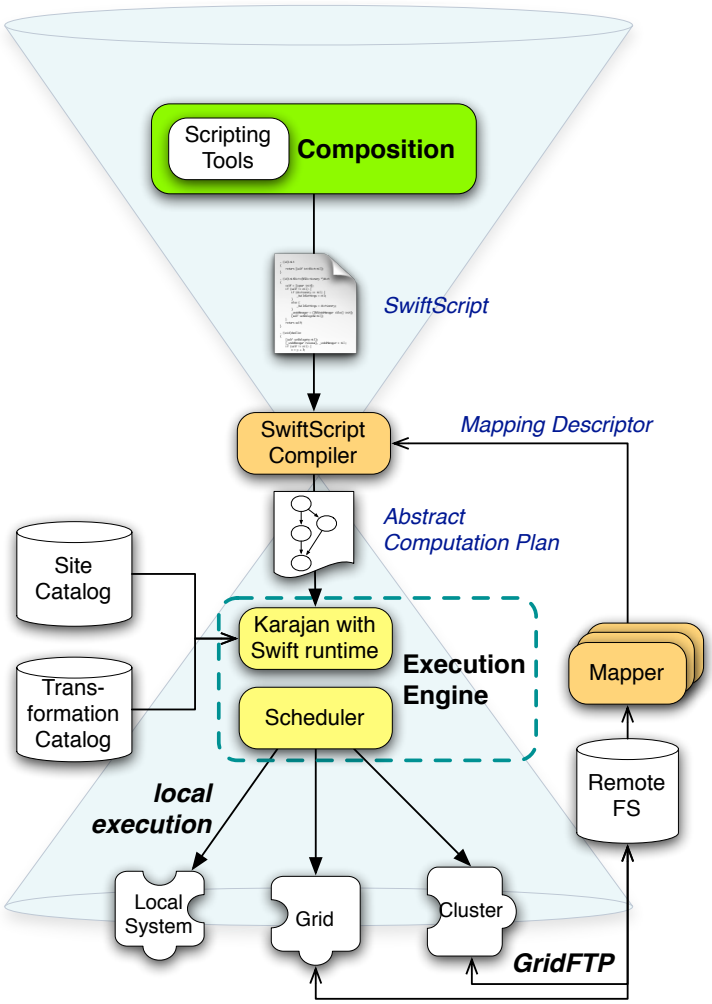


Figure 2.11: Swift system diagram.

Swift also supports the task execution using a provisioning and dispatching system, e.g. Coaster System (Coasters) [94] and Falkon [144]. Coasters is a light-weight node provisioning system for distributed systems that supports the use of *pilot jobs* on Grid, cluster and cloud resources. The initial pilot jobs are distributed to all of the remote sites, and are controlled by a centralised scheduler (any of the execution providers of the CoG Karajan). The pilot-job mechanism, such as Condor Glidein [149] and SAGA BigJob [124], improves the performance when executing pipelines of small jobs. The use of Swift Coasters achieves a 95% CPU utilisation of 2,048 computing nodes on an IBM Blue Gene/P system when executing 20,480 small tasks [178].

Swift use the VDS Kickstart to record execution provenance. Kickstart is used to invoke tasks and monitor their execution. Swift has a remarkable feature, which replicates the invocations, and automatically resubmits failed invocations. Swift does not provide a workbench for workflow composition, but it is used as the backend engine to accelerate

the development of Science Gateways [181], such as building the Generic Portals for Science Infrastructure (GPSI), a general-purpose science gateway infrastructure that improves scientific productivity [169].

2.3.3.3 Kepler

Kepler[28] originated from the Science Environment for Ecological Knowledge (SEEK) project[29] to support ecological research. SEEK aims to build a cyber-infrastructure that integrates three systems: EcoGrid (architecture for data storage, sharing, access and analysis), Semantic Mediation System (advanced reasoning system for data discovery to allow integration of disparate data resources) and Analysis and Modelling System (visual, automated environment where ecologists can design, modify and incorporate analysis to compose new workflows and models—which leads to the development of Kepler) [133]. Kepler is then slowly extended to become a general workflow infrastructure to support other domains through different collaboration projects, such as in chemistry (RESURGENCE[30]), geology (GEON[31]), molecular biology (SDMC Scientific Process Automation[32]) and oceanography (ROADNet[33]).

Kepler is built on the Ptolemy II[34] framework [57] that supports experimentation with actor-oriented design [28]. The actor-oriented model fits the exploratory nature of scientific workflows and provides an approach to workflow design, prototyping and execution of various types of computations across all scientific domains. Each process in a workflow is modelled as an *actor*, which encapsulates certain computing functions. Actors are independent software components that can communicate with each other through message passing using well-defined ports.

To support different execution semantics within a single architecture, Kepler separates the orchestration of actors from the execution engine, and use a set of software components called *directors* to handle the communication semantics among the actors defined in the model of computation. The actors define "*what*" are the processing tasks and the directors determine "*when*" the processing occurs in the workflow. To date, Kepler supports the following models of computation: Process Networks (PN), Dynamic Dataflow (DDF), Synchronous Dataflow (SDF), Continuos Time (CT) and Discrete

[28] Kepler Project: `www.kepler-project.org`
[29] Science Environment for Ecological Knowledge: `http://seek.ecoinformatics.org/`
[30] RESearch sURGe ENabled by CyberinfrastructurE: `http://ocikbws.uzh.ch/resurgence`
[31] `GeosciencesNetwork:http://www.geongrid.org/`
[32] Scientific Data Management Center: `http://sdm.lbl.gov/sdmcenter/`
[33] Real-time Observatories, Applications, and Data Management Network: `http://www.roadnet.ucsd.edu/`
[34] Ptolemy Project: `http://ptolemy.eecs.berkeley.edu/ptolemyII/`

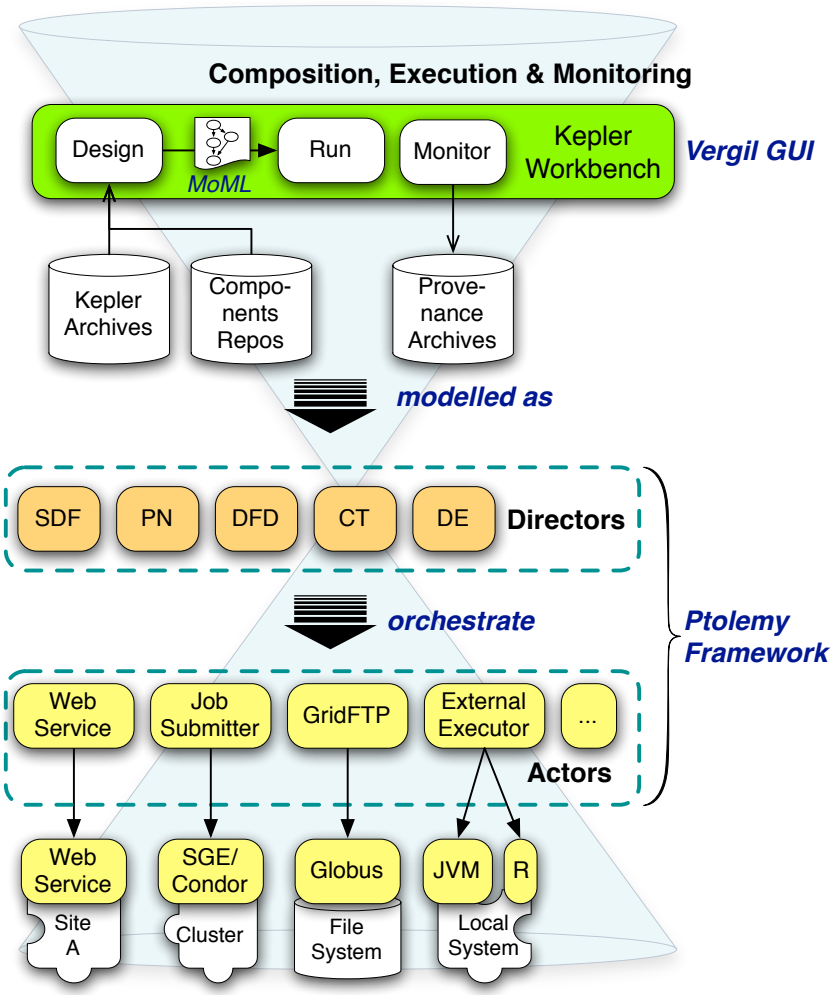


Figure 2.12: Kepler system diagram.

Events (DE). These directors can handle the orchestration of workflows with different requirements. CT and DE are used for workflows that depend on time, e.g. processing data generated by sensors over time and analysing population growth data. DDF and SDF are commonly used for basic transformation and filtering of non time-series data in scientific experiments. PN is for workflows that involve parallel threads and distributed execution.

The actor/director model gives Kepler the power of extensibility and flexibility, as shown in Figure 2.12. Kepler can be easily extended to integrate with new technologies or computing paradigms by developing the necessary set of actors, such as:

- integrating applications, e.g. use RExpression actor to run on R script and use MatlabExpression actor to run Matlab functions/scripts,

- integrating web services, e.g. WebService actor for web services described in WSDL, and set of actors for RESTful web services and Opal[35](a toolkit for wrapping scientific applications as Web services),

- supporting data movement on a variety of protocols, e.g. GridFTP actor copies files from remote Globus servers, SSHFileCopier actor connects to a remote host using SSH protocol to copy files/directories, and FTPClient actor uploads and downloads file from a remote FTP server,

- interacting with cluster (e.g. JobCreator and JobSubmitter actors for creating and submitting executables/commands to cluster systems) and Grid technologies (e.g. GlobusJob actor submits jobs to Globus host and SRBConnect actor allows user to connect to a data Grid provided by SDSC Storage Resource Broker (SRB)[36]), and

- executing shell scripts and applications on local machines, e.g. ExternalExecutor actor calls command line applications.

The director can also be extended to support new models of computation. For instance, Abramson *et al.* built Nimrod/K on Kepler's runtime engine, and created a new Tagged Dataflow Architecture director to support dynamic and parallel workflow execution [1].

As compared with Pegasus and Swift, Kepler provides higher usability through a powerful workbench, as shown in Figure 2.13. Users can graphically construct workflows using the workbench, which is built on top of Ptolemy Vergil GUI [32]. The Kepler workbench also has the execution capability that allows users to monitor the execution with access to the provenance archives (see [6, 29] for further discussion on Kepler provenance). The Kepler provenance framework provides APIs for collecting and recording workflow assertions and data-dependencies, and querying the provenance database. These APIs are used to developed actors to provide a provenance-based fault tolerance mechanism for workflow execution. For instance, Crawl *et al.* discuss the use of the Checkpoint composite actor, which has the "*exception handling*" capability found in programming languages to stop the execution of sub-workflow when an error is detected in a checkpoint [6]. Based on the recorded provenance data, Kepler can perform a fast replay of the workflow as a recovery action [29].

[35]Opal: `http://www.nbcr.net/software/opal/`
[36]SRB: `http://www.sdsc.edu/srb/`
[37]Snapshot taken from Kepler 2.1 getting started guide at `http://kepler-project.org/users/documentation`

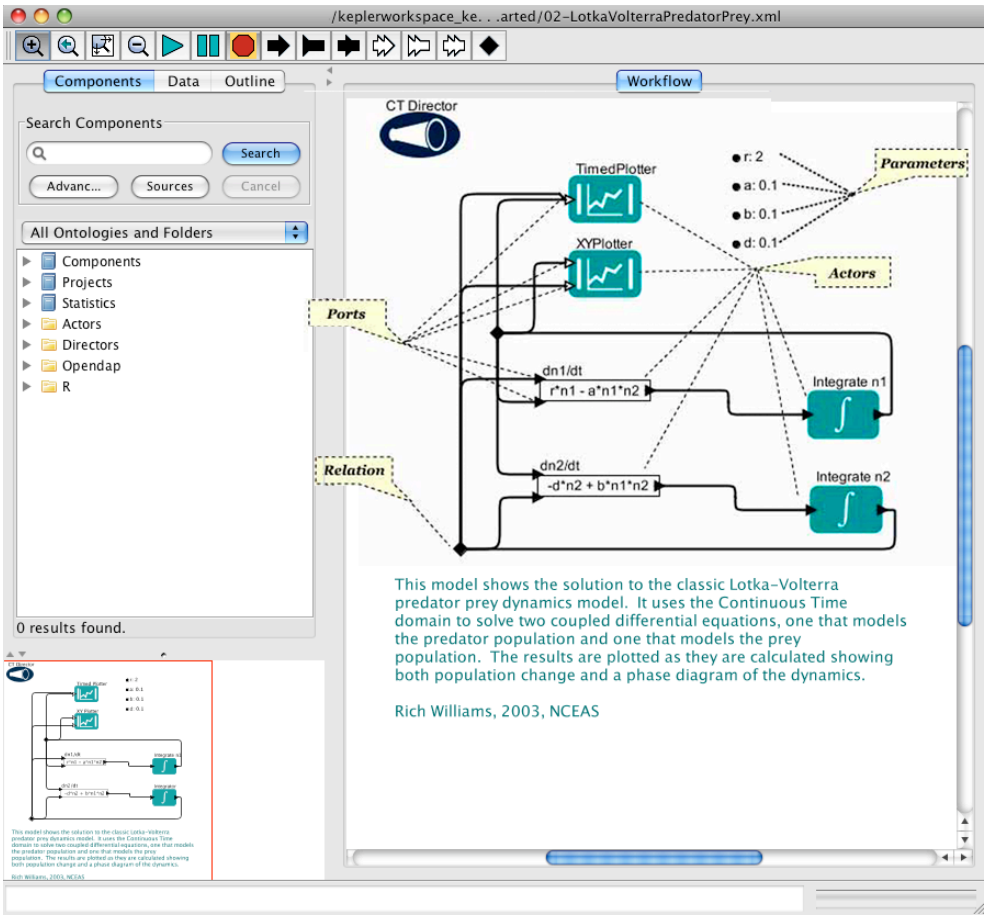


Figure 2.13: Main window of Kepler workbench tagged with Kepler's components[37].

Kepler maintains a central and searchable repository of actors and workflows to increase their usability. Kepler comes with a library of over 350 ready-to-use actors to support desktop-based workflows to large-scale distributed execution on the Grid, with the accessibility to EarthGrid[38] ecological data described in Ecological Metadata Language (EML)[39] format. Kepler workflows are saved in XML format using Ptolemy's own Modelling Markup Language (MoML), a language for specifying components and parameters. Besides, workflows can be saved in Kepler Archive Format (KAR). KAR is an archive format for storing and sharing of actors and workflows. KAR extends Kepler reproducibility where workflows saved in KAR format can be imported and rerun. Kepler is proud of its very own "*smart-rerun*" mechanism for handling parameter sweeps in scientific workflows, where data dependency is taken into account during the rerun, and it only executes parts of the workflows affected by the parameter changes. Together, these features make Kepler a highly usable and automated WMS.

[38]Knowledge Network for Biocomplexity: `http://knb.ecoinformatics.org/`
[39]EML: `http://knb.ecoinformatics.org/software/eml/`

2.3.3.4 Taverna

Taverna[40] is an open-source and domain-independent WMS created by the myGrid team[41], which has primarily focused on supporting the Life Sciences community (biology, chemistry and medical imaging) [139]. myGrid is a multi-organisational and cross-disciplinary research group that provides tools to help facilitating e-Science researchers, i.e. workflow management tool (Taverna), workflow and data sharing facility (myExperiment[42] and SysMO-DB[43]), protein sequence and structure analysis tools (Utopia[44]), and curated catalogue of Life Science Web Services (BioCatalogue[45]). This strong collaboration with the life-science domain experts has given Taverna valuable influence to become one of the popular WMS for "*in silico*" experiments.

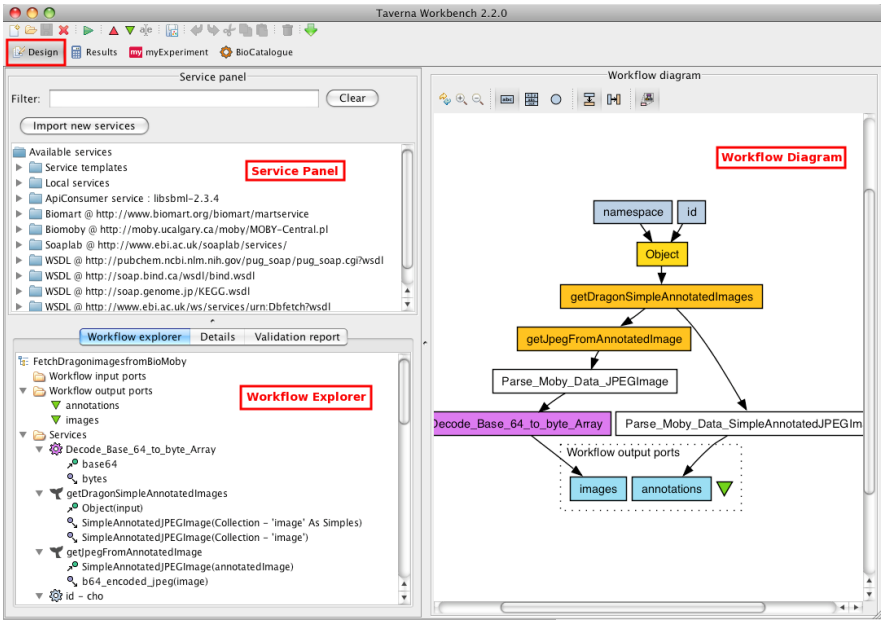


Figure 2.14: Design perspective of Taverna workbench[46].

With high usability as the first priority, Taverna provides an easy way for domain experts to find and design workflows through the workbench, as shown in Figure 2.14. Domain experts can search for existing workflows from a local repository, a remote URL or for a shared workflow in the myExperiment domain—a virtual research environment for collaboration and sharing of experiments [146]. Workflows are described as dataflow objects that can be serialised to t2flow files. Reusability is achieved at multiple

[40] Taverna: `http://www.taverna.org.uk/`
[41] myGrid: `http://www.mygrid.org.uk/`
[42] myExperiment: `http://www.myexperiment.org/`
[43] SysMO-DB: `http://www.sysmo-db.org/`
[44] Utopia: `http://utopia.cs.man.ac.uk/utopia/`
[45] NioCatalogue: `http://www.biocatalogue.org/`
[46] Snapshot taken from Taverna 2.3 user manual at `http://www.mygrid.org.uk/dev/wiki/display/taverna/User+Manual`

levels: *a*) workflows may be reused for different experiment parameters or datasets, by the same or different domain experts, and *b*) workflow fragments may be reused for constructing new workflows for the same or different domains.

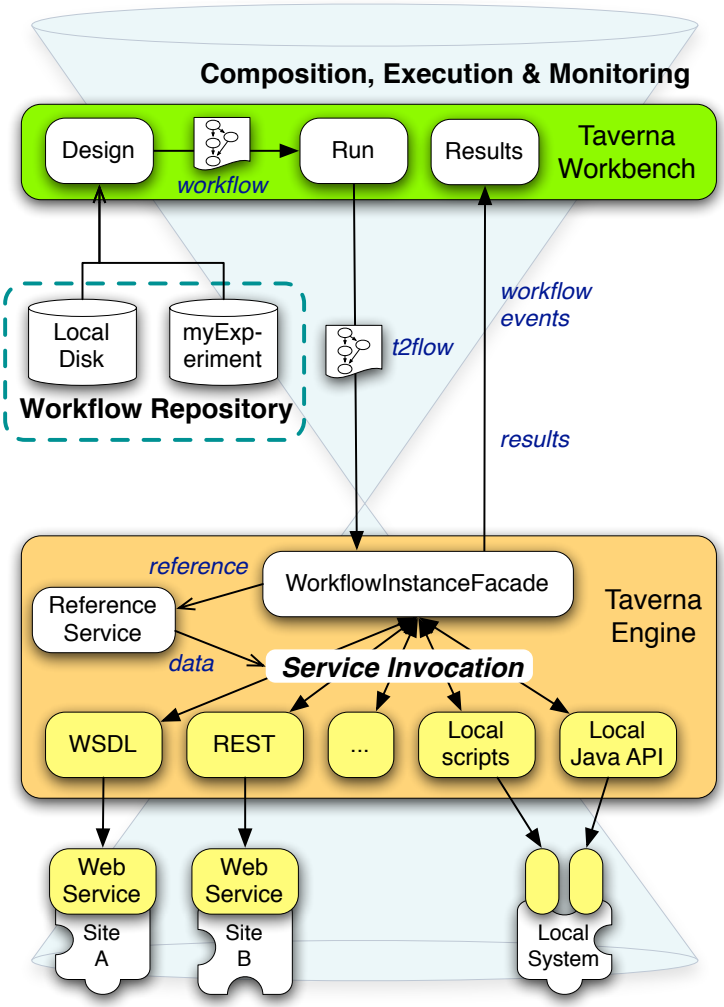


Figure 2.15: Taverna system diagram.

Taverna provides libraries of components for users to construct workflows to access a wide range of services. Users can add and share new services, which can be discovered and used by others. There are over 3,500 services made available. The services-discovery mechanism looks up services published in public registries (e.g. UDDI[47] and Grimoires registry[48], which is developed as part of the myGrid project), URLs submitted by users, or from a local disk [140]. Moreover, the Taverna workbench is equipped with a collection of local services for basic data manipulation and file I/O.

[47] Universal Description, Discovery, and Integration (UDDI) OASIS Standard: `http://uddi.xml.org/uddi-org`

[48] Grimoires: `http://twiki.grimoires.org/bin/view/Grimoires/`

The Taverna workbench is also used to execute workflows by submitting them to a local or remote Taverna Engine, where workflow instances (i.e. WorkflowInstanceFacade) are created to represent the running workflows, as shown in Figure 2.15. There are two major differences that distinguish Taverna from Pegasus and Swift:

- Taverna Workflows are connections of web services and Taverna coordinates their executions and connects the data flow; Workflows in Pegasus and Swift are the logical execution orders of a group of computing tasks (i.e. jobs);
- Taverna has no centralised enactment engine and the workflow itself performs the enactment (each PE is mapped to an object, which independently starts its own execution as soon as all of its input ports are populated with data item, and autonomously transfer its output to the next object upon completion [134]); Pegasus and Swift handle the staging of data/results and dispatch jobs to the execution engines.

Taverna invokes the relevant services and passes to these services the references to the actual data provided by reference services. The invoked services then use these data references to retrieve the data. The provenance information are captured and used for two purposes: execution monitoring (i.e. users can see the intermediate results through their workbench) and reproducibility (i.e. they can repeat workflow execution for performance assessment or debugging, and reproduce a data product for validation). Results are then sent back to the workbench.

Similar to The Kepler actor model, the Tavern plugin model is extensible. Users can develop necessary plugins that allow Taverna to support more than just web services orchestration. For instance, the BioCatalogue plugin enables users to browse and use the published life-science services. The UNICORE plugin extends Taverna capability to access UNCORE[49] resources, while the PBS plugin allows users to define workflows that can run on computational clusters that use PBS queuing system.

An important remark about Taverna is the evolution of its workflow language. In the earlier version of Taverna, workflows are written in Simple Conceptual Unified Flow Language (SCUFL), a high-level XML-based conceptual language [138]. SCUFL is a data-flow language that defines a graph of data interactions between web services. However, SCUFL does not have a unified way to extend service definitions for Taverna plugins nor support for some of the features in the newer Taverna Engine. Thus, it was replaced by t2flows, a serialisable XML format (easy to be shared and transported) in Taverna 2, which is more verbose but allows finer-grained details. Efforts are undergoing

[49]Uniform Interface to Computing Resources: `http://www.unicore.eu/`

to combine the simplicity of SCUFL and expressiveness of t2flows to create a new SCUFL2 workflow language that will be released together with Taverna 3.

2.3.3.5 Meandre

Meandre[50] is a semantic-enabled web-driven data-intensive flow execution environment developed under the Software Environment for the Advancement of Scholarly Research (SEASR)[51] project, which aims to create a research environment for humanities scholars to explore and exploit the rich digital data becoming available in their discipline, and to share their data and research. The main design principles of Meandre are providing a robust and scalable system to support the execution of data-intensive research scaled from a single laptop to a high-performance cluster, and fostering the research collaboration by reusing and sharing components [123]. Meandre is used in the Web-scale music analysis project to run NEMA[52] genre classification workflows on the supercomputing facility in the National Centre for Supercomputing Applications (NCSA)[53] [47].

Among the WMSs that we have discussed in this chapter, Meandre is the only data-driven execution system that is totally built on streaming model. The basic building blocks of computing tasks are called *components*. There are two types of Meandre components: executable components and control components. *Executable components* behave like a black box that performs computational tasks without human interactions during runtime, and are executed according to their predefined firing policy. Components are connected to form a *flow*, which is similar to workflow in our context. *Control components* are used to pause the flow during user-interaction cycles, which can be an HTML form that requires input from users, an Applet or another user interface.

Meandre's key feature of fostering sharing and increasing reusability of components and flows is built on semantic-web metadata manipulation concept. The resource description framework (RDF) provides a standardised exchange format for the metadata descriptions which allow them to be shared and reused across application, enterprise and community boundaries. Both components and flows have RDF descriptors to define the basic metadata, e.g. name, description, tags, and right. For executable components, additional metadata are added to describe the execution behaviour and the location for the implementation, e.g. firing policy, runnable, format and resource location. For flow components, the additional metadata focus on describing the logical connections between the components in forming the data flow, e.g. components instances, connectors,

[50] Meandre: `http://seasr.org/meandre/`
[51] SEASR: `http://seasr.org/`
[52] Networked Environment for Music Analysis: `http://www.music-ir.org/?q=nema/overview`
[53] NCSA: `http://www.ncsa.illinois.edu/`

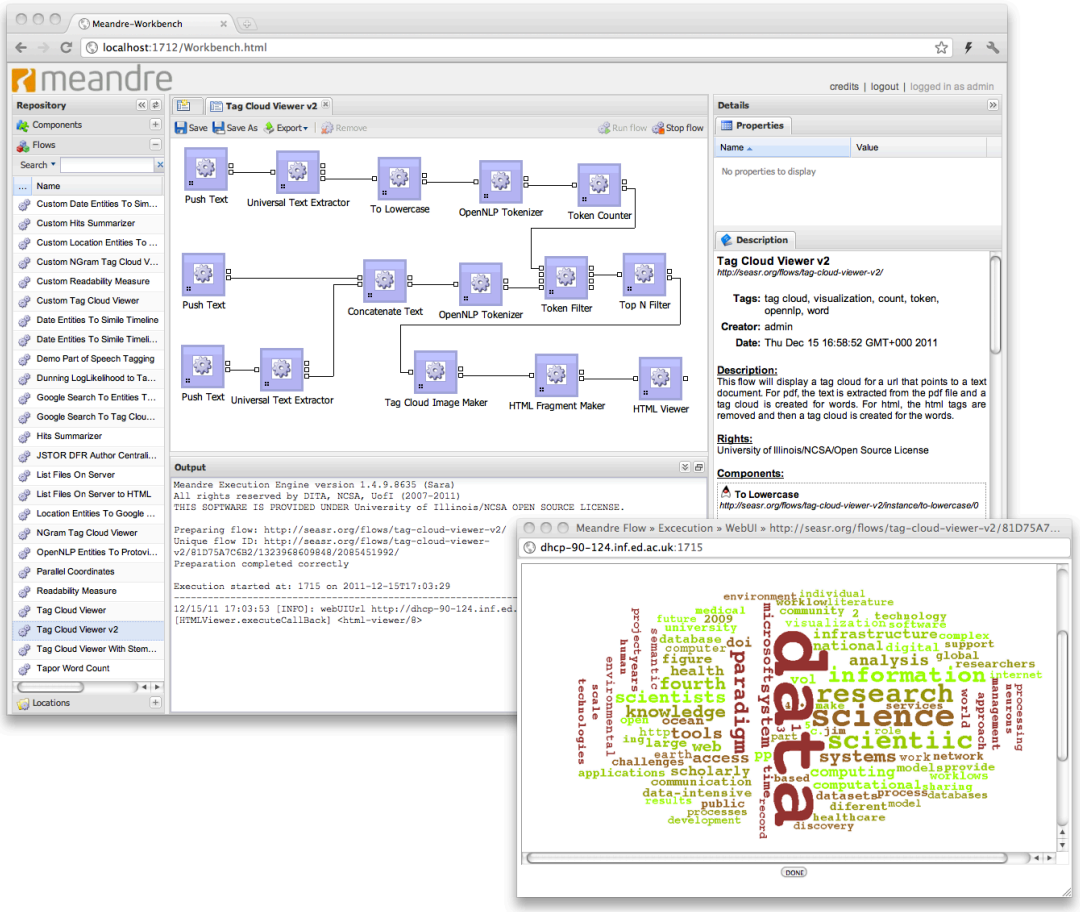


Figure 2.16: Snapshot of an example workflow in Meandre workbench.

connector instance source and connector instance target. The RDF descriptors are read and interpreted by the execution engine to find and initialise the components, determine how to establish the connection plumbing between the components and when to execute them.

The entire Meandre infrastructure consists of three parts [2]: *a*) tools for creating components and flows, e.g. Meandre workbench and Meandre Development Eclipse[54] plugin, *b*) a high-level language for describing workflow named ZigZag, and *c*) a distributed execution environment based on semantic-enabled service-oriented architecture.

The Meandre workbench allows users to discover existing components and flows, create new flows and execute them. Figure 2.16 shows an example flow that is used to display a tag cloud for a url that points to a text document, which is the "*The Fourth Paradigm*"[56] in this case. The workbench provides a visual programming environment

[54]Eclipse: `http://www.eclipse.org/`

[55]Example taken from Meandre documentation at `http://seasr.org/meandre/documentation/for-developers/zigzag/`

[56]The Fourth Paradigm: Data-Intensive Scientific Discovery: `http://research.microsoft.com/`

```
#
# This flow creates a flow converts to uppercase a sequence of
# strings and then prints it to the console
#
# @author Xavier Llor
# @date March 7, 2008
#

#
# Imports the three required components and creates the component aliases
#
import <http://demo.seasr.org:1714/public/services/demo_repository.ttl>

alias <meandre://test.org/component/push-string> as PUSH
alias <meandre://test.org/component/to-uppercase> as TOUPPER
alias <meandre://test.org/component/print-object> as PRINT

#
# Creates four instances for the flow
#
push_hello, to_upper, print = PUSH(), TOUPPER(), PRINT()

#
# Sets up the properties of the instances
#
push_hello.message, push_hello.times = "Hello World!!!", "10"

#
# Describes the data-intensive flow
#
@hello = push_hello()
@upper = to_upper(string:hello.string)[+AUTO!]
print(object:upper.string)
```

Figure 2.17: Example ZigZag program[55].

for users to develop flows by dragging and dropping components from the repository panel, and to link components by clicking on their ports. Another way to create data flow is using the Meandre ZigZag scripting language.

ZigZag is a simple declarative language for expressing the directed graphs that describe the flows. Figure 2.17 shows an example ZigZag script that creates a flow to convert a string into uppercase and display it on the console. The language provides four basic constructs. The script starts by *importing* three components (i.e. push-string, to-upper and print-object) from the server and *setting up aliases*. Then the aliased executable components are *instantiated* and properties are set (*change the behaviour of instances*). The last construct creates the directed graph that describes the data flow.

Meandre has a compiler to convert ZigZag programs into self-contained tasks, called the Meandre Archive Unit (in .mau extension). A mau file contains all of the metadata

en-us/collaboration/fourthparadigm/

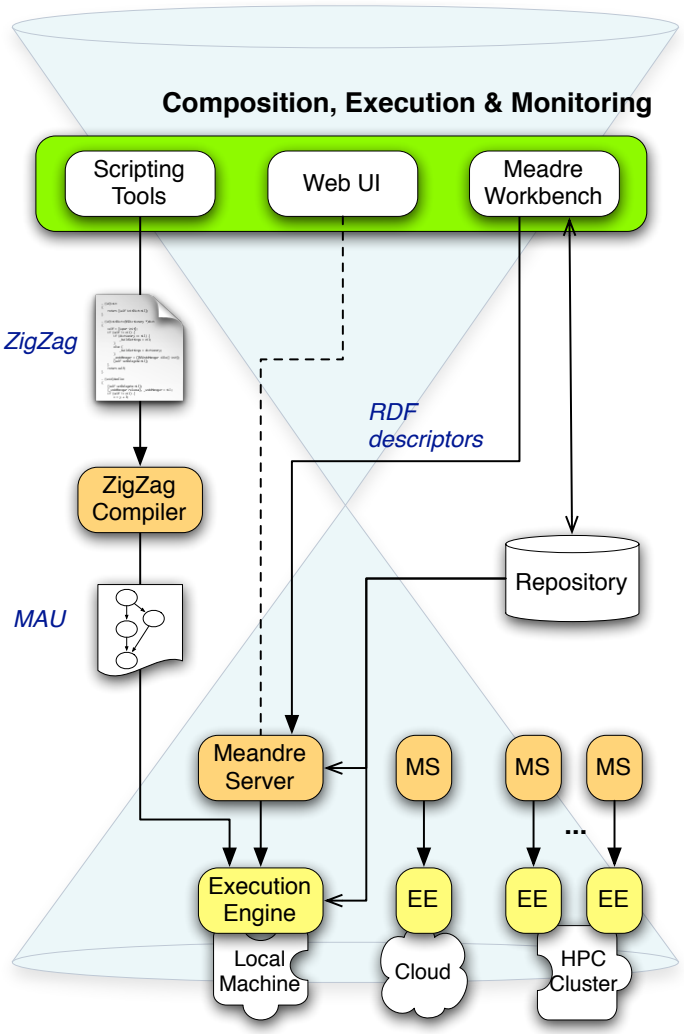


Figure 2.18: Meandre system diagram.

describing the executable components and flows, and the required implementation. The heterogeneity and scalability of ZigZag executions are transparent to users. For instance, the to-upper component is a Python executable whereas the other two are Java-based executables, but they appear no different from the users' perspective in creating the flow. The mau file can be executed by Meandre execution engine on a laptop or executed on their own in Grid environment via SGE as batch jobs. The mau file can be shared on myExperiment [145].

Another significant achievement of ZigZag is automatic parallelisation. The [+AUTO] tag on to_upper instance tells the compiler to parallelise the instance based on the underlying architecture. Alternatively, users can specify the number of parallel instance, e.g. [+4] for creating four instances.

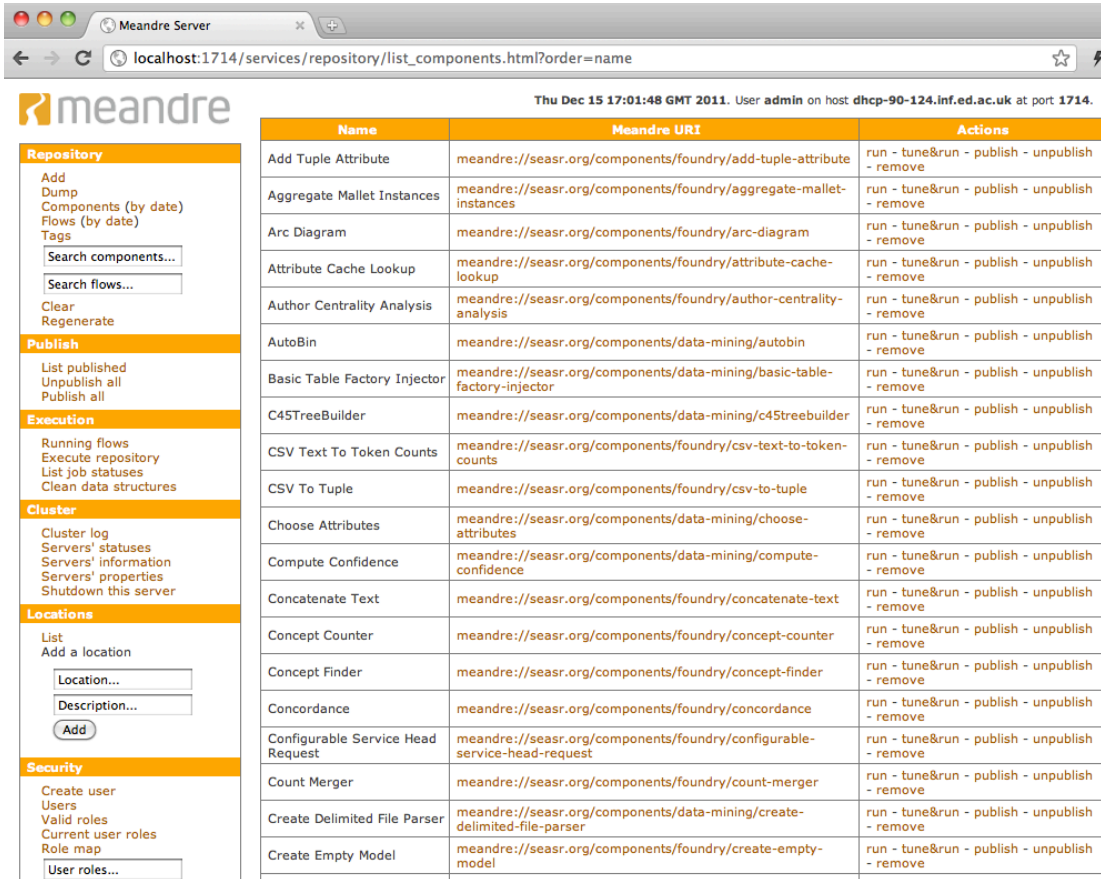


Figure 2.19: Web user interface for Meandre infrastructure.

Meandre provides a simple and flexible execution environment for data flow. The Meandre server comprises a metadata store, UI services and an execution engine. As a scalable architecture, Meandre server can be instantiated on demand, i.e. running single server on a local machine or adding more servers on a high-performance cluster if more resources are available (as shown in Figure 2.18). The Meandre server can be managed through a web GUI, as shown in Figure 2.19. Users can browse and manage the shared components and flows, run and monitor flows and perform other administrative tasks, e.g. managing user authentication. The execution engine initiates the thread for each of the components, executes them based on their firing policy and de-allocates the localised resources after the execution.

2.3.3.6 Summary

We look at the characterisations that have been defined in Section 2.3.2, as summarised in Table 2.1.

	Pegasus	**Kepler**	**Taverna**	**Swift**	**Meandre**
processing element	executable program	executable program & web service	executable program & web service	executable program	web service
system architecture	orchestrate	orchestrate	orchestrate	orchestrate	orchestrate
construction approach	bottom up	bottom up	bottom up	bottom up	bottom up
optimisation stage	build-time	none	none	run-time	run-time
data processing model	single input	single input & data stream	single input & data stream	single input & data stream	data stream

Table 2.1: WMSs taxonomy mapping.

Pegasus and Swift share a lot of similarities as they were evolved from the GriPhyN VDS project. The processing elements are executable programs. Both systems provide a logical-workflow layer to define the dependency among the processing elements, and mapper modules to map logical file names and tasks to physical files and executable programs. Both systems have demonstrated their capability in handling large-scale workflows that comprise hundreds of thousands of jobs. The current processing model of Pegasus is the *single input* model (⇝*2.3.2*) and efforts are underway to incorporate the data-stream model. Swift supports pipeline execution to improve efficiency [185].

Kepler and Taverna originated from two separate projects that involve different communities (i.e. ecology and life sciences), but they were created for the same reason: to facilitate the coordination of scientific experiments that involve web services and data integration across organisational and geographical boundaries. Their targeted system is a high-usability workbench that provides simple GUI for domain experts to design and run their workflows that connect web services. Beside web services, both systems extend their architecture to support program execution on local machines. Both systems support the data-stream execution model. Kepler supports pipeline execution [132] and has several successful stories on executing streaming workflow on cloud platforms [56, 187]. As for Taverna, one of the major improvements from Taverna 1 to Taverna 2 is the support of pipeline streaming to reduce the workflow execution time [134]. The granularity level of both WMSs is different. Taverna performs a coarse-grained pipelining execution that allocates a new thread for each of the element in the input collection, while Kepler supports pipelining of nested collections [132]. However, neither WMSs mention fine-grained streaming pipeline that comprises PEs that have multiple inputs and outputs with different data processing rates.

Meandre is built on a web-oriented data-driven execution concept that fully utilises the benefits of the streaming model. The Meandre components are executable programs that act as the operands to process a stream of data. The construction approach is bottom up. Data-analysis experts develop the components implementations and publish the components in a repository. Then the workbench is used to build the workflow by connecting components together.

In terms of coordination method, all of the five systems are categorised as orchestration, with a centralised controller that overseas the workflow execution on distributed environment. As for the workflow construction approach, all of the five systems are using the bottom-up approach, where each individual programs/web services are developed/deployed in the first place, and workflow is constructed by connecting them with visual tools or a particular workflow language. The use of workflow language brings us to the discussion of the level of abstraction provided by each of these systems.

Pegasus does not have a workflow language. The DAX format is mainly an XML-based description of the directed graph that forms the workflow. The abstract workflow used to describe the logical flow is a direct translation using two catalogues. The writing of DAX requires too much technical information from the users and changes at the implementation layer will trigger the reconstruction of the DAX. On the other hand, Swift has its own scripting language and provides a better abstraction with the SwiftScript compiler and mapper. The compilation of SwiftScript into parallel execution programs is well handled by the system and is totally transparent to the users. The use of a mapper in binding datasets to physical file systems reduces the complexity of data management in executing workflows that involve large-scale distributed and heterogenous data.

Kepler and Taverna have their own workflow language, i.e. MoML and SCUFL. MoML is mainly used for describing the workflows and not for providing abstraction. However, Kepler has its own mechanism to hide the system complexity and diversity. The actor/director model in Kepler is an extensible design that allows data-intensive engineers to scale out the Kepler architecture. For instance, Kepler extends the support to execution on a Grid by providing Grid-related actors. As for Taverna, SCUFL is designed to be extensible and to provide abstraction over different styles of Services. However, Taverna 2 has abandoned this simple language and replaced it with a verbose format.

The last characteristic is the phases of the workflow life-cycle when the optimisation is performed. Kepler and Taverna do not perform workflow optimisation, even though both WMSs have a good provenance and monitoring system, which may provide crucial

data to support optimisation. Swift has implicit parallelisation and pipeline execution support to allow runtime optimisation. Meandre has automatic parallelisation capability that creates multiple instances for components that have been tagged in the ZigZag script. Pegasus offers more build-time optimisation options. The next section will provide a detailed discussion on workflow optimisation.

2.4 Workflow optimisation

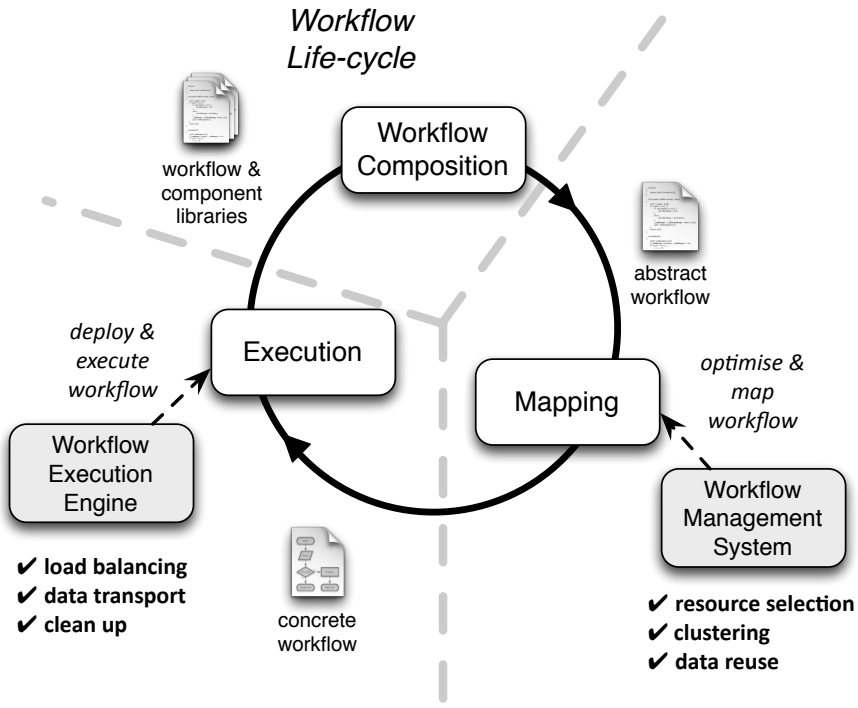


Figure 2.20: Timing of optimisation workflow-lifecycle.

Figure 2.20 shows the phases in the workflow life-cycle. In the *workflow composition* phase, a high-level abstraction of workflow known as an *abstract workflow* is constructed. Abstract workflows identify the application components and data needed without the details of physical resources to be used. An abstract workflow is mapped into an executable plan, called a *concrete workflow* in the *resource mapping* phase. Concrete workflows specify the resources in the execution environment to be used for the computation. A good mapping can improve resource-usage efficiency and enactment performance. The mapped workflow is deployed and executed in the execution environment during the *execution* phase. In general, workflow optimisation can be performed at the mapping phase (build time) and at the execution phase (run time). The optimisation approaches vary depending on the type of processing elements, i.e. executable programs or web services (⇝*2.3.2*). Glatard *et al.* [76] define two workflow manager

architectures: *task-based*, where the workflow manager is responsible for handling computing tasks; *service-based*, where computation is handled by external services. The former matches our characterisation of WMSs where the processing elements are *executable programs*, i.e. Pegasus and Swift, and the latter conform to *web-service* orchestration WMSs, such as Kepler and Taverna. We will discuss their resource mapping phase and execution phase in separate subsections.

2.4.1 Task-based workflows

Task-based workflow is *job-oriented*, where a set of jobs need to be executed on computing resources. The data-dependencies between the jobs have to be defined explicitly inside the job descriptions. The WMS is handling the control-flow of their executions, based on the abstract workflow. During the resource mapping phase, the mapper finds the resources that will be used for the job's execution. Two resource types are involved in a mapping: *data resources* and *computing resources*. For data resources, the mapping algorithm should select based on the relative location of the data and computing resources, and the current load on the data resources (when replicas provide a choice). When there is no copy already available at a site of computation for all of the possible computing sites, the mapper must generate *data-transfer tasks* and insert them into the concrete workflows, and *clean-up tasks* to eliminate the storage and access associated with the temporally re-located data. Similarly, result data must be moved back to storage or next-stage sites and cleaned up with the insertion of data-transfer tasks and clean-up tasks. For computational resources, the mapping should maximise the resources used by partitioning the workflow into sub-workflows and then executing those sub-workflows in parallel without increasing communication costs significantly.

During the execution phase, a workflow execution engine is responsible for scheduling the tasks on assigned resources and for returning the results or diagnostics. Scheduling is a match-making problem that assigns workflow tasks across computing resources. In a match-making process, criteria are defined (e.g. capabilities, workload and financial cost) and potential matching candidates are identified. The selection is performed by an algorithm that looks for an optimum matching. Scheduling is similar to resource mapping as both are doing match-making, but it is different in the input. A mapper maps tasks to categories of resources, then, during enactment, the scheduler dynamically assigns work to specific instances. Optimisation during the enactment stage could *a*) increase job throughput with a *load balancing* algorithm, *b*) increase reliability with *fault tolerance* mechanisms, and *c*) reduce communication cost by *minimising high-cost data transfers*.

We now examine three WMSs: Pegasus, Swift and ASKALON. Pegasus provides the optimisation methods below [53]:

1. *workflow reduction*—reusing available intermediate data products and removing the corresponding tasks to down size the workflow (with the assumption that the computation cost to reproduce the data is higher than accessing and transporting the available data);

2. *tasks clustering*—reducing scheduling overhead by combining small tasks together and submitting them as a single task;

3. *data cleanup*–removing data that are no longer needed to increase resource efficiency (particularly important when running data-intensive workflows on modest computing resources);

4. *workflow partitioning*—partitioning an abstract workflow into a set of smaller sub-workflows to adapt to changes in the execution environment (where the planning of the execution of a large-scale workflow on the dynamic and fast-changing Grid environment is difficult).

Pegasus also improves workflow performance using *placeholders*. Placeholders are units of work (either shell scripts or MPI wrappers) that are submitted to the queue of the execution engine, which once launched can be used to execute multiple tasks. The placeholder implementation is similar to the Glidein approach [149] in Condor. It is important to highlight here that these optimisation techniques are hand-crafted. The current system is not able to support automated optimisation.

Swift does not provide build-time optimisation options like Pegasus, but it provides a few mechanisms to improve run-time efficiency. Its implicit parallelism and pipeline-manner execution support enables the iterative processing of the array's items to run in parallel. Swift also performs clustering techniques to bundle groups of tasks to reduce submission overhead. The "*pluggable*" execution provider model provided by its execution engine, Karajan, enriches Swift scheduling capability so that it can benefit from any of the optimisation capabilities provided by different execution engines, e.g. pilot-job mechanism provided by Coasters.

ASKALON [60] is another example of a task-based approach WMS. There are three core services within the ASKALON working closely to optimise execution: the grid resource manager (GridARM), the scheduler and performance prediction. The resource manager handles negotiation, advanced reservation, allocation of resources and deployment of services. The scheduler is responsible for mapping workflows and monitoring execution, while the performance prediction service estimates execution times. To-

gether, they provide quality of service (QoS) by dynamically adjusting the optimised schedules according to the infrastructure status.

2.4.2 Service-based workflows

For service-based workflows, tasks are wrapped into web services. The abstract workflow defines the services which need to be invoked to accomplish the steps in an scientific experiment. During the mapping phase, the WMS discovers existing services running on particular machines for each of the tasks, binds the abstract tasks to the selected concrete services, and connects their inputs and outputs with respective services (based on their data-dependencies) to form a processing *pipeline*. The initialisation and execution of services are determined by their firing policies. For instance, a service with two inputs connected to two preceding services may start the execution only after the preceding services are terminated, or may start the execution as soon as it has received sufficient input data. The latter execution model, i.e. *pipeline streaming model*, allows the services' execution to overlap (⇝*2.6.3*). The binding information, pipeline description and firing policies are defined in the concrete workflow. During the execution phase, WMS uses this executable plan to invoke the web services.

There are two major differences between task-based and service-based workflows. First, service-based workflow can start executing without knowing the complete input dataset, which are dynamically fed in during the execution. For task-based workflows, datasets are defined prior to execution for job submission and data-staging. Dynamic-dataset support is very useful for scientific experiments, especially for those that involve incremental learning processes (i.e. the data production PEs are stopped once the data produced are sufficient to show meaningful results), and for those for which their input data are results from a query to the database. A second difference is the support of looping control structure. Service-based workflows are usually represented as data-flow graph, where the edges connecting the processing elements indicate their data-dependencies. A loop in the graph may indicate a feedback control of a particular PE with optimisation capability that can dynamically adjust its data production based on the feedback result, or a fault tolerant PE that is repeated until a desired output is produced. On the contrary, the edges of a task-based workflow represent the control-flow of the tasks, e.g. in a Pegasus DAX, where a task cannot be the parent or child to itself. These differences are discussed in MOTEUR[57] [76] and Taverna [134].

[57]MOTEUR: `http://modalis.polytech.unice.fr/softwares/moteur/start`

The efficiency of enacting service-based workflows is influenced by the factors below:

- *timing of binding*—abstract tasks need to be bound to concrete services for execution. Binding takes place during workflow composition, i.e. *early binding*, may yield a less optimal enactment (i.e. the information used to optimise the binding during composition stage are obsolete by the time it is executed), but binding which takes place immediately just before execution, i.e. *late binding*, may result in local optimal at individual services level but be less optimal due to lack of a global view of the whole workflow;

- *service discovery and selection mechanisms*—the execution performance is affected by the capability of the WMS to discover all of the available services and provide a selection algorithm to bind the optimal services. Walker *et al.* propose a dynamic service selection architecture based on realtime performance data [174].

- *data transfer mechanism*—the current technologies do not allow direct data transfer between web services (i.e. input data and results have to go through the workflow engine that coordinates their executions) and sending data as messages during invocation is not suitable for data-intensive situations. The former can be solved by using a data handler. Instead of sending data to the service, the orchestration engine sends a reference to the data, e.g. a URI to the subsequent web service, which is then dereferenced and retrieved from the data handler [134]. The solution to the latter is wrapping a data transportation protocol as a service to enhance the data movement across sites, e.g. using GridFTP actor in Kepler [125]. Glatard *et al.* propose a general strategy to reduce the overall data movement overhead, as well as other costs such as scheduling and queuing by grouping workflow services [74].

- *parallelism*—a common optimisation technique for workflow execution is exploring and exploiting parallelism opportunities. Firstly, tasks with no dependencies can be executed in parallel on multiple machines. This is defined separately in a few studies as task-parallelism [90], intrinsic workflow parallelism [75] or simple parallelism [142]. Secondly, independent data segments can be processed in parallel with multiple services, commonly known as data parallelism. Parallelism happens in both task-based and service-based workflows. We will discuss these parallelisms in detail in Section 2.6.3. In general, parallelism has been exploited in many service-based WMSs, such as MOTEUR[75], Taverna [134], Kepler [132] and Meandre [122].

2.5 Optimisation in other problem domains

In designing our optimisation algorithm, we learn from various optimisation strategies in other domains, such as in query processing. This section gives a short conclusion on how their approaches are useful and adopted in our design.

We first look at query processing. A query written by user in a high-level language is validated and translated by a *query optimiser* into a query execution plan or query evaluation plan (QEP), which is sent to *query execution engine* that execute the plan and return the result to user [39] (query evaluation techniques are discussed in [83] by Graefe). A query execution engine implements a set of operators that are the building blocks for operator trees, i.e. the representation for execution. The query optimiser is responsible for finding a QEP (from a set of logically equivalent QEPs) that minimise the performance measure, e.g. response time, CPU and I/O cost, and energy consumption. Based on the statistical data of the database, the query optimiser uses a cost model to estimate the cost of every QEP and the size of the data stream for output of every operator in the plan.

One of the main issues in cost estimation is the use of user-defined functions (UDFs), or stored procedures. UDFs are a powerful mechanism to allow users to incorporate application semantics in queries. It is very useful from both user and system perspective: the user can avoid to spend time on low-level programming language and the execution of UDF on the database engine yields better performance. Estimating the cost for a UDF is far more difficult than common operators, e.g. join, scan, aggregate and other arithmetic operators. Fomkin *et al.* profiled grouping methods for optimising complex scientific queries that uses UDFs [64]. This strategy first fragments the query into subqueries, named *groups*, based on application knowledge. Each group is then optimised independently and profiled for real execution on a sample of runs to measure real execution cost and cardinality. The profiled grouping method gives a significant improvement over a static cost model.

Moving towards the late '90s, new challenges for query processing arose when distributed databases became more feasible and demanded. For instance, new databases need to be integrated with legacy systems, the integration of multiple software modules needs access to multiple databases, and the demand for scalable and better performance in supporting new applications needs to be addressed. Distributed Query Processing (DQP) emerged to meet these requirements [127, 110]. A query is translated into fragment of QEPs, and then executed by more than one query execution engine. Reordering rules and parallelisation are extended to permit migration of operators along the data

tree, which will be partitioned and executed on distributed nodes.

Following the rise of service-based architectures in distributed computing, efforts are invested in integrating service-based architecture with DQP [127]. Relational operators are now mapped to web service instances to execute the QEP. Dobrzelecki *et al.* discuss the use of OGSA-DAI DQP in creating a "*federated*" database that integrates a set of distributed databases [55]. The streaming execution model of OGSA-DAI DQP provides implicit parallelism. We will discuss the streaming model and parallelism in the next section.

2.6 Data streaming model

The term "*data stream*" is widely used in many contexts and is becoming increasingly important. At the very low level, a data stream refer to a sequence of digital signals transmitted from a device; at the higher level, a data stream is used as an abstraction of a collection of data units in application programming. Data streams can be sent out from a sensor in real-time, and can also be continuously generated by a source, e.g. a database engine or a simulation program. In general, stream processing is used in the literature to describe a variety of systems, such as in data-flow analysis, logic programming, signal processing networks and embedded systems, as observed by Stephens [158]. This section discusses the various definitions of data streams and the streaming processing model, and establishes the purpose of pipeline streaming and parallelism in optimising workflow execution.

2.6.1 Data streams

There are several interpretations of data stream in the literature from different research domains and application areas, e.g. sensor networks, multimedia, database and file processing. We describe two of the most relevant interpretations below:

1. A sequence of digitally encoded signals used to represent information in transmission;

2. An ordered collection of data items $[s_1, s_2, \ldots]$, that has the following properties: (a) data items are continuously generated by one or more sources and sent to one or more processing entities, and (b) the arrival order of data items cannot be controlled by the processing entities that receive the values.

The first interpretation came from telecommunication science [102] and includes the data transmitted back from sensors that are placed remotely on a network, e.g. temperature, motion sensor and seismometer. Our research here is more related to the latter. The second interpretation is taken from the Encyclopaedia of Database Systems [78] to describe data extracted from databases, but this abstraction is also suitable for those data generated from modelling and simulation, data packets flowing through the network, events triggered on live monitoring systems, and live message feeds from social webs. Instead of digital signals, a data stream may be composed of relational tuples, raw data packets or pieces of text. The processing entities may be query operators, processing software of a network routers or sense-and-response stream-events processing systems (e.g. IBM System S [7], Microsoft StreamInsight [82] and Oracle Complex Event Processing [156]).

2.6.2 Streaming processing model

Stephens [158] defines a stream processing system as "a system that comprises a collection of *modules* that compute in parallel, and that communicate data via *channels*". The modules, also known as filters [166], operators [31], tasks [26], kernels [108] and actors [79] in other studies, are the basic processing units (we use "*task*" for the rest of discussion in this section). Tasks perform certain transformations on the input data streams received from one or more *upstream* tasks (i.e. predecessors), and send the results to the output data streams connected to one or more *downstream* tasks (i.e. successors). The transformation in each task is independent and self-contained, even though they have data-dependencies between them. Beside the basic transformation task, there are two other types of task: *sources* (pass data into the system) and *sinks* (pass data from the system). Tasks have no control over the order of the arrival of the data elements, nor the arrival rate. For a live-stream system, e.g. seismic activity monitoring and web logs analysis, data streams are potentially unbounded in size[58] and data elements arrive in real time [13].

The streaming processing model has a great capability to perform data-intensive computations. Large-scale data streams impose challenges on the infrastructure to *transmit* (i.e. sending the entire data into and from the system, and between the tasks), *compute* (i.e. processing large-volume of data using modest computing and memory capabilities) and *store* (i.e. temporary keeping the raw data and intermediate results, and archiving the derived results) the data. The underlying infrastructure does not have the luxury of

[58] A seismometer stops producing data when it is being switched off due to maintenance or failures, thus the seismic data is finite. However, when viewed on computational time scale, the data are continuously generated by a seismometer during the tasks' execution. So the data are unbound.

storing the entire dataset for analysis, but has to process on-the-fly with the use of one-pass algorithms [148], also known as single-pass algorithms and streaming algorithms. The concept of making a single pass over large-scale data for computation was first discussed in late '70s. Munro *et al.* [135] addressed the problem of searching and sorting data stored on a one-way, read-only tape with limited internal storage, and presented a single-pass algorithm for that purpose. Research in streaming algorithms received a boost following the growth of global networks [180]. A good survey of data streaming algorithms and application used in various domains, e.g. network traffic monitoring, text mining, and real-time streaming applications on the web, can be found in [136].

Another significant advantage of the streaming processing model is the capability of *parallel execution* where the tasks are independent of each other. The advancement of multicore architectures and high-speed communication networks has opened up the opportunity of executing stream tasks in a parallel and distributed environment. New software systems, e.g. Imagine [108] and Streamware [88], are developed to support the compilation and execution of streaming programs on multicore processors. The creation of high level language for streaming application (e.g. StreamIt [166]) and stream processing middleware (e.g. SPADE [70]) enables users to write applications that are automatically parallelised and mapped onto multiple computing resources and run in parallel. See Section 2.6.3 for more discussion on parallelism.

Our research adapts the streaming processing model in workflow enactments. PEs in the workflow are similar to the tasks in the stream processing system in which data are passed in a streaming manner between the PEs that are executed in parallel. The data streams in our model include live feeds from sensors, query results from databases, binary stream reads from file systems and data in various structural types. We will discuss our streaming model in detail in Chapter 3.

2.6.3 Parallelism

Studies have been conducted in both stream processing systems and workflow systems to explore methods of increasing efficiency through exploitation of parallelism. We adapt the characterisation found in [76, 79, 142], and broadly divide parallelism into three categories: task parallelism, data parallelism and pipelining.

Task parallelism refers to a set of tasks, which have no dependencies between them, and so can be executed concurrently on multiple parallel threads. This is the easiest type of parallelism that can be exploited by observing the branching in the data stream. Two tasks are said to have no dependency if the output stream of one of the task never

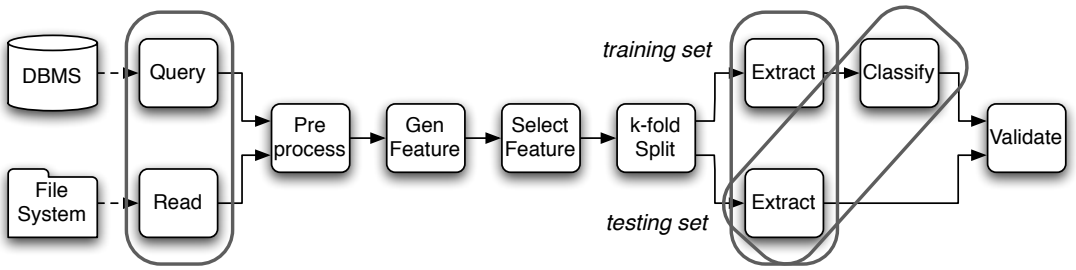


Figure 2.21: Task parallelism.

reaches the input stream of the other. We use the workflow shown earlier in Section 2.2 to aid the discussion. As illustrated in Figure 2.21, the workflow comprises two merging points and a splitting point. These pairs of tasks, i.e. [Query, Read], [$\mathsf{Extract}_{\mathsf{training}}$, $\mathsf{Extract}_{\mathsf{testing}}$] and [Classify, $\mathsf{Extract}_{\mathsf{testing}}$], are independent from each other and can be executed in parallel on multiple processors. Otherwise, tasks have to be executed sequentially, e.g. Preprocess and GenFeature. Task parallelism can improve the efficiency of executing both task-based and service-based workflows, provided if the deployment, communication and synchronisation (i.e. splitting and merging data stream) overhead are lower than the time gained from parallel execution.

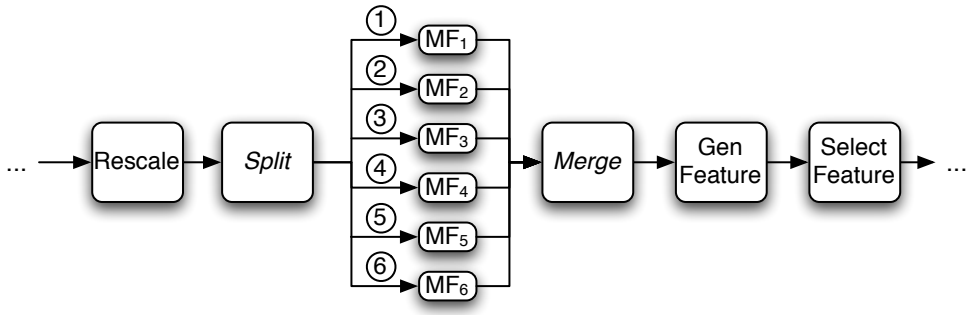


Figure 2.22: Data parallelism.

Data parallelism happens when data elements with no dependencies among them, are processed in parallel. From the control-flow perspective, data parallelism refers to tasks that have no dependencies between one execution and the next. This is a Single Instruction, Multiple Data (SIMD) type parallelism in the parallel architecture defined by Flynn [63]. Assume that the data elements (indicated as circles with number in the figure) are independent from each other and the Preprocess task comprises two separate sub-task, i.e. Rescale and MedianFilter (showed as MF in Figure 2.22), where the processing rate of MF is slower than the remaining tasks. Given a large number of computing resources, the task MF can be instantiated multiple times on different processors to process data elements in parallel. This is done by inserting a Split task

before MF to split the data stream into six parallel streams connecting six instances of MF. A Merge task is placed after them to merge the data streams back into a single stream and then continue with the subsequent tasks.

Data parallelism can reduced the overall make span of data-intensive workflow by speeding up the slow processing rate tasks in a workflow. This is very efficient way to execute *naturally parallel computations*[59] such as those that involve parameter sweeps and workflows with an iterative processing pattern on multiple datasets. The efficiency of data parallelism relies on two important factors: the partitioning mechanism and granularity of data. *Partitioning mechanism* concerns whether the partitioning of data elements is done statically (build-time) or dynamically (run-time). The static partitioning is easier to implement but lacks flexibility to cope with a fast-changing distributed environment. The partitioning done at build-time may not be the optimal choice for execution, where the number of computing resources may have changed. The latter requires the workflow engine to partitioning the data at run-time based on the available resources. For instance, Meandre[60] performs automatic parallelisation that creates multiple task instances according to the status of the underlying infrastructure. Pautasso and Alonso discuss the adaptive partitioning strategy for data parallelism used in executing workflows on a Grid and conclude that the number of data partitions should be a multiple of the number of available processors for optimum resource utilisation [142]. The downside of data parallelism is the increasing of buffering and latency, which is related to the *granularity of the data*. Based on the size of data elements, computation cost per data element, and the resource layout, fine-grained data parallelism may incur excessive communication and synchronisation overhead. Gordon *et al.* experimented with data parallelism with different granularity levels, and have showed that coarsening very fine-grained data parallelism can achieve a higher speed up [79].

Pipelining allows a sequence of more than one tasks to execute simultaneously on a vector of data elements. In general, pipelining refers to the chaining of a set of tasks that are executed in order, and the implementation can be streaming or non-streaming, i.e. a task starts execution when it predecessors has completed. Our discussion in this section is focusing on the former. In the pipeline, each task starts the processing as soon as it has received sufficient input data from its upstream tasks. As illustrated in Figure 2.23, SelectFeature starts the processing on the first data element once it received it from GenFeature, while Rescale is already processing the fourth data element. This overlapping of execution reduces the overall execution time by a factor proportional to

[59] It is commonly known as *embarrassingly parallelism* among the HPC community. However, there is no cause for embarrassment, thus a proper term, *naturally*, has been used lately.

[60] Meandre documentation: `http://seasr.org/meandre/documentation/for-developers/zigzag/`

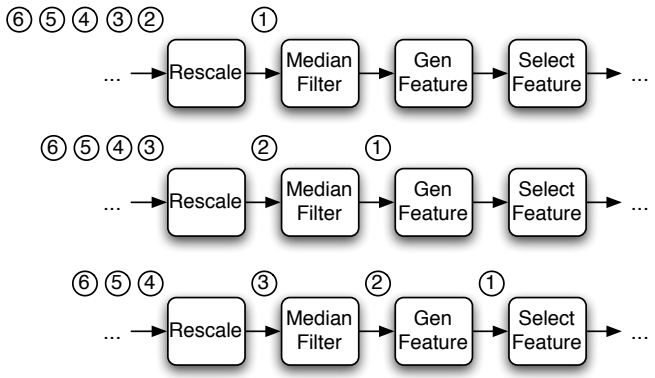


Figure 2.23: Pipelining.

the number of pipelined tasks, with the assumption that the tasks and data elements are homogenous and are executed on dedicated computing resources [142]. In reality, the unit cost, i.e. time to process a unit data element, of each tasks are very unlikely to be the same. Fast processing tasks wait for the slower ones. Blocking mechanisms are introduced to avoid data being discarded by upstream tasks when the downstream tasks are busy. For instance, Rescale stops sending the second data element to MedianFilter, which is still busy processing the first data element, and stops its execution on the next data elements. To further improve this inter-task synchronisation, a buffer queue can be placed between both tasks, just as the producer-consumer problem happens in inter-process communication. Another approach is adding more instances for the high unit-cost tasks to balance the processing rate, described as a Superscalar pipeline by Pautasso and Alonso [142]. A similar idea also been proposed by Gordon *et al.* [79] by combining pipelining and data parallelism to create multiple pipelines, as shown in Figure 2.24.

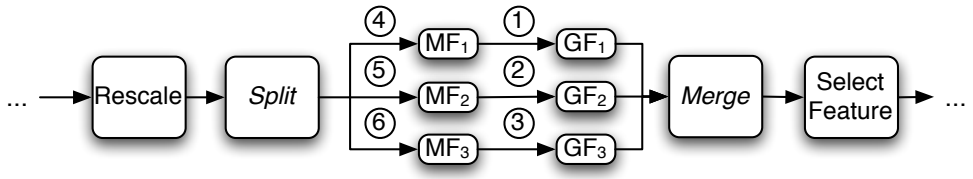


Figure 2.24: Pipelining and data parallelism.

Compared with data parallelism (shown in Figure 2.22), this "*parallel pipeline*" execution on heterogenous environments yields a lower communication overhead. However, it is difficult to determine the optimum point for "*fan-out*" (split) and "*fan-in*" (merge) in a workflow. These decisions are not only influenced by the unit cost of the tasks, but also by the volume of the data flowing between them.

2.7 Summary of related work

In this chapter, we have discussed the diversity and complexity of the challenges faced by the scientific community and explained why they can not be solved by a simple unification of technologies. We then looked into the data deluge challenge and how WMSs are used to facilitate the execution of scientific experiments on a distributed environment. We have reviewed five well-known WMSs based on the requirements and classifications defined in an earlier section.

Based on our observation, we conclude that:

1. Abstraction holds the key for supporting the diversity of both applications and execution infrastructure. A workflow language that allows separation of concerns is needed.
2. Centralised workflow orchestration meets a performance bottleneck when processing data-intensive applications. A hybrid model that retains a centralised orchestration controller locally with choreography-style workflow propagation capability across organisational and geographical boundaries is crucial.
3. The exploratory nature of scientific experiments and increasing use of WMSs for data-intensive applications demand that the workflow optimisation be automated. Hand-crafted optimisation is not a durable solution.
4. Performance data from previous executions can be used for optimisation as scientific users tend to repeat their workflows.
5. Stream processing methods can process data-intensive workflows efficiently.

We will discuss our optimisation model in the next chapter. A streaming workflow model will be presented, followed by the discussion about the definition, requirements and context of the optimisation problem. We will then present our mapping algorithm that makes use of performance data.

CHAPTER 3

Optimisation model

In the previous chapter, we have discussed the diversity, complexity and data deluge challenge faced by the scientific community and discussed how WMSs are used to facilitate the execution of scientific experiments on distributed computational environments. We have briefly described the workflow life-cycle and relevant workflow optimisation work. This chapter will focus on our optimisation model.

We first look at the fundamental concept of the workflow that we are working on by clarifying our streaming workflow model in Section 3.1. This is followed by the definition of the optimisation problem in Section 3.2 by identifying two sub-problems: graph transformation and resource mapping. Our work focus on the latter. Before we present our proposed mapping algorithm, we will go through the list of goals, requirements and context of the optimisation in Section 3.3. In Section 3.4, we describe the conceptual model for our mapping algorithm. This section discusses our understanding of the cost model of streaming workflow and our approach to optimise the mapping. A potential three-stage mapping algorithm is then presented in Section 3.5. The last section presents a summary of this chapter.

3.1 Streaming workflow model

A typical workflow comprises a sequence of tasks that represent steps in a computational process that composes data and operations that may be independently defined. A workflow can be control-driven or data-driven: the former has dependencies to show the execution ordering or control flow of the workflow, while the latter represents the flow of data from one task to another. Workflow structure is also different according to

workflow engine, scheduling method and enactment platform. We focus on workflows where the data flow is explicit represented as a directed acyclic graph (DAG) and the control flow implicit. The vertices, V denote the processing elements (PEs) that perform computational tasks, and the edges, E represent the data flow between the PEs which are implemented as stream. Figure 3.1 illustrates a partial workflow that comprises 9 tasks $V = \{PE_i, PE_j, PE_k ... PE_z\}$ with edges representing the data flow.

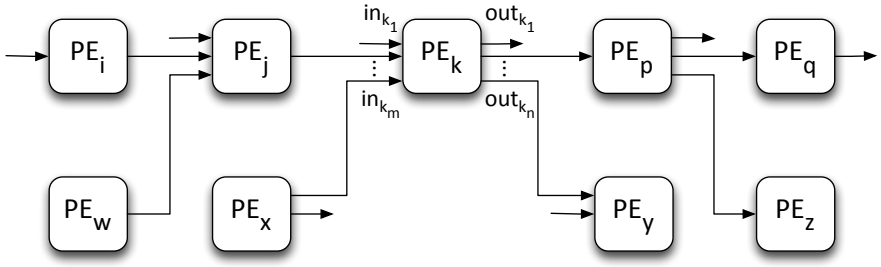


Figure 3.1: A DAG comprises PEs with different number of inputs and outputs.

In general, PEs have *input(s)* to receive data from preceding PEs, process the data, and send results to the succeeding PEs through the *output(s)*. For example, $\mathsf{PE_i}$ processes data and passes the output data to the succeeding $\mathsf{PE_j}$. Each PE can have zero, one or many inputs, as well as outputs, as shown in Figure 3.1. $\mathsf{PE_i}$ and $\mathsf{PE_q}$ are typical of the type of PEs found in streaming workflows, which read data from an input stream and produce data for an output stream. $\mathsf{PE_j}$, $\mathsf{PE_k}$ and $\mathsf{PE_p}$ illustrates the variations in terms of number of inputs and outputs. $\mathsf{PE_w}$ and $\mathsf{PE_x}$ are another type of PEs that do not have predecessors but produce data for the following PEs. We named them *source PEs*. An example is a PE that reads data from a relational database[1]. The final type of PEs read data from input stream but do not output data to successors, we named them *sink PEs*, e.g. $\mathsf{PE_y}$ and $\mathsf{PE_z}$. These PEs are commonly found in workflows which mainly handling data delivery, e.g. store data to a file system or return final results to a user.

Characteristics of PEs can be summarised as below:

- PEs have *input(s)* to receive data and *output(s)* to send data, except source PEs and sink PEs, which obmit inputs and outputs respectively.
- PEs have different data processing rates, e.g., $\mathsf{PE_i}$ and $\mathsf{PE_w}$ may take different amounts of time to transform a unit of data.
- PEs may have different input consumption rates, e.g., if $\mathsf{PE_k}$ is a sort merge, it may consume data from one input much faster than from the other.

[1]It has no predecessors, but usually takes a string literal, i.e. query expression, to specify which data is required.

- PEs start to process as soon as they have received sufficient data for the computation. They may emit data as soon as the processing on a unit of input has finished.
- Some PEs are *aggregative*, that is, they combine data from a sequence of units in their input to produce a single derived value in its output, e.g. $\mathsf{PE_q}$ can be a mean calculator that reads all of the value from $\mathsf{PE_p}$ and produces a single output, i.e. the mean of the input stream.
- The relationship between inputs and outputs may be specified. For instance:
  a) that a PE consumes lists from its input and generates a tuple for each list on its output that is an aggregation of the list,
  b) that a PE takes lists of tuples on its input and emits corresponding lists of tuples with the tuples partitioned between the outputs, or
  c) that a PE takes lists of tuples on input a and tuples on input b consuming one list and one tuple at each step and emitting a list of tuples which are the original tuples from a extended by the tuples from b.
- A PE ceases processing when an input it requires has signalled it has no more data or when all of its consumers have indicated they no longer require data, or when it is sent a stop signal by the enactment system.

PEs are connecting via *data streams* to form a DAG. Due to different consumption rates of PEs, *blocking mechanisms* and *buffering* are needed in a data stream (⇝*2.6.3*). The buffers can be implemented in main memory or spill onto disk. When a data stream connecting two PEs resides on separate machines, the stream implementation uses communication protocols.

One of the important characteristics of the streaming model is the PEs are connected in a pipeline which allows task executions to overlap. Moreover, PEs have different data processing rates and input consumption rates, which incurs the demand for large buffers in the data streams and may cause deadlock during workflows enactment if they have hit the buffer limit (for PEs with multiple inputs). All of these factors contribute to the difficulty of enacting a streaming workflow.

Before a PE can be used as a workflow component, an instance of that PE must be created. A PE can be instantiated many times, and each of these instances is referred to as a processing element instance (PEI). A PEI is a concrete object used by the enactment engine while assigning resources.

We define a Data-Intensive Virtual Machine (DIVM) as an abstraction for the computational environment in which a PEI runs during enactment. Figure 3.2 shows a DIVM

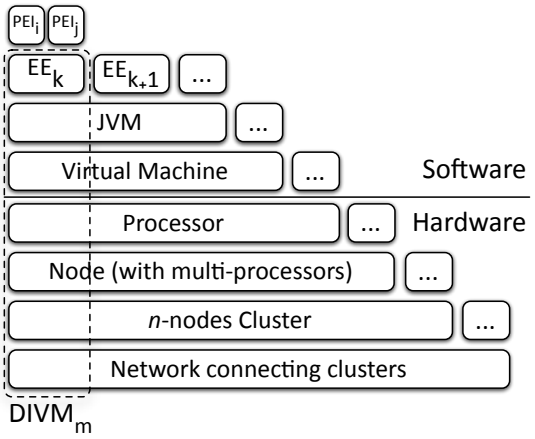


Figure 3.2: DIVM abstraction.

which is an abstraction of the layers of software and hardware. As a typical operational environment is complex, deriving relevant information, e.g. whether two PEIs are competing for the same CPU, is difficult. The DIVM abstraction is used to reduce the complexity by suppressing detail, so that queries on the performance database (PDB) can discriminate such conflict criteria (⇝*4.4*). This abstraction also has to reflect locality so that relative costs of inter-PEI communication can be estimated using the PDB.

3.2 Optimisation problem

Optimisation happens in the mapping phases of the workflow life-cycle, as shown in Figure 3.3. The abstract workflow produced in the workflow composition phase is parsed by a language processor to generate a graph, which will go through a series of *transformations* into a fully expanded and annotated graph of PEIs. The next step is to *map* these PEIs on the DIVMs. The output of the mapping phase is a set of concrete workflows which define where to deploy PEIs, i.e. on which DIVMs each should run (see Section 4.2.2 for a detail discussion of DISPEL enactment).

The performance of the execution relies on various factors and decisions made during the mapping phase. For instance, in a typical classifier-training workflow, 90% of the dataset are used for training a classifier and the remaining 10% will be used to evaluate the trained classifier, as illustrated in the top-right corner of Figure 3.3 (the dataset is split into training and testing pipelines with $\mathsf{PE}_{\mathsf{split}}$ and merged back with $\mathsf{PE}_{\mathsf{eva}}$). The testing pipeline will finish faster than the training pipeline. Thus, the optimiser needs to speed up the training pipeline by performing certain transformations on the

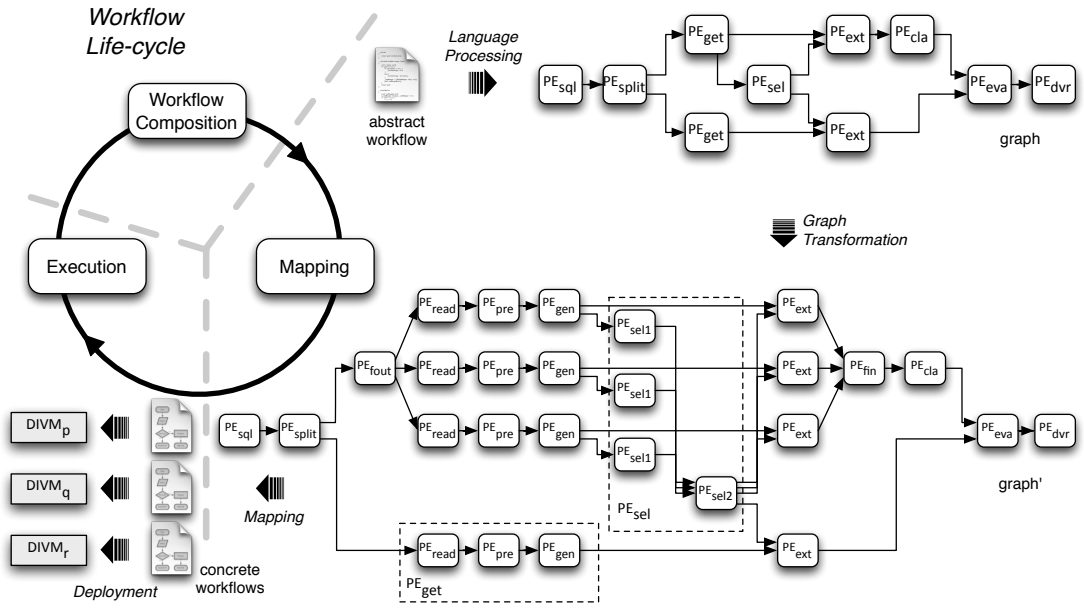


Figure 3.3: Mapping phase in workflow life-cycle.

graph, e.g. parallelises the pipeline. Then, all of the PEIs need to be mapped onto the appropriate execution engines, in terms of capability, workloads, and availability, in order to achieve maximum efficiency. We divide the optimisation task into two parts: graph transformation and DIVM allocation.

3.2.1 Graph transformation

The graph generated in the DISPEL *Language Processing* stage will go through a series of transformations as a result of optimisation. This process is conducted repeatedly until a final graph, *Graph′* is produced, which is ready for *deployment* stage (see Figure 3.3). In this section, we further examine four types of common transformations: *sub-graph substitution*, *parallelisation* and *reordering*.

During the optimisation stage, the optimiser tries to identify candidate implementations for every abstract PE (nodes in the abstract workflow). An abstract PE can be mapped to a single physically located PE instance, or a group of implemented PEs, if it is a composite PE. In the latter case, the PE node will be *substituted by a sub-graph* of PE nodes. This process continues until all concrete PEs are selected. Assume that the feature selection PE, $\mathsf{PE}_{\mathsf{sel}}$, used in the example shown above has two implementations: sequential and parallel. The parallel implementation is defined using composition of two implementable PEs: $\mathsf{PE}_{\mathsf{sel1}}$, which performs the partial standard deviation calculation and $\mathsf{PE}_{\mathsf{sel2}}$, which combines the results from all of the $\mathsf{PE}_{\mathsf{sel1}}$ instances and performs the Fisher Ratio calculation to select the most significant features set (⇝*5.1.1*). If more

resources are allocated for this enactment, the optimiser should choose the parallel implementation of the $\mathsf{PE}_{\mathsf{Sel}}$, which is expanded into a sub-graph of PEIs.

The optimiser also improves the enactment performance by exploring *parallelisation* opportunities. To speed up a slow-processing rate subgraph, the optimiser looks for parallel-executable PEs and splits the data streams. This parallelisation is categorised as data parallelism (⇝*2.6.3*), where a data stream is split into multiple streams for parallel execution. As discussed earlier in this section, the execution of a classifier-training workflow can be improved by splitting the training pipeline and executing it in parallel. In Figure 3.3, the optimiser inserts a $\mathsf{PE}_{\mathsf{fout}}$ and duplicates the feature generation pipeline, i.e. [$\mathsf{PE}_{\mathsf{read}}$,$\mathsf{PE}_{\mathsf{pre}}$,$\mathsf{PE}_{\mathsf{gen}}$] to speed up the processing of training dataset. The main concern is for the optimiser to find the optimum number of pipelines, the right granularity of the data and the most suitable splitting and merging points in the graph.

Reordering is about transposing the PE's order based on a QoS metric. It follows the approach in database optimisation where the reordering is guided by a set of rules based on relational algebra. For instance, placing a filtering operator before a projection operator may reduce the computation cost of performing these operators in reverse, where the filtering operator reduces the amount of data to be processed by the projection operator. In order to support reordering PEs, the optimiser needs to obtain semantic information from the registry which shows whether these PEs are transposable.

3.2.2 DVIM allocation

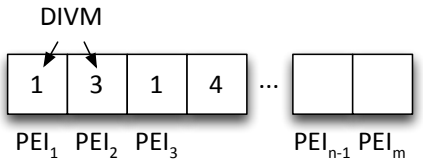


Figure 3.4: Allocating PEIs onto DIVMs.

Given a set $P = \{PEI_i\}$ of PE instances with different workload and transfer requirements, and a set of $D = \{DIVM_j\}$ of available DIVMs for the enactment with computation and communication capacity, we need to find the optimum assignment by searching among a list of potential mapping candidates, as shown in Figure 3.4. Let $|P| = m$ *and* $|D| = n$, then the number of *possible assignments*, $A(P, D)$, is n^m. This mapping is a DAG scheduling problem in distributed systems [37] which is *NP-complete* in general [61, 69], where the optimal mapping can be found with exhaustive search. Many heuristic-based techniques have been proposed over the years [30, 101, 167]. On

top of the distributed and heterogenous complexity demonstrated in these works, we are facing a bigger challenge dealing with streaming workflows.

In a streaming model, there is an overlap between PEIs execution, where each PEI processes a portion of the data stream. This overlapping behaviour[2] in the PEIs execution complicates calculating the total execution time of the DAG. Hitherto, based on our survey of the related work, there is no existing model that is appropriate for our streaming workflows. One of the possible approaches is to model the *time to produce an element of the final result*, and think of this as a DAG scheduling problem.

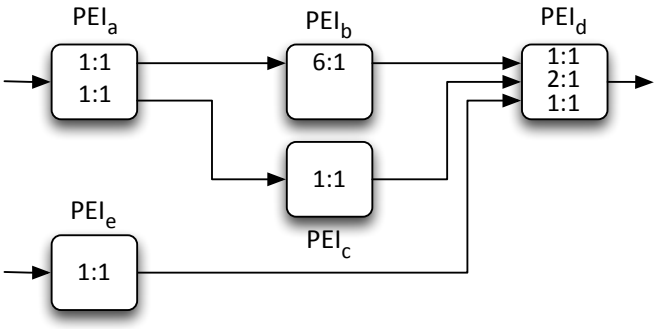


Figure 3.5: Example DAG of streaming workflow.

We use the DAG shown in Figure 3.5 to discuss this approach. Each PEI processes input elements from preceding PEI(s) to produce output elements for the succeeding PEI(s). For instance, $\mathsf{PEI_b}$ takes 6 input elements from $\mathsf{PEI_a}$ to generate 1 output element, which will be consumed by $\mathsf{PEI_d}$. The processing of output data is started as soon as the input elements read from a preceding PEI are sufficient for generating an output element. In other words, $\mathsf{PEI_b}$ starts processing once it has received 6 input elements from $\mathsf{PEI_a}$, while $\mathsf{PEI_a}$ still processing the remaining data—there is an overlap between the two PEIs.

We define a *job*, $Job_{i,j}(k)$ as the processing of the $\mathsf{PEI_i}$ to generate a data element on output j and k is an element counter $\{1, 2, ...\}$[3], e.g. $Job_{a,1}(1)$ is the execution of $\mathsf{PEI_a}$ to produce the first element on output 1. We then define a dependency graph DG between all of the jobs $Job_{i,j}$ according to the element flows along the data stream, as shown in Figure 3.6. For instance, the processing of the first output element of $\mathsf{PEI_d}$, $Job_{d,1}(1)$ is dependent on the output of $Job_{b,1}(1)$, $Job_{c,1}(1)$, $Job_{c,1}(2)$ and $Job_{e,1}(1)$ which has dependencies on other jobs accordingly.

The jobs will be scheduled to available DIVMs. Figure 3.7 illustrates a scheduling

[2]Which is intended to allow multiple PE steps to work on data while they are close to the processors in the execution engines' memory hierarchy.

[3]We use 1 as the first element to match the notation in the diagrams

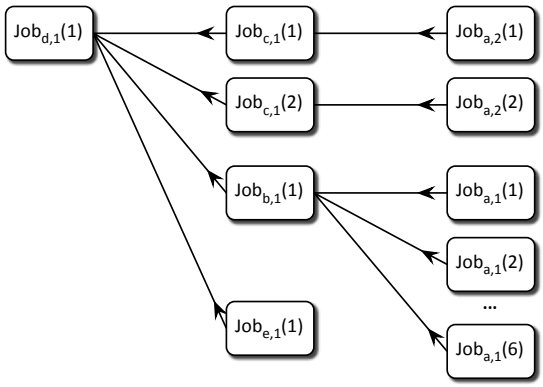


Figure 3.6: Dependency graph for individual processed items of the jobs.

candidate of the DG on 2 DIVMs: $\mathsf{DIVM_i}$ and $\mathsf{DIVM_j}$. Jobs for $\mathsf{PEI_a}$, $\mathsf{PEI_b}$, $\mathsf{PEI_e}$ and $\mathsf{PEI_d}$ are scheduled to $\mathsf{DIVM_i}$, and the remaining to $\mathsf{DIVM_j}$. However, this scheduling requires moving data elements between these DIVMs, which incurs additional *communication time*. To model the communication time, we consider the data movement as separate jobs that can only be executed in an abstract DIVM, *transport DIVM*. We propose a transport DIVM, $\mathsf{TM_{s,d}}$ to move an element of data from $\mathsf{DIVM_s}$ to $\mathsf{DIVM_d}$. For instance, a data movement job $a_{2,1}$ is added to $\mathsf{TM_{i,j}}$ to move the output element of $Job_{a,2}(1)$ from $\mathsf{DIVM_i}$ to $\mathsf{DIVM_j}$ for $Job_{c,1}(1)$. In terms of dependency, $a_{1,3}$ must scheduled after $Job_{a,1}(1)$, and before $Job_{c,1}(1)$ without any overlap.

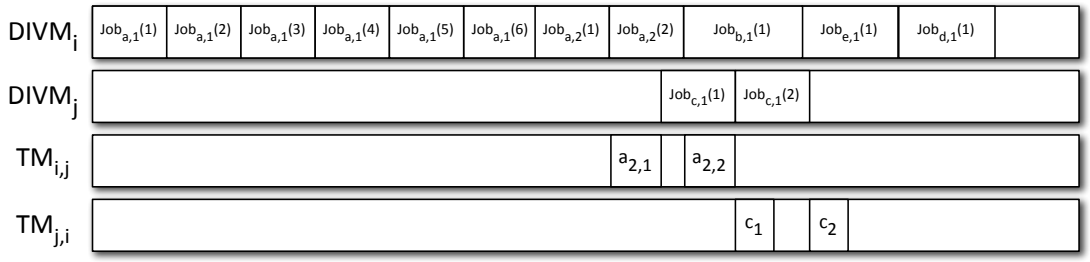


Figure 3.7: Scheduling of jobs defined in Figure 3.6 on 2 DIVMs.

Then we can define the optimisation goal as: to *minimise the time to produce a unit of result* (a unit of output element of the PEI). In other words, if $T(DIVM_i)$ is the time spent in processing all of the jobs scheduled in $\mathsf{DIVM_i}$ to produce the first output, we want to minimise the $max(T(DIVM_i))$, for all $1 \leq i \leq$ *total number of DIVMs*. There are two main factors to determine the $max(T(DIVM_i))$: the number of DIVMs used and data movement cost. We can minimise $max(T(DIVM_i))$ by scheduling independent jobs on multiple DIVMs. However, the gain from increasing the number of DIVMs is offset by the data movement costs that arise from the distributed executions. We have to find a trade-off between the two factors.

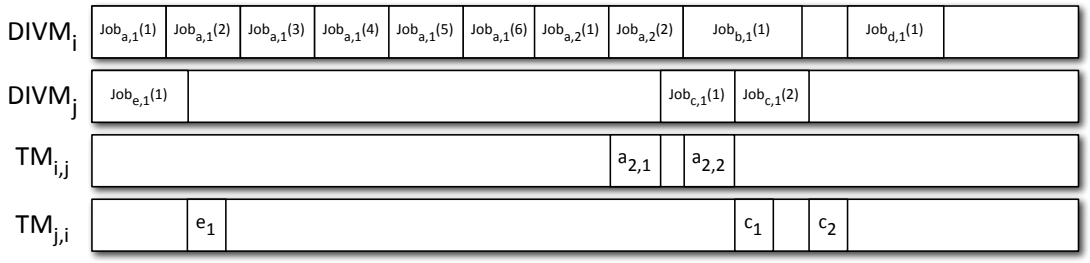


Figure 3.8: Scheduling of jobs defined in Figure 3.6 on 2 DIVMs (another candidate).

Figure 3.8 shows another scheduling candidate that reduces the $T(DIVM_i)$ by scheduling $Job_{e,1}(1)$ in $\mathsf{DIVM_j}$ prior to $Job_{c,1}(1)$, seeing that $Job_{e,1}(1)$ is not dependent on other jobs. This scheduling incurs additional communication cost on moving the result of $Job_{e,1}$ to $\mathsf{DIVM_i}$. In the above example, the communication cost of moving data between DIVMs is smaller than the jobs' processing time, thus, it does not lengthen the $T(DIVM_i)$. In contrast, scheduling *small jobs* that involve *large data* may result in a longer $T(DIVM_i)$.

The DAG scheduling method is valid with the presence of constraints below:

- all of the jobs from the same PEI must be scheduled on the same DIVM and in the order given by k in $Job_{i,j}(k)$;
- precedence of data must be obeyed (jobs must executed in partial order, i.e. output from $\mathsf{Job_a}$ being needed as input $\mathsf{Job_b}$ implies $\mathsf{Job_a}$ runs before $\mathsf{Job_b}$);
- all of the PEIs are allowed to overlap their execution, but this is NOT necessary;
- the jobs are not preemptive; and
- each DIVM is executing one job at a time (even though each DIVM is scheduled multiple PE instances during enactment).

There are few problems with this approach. First, the definition of job time for conventional job scheduling is different, where we can calculate the estimated execution time of jobs. In the streaming model, we can only calculate the unit cost, but the execution cannot be estimated because the data size is unknown. The second problem is the DIVM abstraction. The assumption of having each DIVM to execute one job at a time is not feasible with the prevalent of multi-processors and multi-cores architecture. Moreover, it is hard to model the suggested abstract DIVM for data movement, i.e transport DIVMs. The communication time between DIVMs that may run on different configurations, e.g. same JVM, same processor but different JVMs, and distinct computing nodes across network. The third problem is to find the optimal mapping in a large search space is intractable in practice. Thus, we have to investigate a way to reduce the search space.

3.3 Goals, requirements and context

Optimising both graph transformation and PEI-to-DIVM mapping at the same time is a big challenge, and it is difficult to solve within our time and resource constraints. Thus, our optimisation research is focusing on the mapping problem. Having said that, we have conducted a preliminary experiment that exploits parallelism in streaming workflow as a proof-of-concept for optimisation with graph transformation (⇝*5.3.2*).

3.3.1 Goals

Given a fully expanded and transformed graph that comprises a set $P = \{PEI_i\}$ of PE instances, and a set of $M = \{DIVM_j\}$ of available DIVMs for the enactment, the optimisation goal is to find an optimal assignment of P onto M, that minimises the overall *makespan*[4] of the enactment. The assignment attempts to achieve:

- a balance-computational workload distribution among the DIVMs, and
- minimum communication cost of data movement between DIVMs.

Both these criteria conflict with each other in practice. Assigning all of the PEIs onto a single DIVM yields minimum data movement cost but imbalances the workload that further increases the overall makespan. In contrast, having too many DIVMs in the enactment will incur large communication cost. Finding the right number of DIVMs requires substantial work and is beyond the scope of this research. Besides, there may be some capability and accessibility constraints that restrict the execution of certain PEIs on the selected DIVMs. Thus, the mapping algorithm should find a balance point between these criteria.

3.3.2 Requirements and context

To perform such an assignment, the optimiser must obtain three important pieces of information. First, the optimiser needs the the *descriptions of components*, i.e. the semantic descriptions of both PEs and data sources. For PEs, the information should include the properties of every PE (e.g. name, version, input/output data type and data rate), the implementations of the PE and its location, i.e. on which DIVM[5]. For data sources, the descriptions should include the logical name of the data source and the physical location where it has been stored, i.e. the URI of the DIVM. These descriptions

[4]Time difference between the start and finish of the enactment.
[5]For those PEs with capability and accessibility restrictions.

are provided by the domain and data-analysis experts who designed and implemented them, and are stored and organised in the registry (⇝*4.1.1*).

The second information is the *performance of the component instances* gleaned from observing previous enactments, e.g. processing time per unit of data (⇝*3.5*) and memory footprint. We have discussed the exploratory nature of scientific experiments in Chapter 2, where users repeat similar requests over similar data as they iterate their understanding or process various samples in the exploration of variants and experimentation settings. The understanding of the previous enactment behaviour of the components is crucial for the mapping process. The performance data are captured by the measurement framework and stored in the performance database (PDB).

The last piece of information is the *descriptions of the DIVMs*. Each gateway maintains the current status of its resources in a resource catalogue.

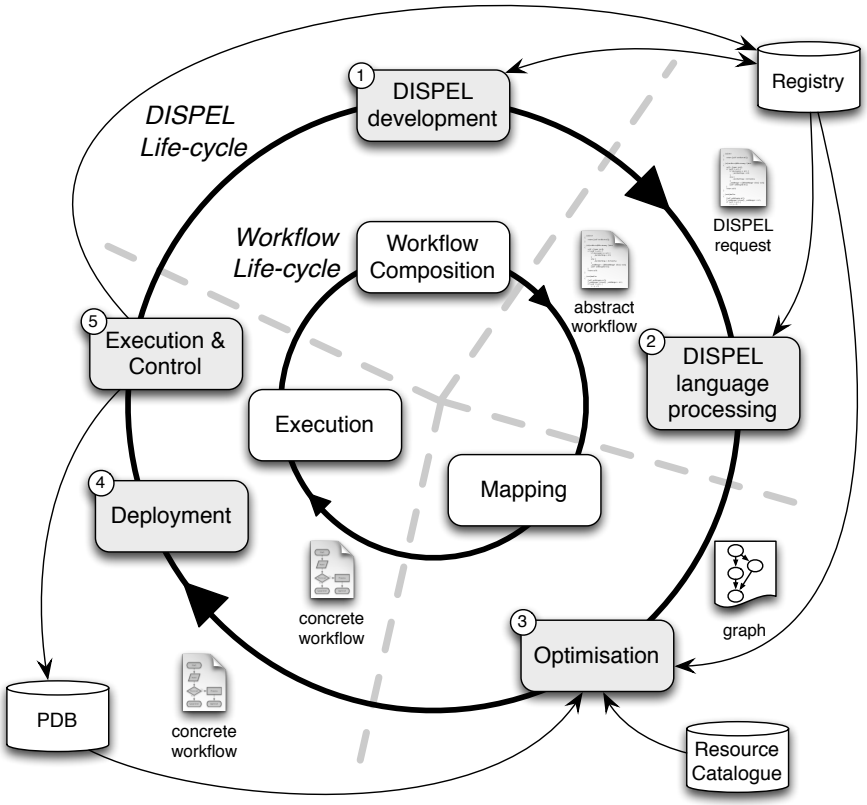


Figure 3.9: Information infrastructure for workflow life-cycle.

In Section 1.4, we gave a preview of the data-intensive architecture and briefly described the various phases of the DISPEL life-cycle. Our optimisation work is part of a research project that has designed this architecture, and implemented a prototype that comprises all of the modules needed in supporting the life-cycle, as shown in Figure 3.9.

Below is a list of the modules that are supporting our work:

1. A DISPEL development environment that provides the tools and workbench to develop DISPEL workflows.
2. A DISPEL language processor that parses a DISPEL request (abstract workflow) and generates a graph of PEs.
3. A registry that manages the semantic descriptions.
4. The underlying data-intensive platform that accepts the DISPEL request and manages the resource catalogue (gateway), and organise the enactment (enactment engine).
5. Execution engines that are used for deployment and execution of PEIs.

The data-intensive architecture and the DISPEL enactment process will be discussed further in Chapter 4. Now, we will look into the mapping algorithm.

3.4 Conceptual model for mapping algorithm

Finding the scheduling candidates of PEIs on DIVMs involves exhaustive computation. This is especially taxing for large workflows that comprise many PEIs. To reduce this cost, we reduce the search space of potential mappings by studying the behaviour of PEIs to partition them into different subsets, and then apply different allocation techniques to each subset.

3.4.1 Definition of unit cost

Before we discuss the partitioning process in detail, we first look at the definition of *unit cost* in our context.

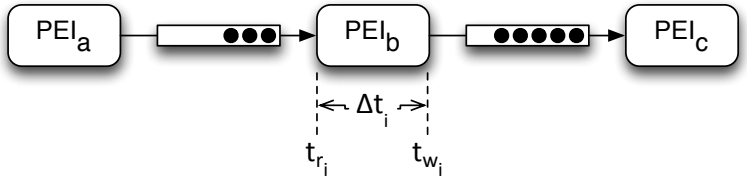


Figure 3.10: Streaming model of three PEIs.

Figure 3.10 shows timing data as observed during the enactment. There are three PEIs used in this workflow, which are connected via two data streams. PEI$_b$ *reads* data,

i.e. r_1, r_2, ... from the buffer space in the data stream $ds_{a,b}$, which is connected with its predecessor, $\mathsf{PEI_a}$. $\mathsf{PEI_b}$ then *writes* the results, i.e. w_1, w_2, ... into the data stream $ds_{b,c}$ for the succeeding PEI, $\mathsf{PEI_c}$. We define t_{r_i} as the timestamp of $\mathsf{PEI_b}$ reading a unit of data item from the stream buffer, and t_{w_i} as the timestamp captured when a write is performed. The interval between t_{r_i} and t_{w_i}, Δt_{p_i} is classified as the time to process a unit of data (also known as as *unit cost*) for a PEI. Thus, for PEIs with single input and single output, we can derive the unit cost by subtracting t_{r_i} from t_{w_i}. Based on the number of input(s) and output(s) of PEIs, we define the unit cost of processing the ith data item, t_{cost_i} as below:

$$t_{cost_i} = \begin{cases} t_{w_i} - t_{r_i} & \text{for PEI with single input and output} \\ min(t_{w_i}) - max(t_{r_i}) & \text{for PEI with multiple inputs and outputs} \\ max(t_{r_i} - t_{r_{i-1}}) & \text{for PEI with input(s) only} \\ max(t_{w_i} - t_{w_{i-1}}) & \text{for PEI with output(s) only} \end{cases}$$

where $max(t_{r_i})$ is the timestamp when PEI read the *last* ith unit of data from one of the inputs before the processing, and $min(t_{w_i})$ is the timestamp when PEI wrote the *first* ith unit of data to one of the outputs after the processing. A typical example is a $\mathsf{PEI_m}$ that reads a data item from two data streams (i.e. $\mathsf{in_1}$ and $\mathsf{in_2}$), merges into single data item and writes into output data stream (i.e. $\mathsf{out_1}$). Assume that $\mathsf{PEI_m}$ read the first data item from $\mathsf{in_1}$ then read the first data item from $\mathsf{in_2}$ before start producing the first output, the read time for $\mathsf{in_2}$ should be used in calculating t_{cost_1}. We then calculate average unit cost for $\mathsf{PEI_m}$ that process n units of data as $\frac{\sum_{i=1}^{n} t_{cost_i}}{n}$.

Due to the difference in the data consuming and data processing rates among the PEIs, there are another two monitoring observations captured by the measurement framework (*⇝4.3*): t_{rb_i}, which indicates a read attempt by the succeeding PEI is being blocked due to no data in the data stream, and t_{wb_i}, which indicates a write attempt by the preceding PEI is being blocked due to the buffer space in the data stream being full. An important remark here is the i in both t_{r_i} and t_{w_i} are a continuous sequence of integers, but not in both t_{rb_i} and t_{wb_i}. Not every Write attempt will trigger a WriteBlocked event, and the same intermittence applies to Read and ReadBlocked. Now, the calculation of Δt_{p_i} becomes more complicated. We try to visualise these events with a timeline.

Figure 3.11 shows the events traced in $ds_{a,b}$ and $ds_{b,c}$. The w_i and wb_i events on $ds_{a,b}$ indicate that PEI_a is writing a unit of data into the data stream, whereas r_i and rb_i events are triggered by a read from PEI_b. We assume that all of these events are captured by the measurement framework. To determine the Δt_i for a particular PEI,

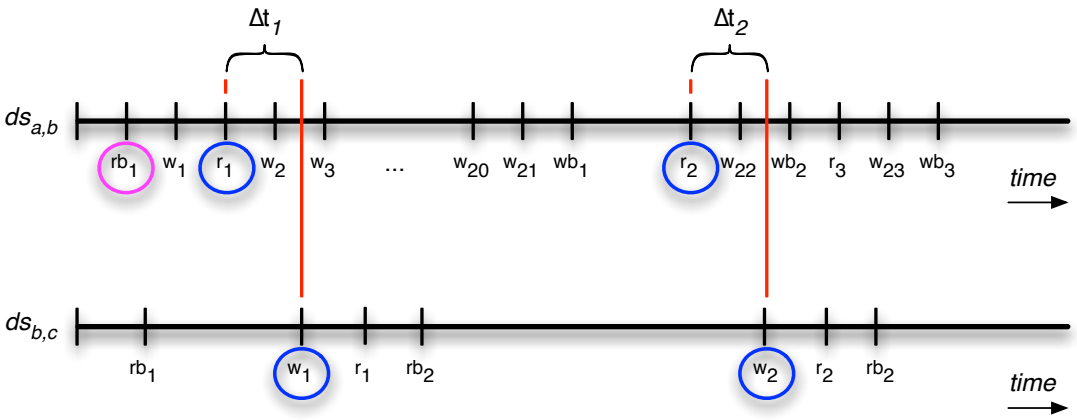


Figure 3.11: Events traced in both data streams in Figure 3.10 (part 1).

we need all of the data streams that are connected with it. In this scenario, the r_i and rb_i events on $ds_{a,b}$, and the w_i and wb_i events on $ds_{b,c}$ are related to the Δt_i of $\mathsf{PEI_b}$.

Now we look at Figure 3.11 to understand the events that have occurred. At the very beginning, $\mathsf{PEI_b}$ tried to read a unit of data from $ds_{a,b}$, but its attempt has been blocked. $\mathsf{PEI_b}$ only manage to read the first unit after $\mathsf{PEI_a}$ wrote the 1st data unit into the stream. To calculate Δt_1 for PEI_b, we subtract t_{r_1} (on $ds_{a,b}$) from t_{w_1} (on $ds_{b,c}$). The r_1 event is used in the derivation of Δt_1, instead of rb_1 event is to distinguish the waiting time from the processing time of $\mathsf{PEI_b}$. Thus, $\Delta t_1 = t_{w_1} - t_{r_1}$.

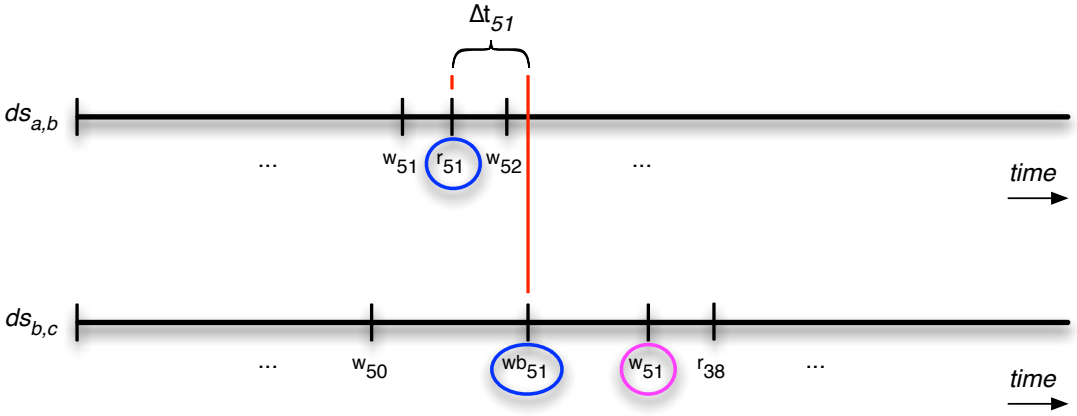


Figure 3.12: Events traced in both data streams in Figure 3.10 (part 2).

However, the situation is different with w_i and wb_i events. Figure 3.12 shows the second part of the events traced in $ds_{a,b}$ and $ds_{b,c}$, where $\mathsf{PEI_b}$ attempts to write into the $ds_{b,c}$ but it was blocked, recorded as event wb_{51}. The write attempt was eventually successful and is recorded as event w_{51}. In the calculation of Δt_{51}, $t_{wb_{51}}$ should be used instead of $t_{w_{51}}$ because the former is the exact time when $\mathsf{PEI_b}$ has finished the processing of data item 51.

3.4.2 Partitioning PEIs

From our observation on the preliminary experiment (⇝*5.3.1*), the PEIs can be generally divided into two categories. The first group of PEIs are more compute intensive and therefore have a higher unit cost. We named this group *heavy PEIs*. The other group of PEIs have a relatively small unit cost, i.e. *light PEIs*, and the assignment of these PEIs onto any of the DIVMs may not impose a significant workload. These are mostly utility PEIs that perform simple transformation or filtering on a data stream, e.g. type conversion PEIs. Most of the PEIs have a relatively small unit cost and a few have much larger unit costs. Therefore, to achieve the first optimisation goal, i.e. balance distribution of workload among DIVMs, we focus on the assignment of heavy PEIs based on their unit cost.

The main research question here is how to define the *performance threshold*, which divides the PEIs into these two categories. First, the performance threshold can be be determined from experiments. For those PEIs that have been used before, this can be observed from their previous enactment data stored in the PDB. A learning algorithm can be used to incrementally improve the threshold value. For those newly implemented PEIs, a sampling technique can be applied to do a trial run on a smaller dataset to obtain their unit cost. Second, the performance threshold should be domain specific. The threshold value defined for seismology applications is different from cell-biology workflows because the former may have more heavy PEIs that perform joining operation and compute-intensive calculation where the latter may have fewer heavy PEIs to handle the major transformation. Third, the performance threshold should be site specific, e.g. the number and architecture of the execution engines, the computational power of each processing node, and the network bandwidth. Setting a low threshold for the enactment of a small scale workflow on a large number of DIVMs will result in larger communication cost because the PEIs are spread widely across DIVMs.

The light PEIs may be neglected in terms of the computational cost. However, simply assigning these PEIs may incur a large communication cost on the enactment. Figure 3.13 shows a DAG with 12 PEIs, which are mapped onto 3 DIVMs. PEIs coloured in blue are categorised as heavy PEIs and are distributed evenly over DIVMs. The data stream connecting PE_1 and PE_2, $ds_{1,2}$ is buffered in memory. Passing a data item between these two PEIs involves only passing references because both PEIs are executing within the same runtime environment. When data are passed across DIVMs that are executed on different runtime environments, the data need to be serialised and then sent using a selected network protocol and de-serialised. If there are security and privacy concerns, they may be encrypted and decrypted. This will incur additional

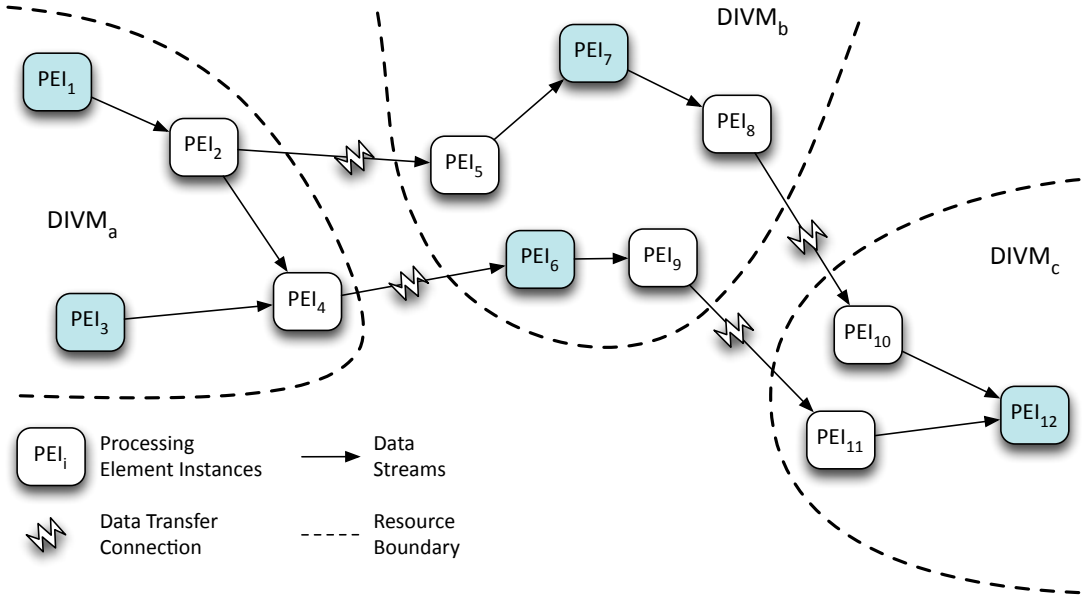


Figure 3.13: Partitioning PEIs across resource boundaries.

computational cost and communication delays[6]. Lets look at $ds_{7,8}$, $ds_{8,10}$ and $ds_{10,12}$. Assume that the volume per data unit of $ds_{7,8}$ is $vol_{7,8}$, and $vol_{7,8} > vol_{8,10} > vol_{10,12}$. From the computational cost perspective, the allocation of $\mathsf{PE_8}$ and $\mathsf{PE_{10}}$ to $\mathsf{DIVM_b}$ and $\mathsf{DIVM_c}$ is less significant. However, to fulfil the optimisation goal, both PEIs should be assigned to $\mathsf{DIVM_b}$, where the communication cost is minimum because $vol_{10,12}$ is the smallest along the path from PE_7 to $\mathsf{PE_{12}}$.

The optimiser also needs to understand the list of constraints that have to be satisfied, such as access restrictions. In principle, the PE implementation codes are stored in the repository, and are loaded to the execution engines during the deployment stage. However, some PEs may be mapped only to proprietary implementation accessible at a specific gateway. Similarly, some confidential data sources are only available via specific gateways. Another factor that restricts the choice of mapping candidates is processing capability. Some PE implementations are platform dependent and require a dedicated execution engine for their deployment, such as GPGPU [141], and some may be restricted by licensing issues. This type of PEI should be marked as an *anchor* and handled separately during the assignment process.

[6]Compression techniques may be applied to save transfer time, but it is beyond the scope of our discussion.

3.4.3 Conceptual model

Before we discuss the mapping algorithm, we first look at the conceptual model behind the algorithm.

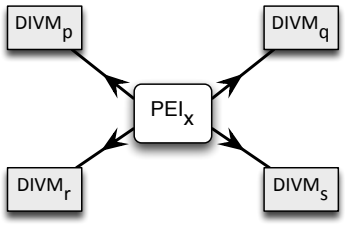


Figure 3.14: Conceptual model for optimisation.

Given a PEI and a set of potential DIVMs, as shown in Figure 3.14, we examine various factors that affect the allocation decision. These factors can be seen as *forces* that are pulling the PEI towards or away from a particular DIVM. We have defined three type of forces in this context: F_{anchor}, $F_{transfer}$ and $F_{conflict}$. $F_{anchor}(PEI_i, DIVM_j)$ represents the case when PEI_i is an anchored PEI and can only be executed on $DIVM_j$. When two PEIs (PEI_i and PEI_j) are connected, $F_{transfer}(PEI_i, PEI_j)$ is added to the force calculation and it is proportional to the data volume between them. PEIs connected via a large data stream are allocated to the same DIVM to minimise the communication cost. At the same time, there is another repulsive force, $F_{conflict}(PEI_i, PEI_j)$, that prevents heavy PEIs from being assigned to the same DIVM.

Thus, we define the force of assigning PEI_x into $DIVM_q$ as:

$$\begin{aligned} F_{assign}(PEI_x, DIVM_q) = {} & F_{anchor}(PEI_x, DIVM_q) \\ & - \sum F_{conflict}(PEI_x, PEI_y) \\ & + \sum F_{transfer}(PEI_x, PEI_y) - \sum F_{transfer}(PEI_x, PEI_z) \end{aligned}$$

where PEI_x is connected to PEI_y which is running in $DIVM_q$ and PEI_x is also connected to PEI_z which is running in $DIVM_p$ ($p \neq q$).

3.5 Three-stage mapping algorithm

The mapping algorithm aims to find a mapping that minimise workflow makespan with the consideration of all of the forces in three stages. In the first stage, the algorithm looks at F_{anchor} to assign those PEIs that have been marked as anchors. When PEI_x is anchored to $DIVM_q$, the $F_{anchor}(PEI_x, DIVM_q)$ can be seen as an infinite force that causes the assignment of PEI_x to $DIVM_q$, neglecting the other forces. All of the anchored PEIs will be assigned in this stage. In the second stage, the algorithm focus

at the assignment of heavy PEIs, based on $F_{conflict}$ and $F_{transfer}$. We use the unit cost as an approximation in calculating $F_{conflict}$. The algorithm assigns heavy PEIs evenly among the DIVMs. When come to the final stage, only light PEIs are unassigned. Seeing that light PEIs have neither anchor point (otherwise would have been assigned in the first stage) nor significant computational costs, the algorithm only considers the $F_{transfer}$ and find the mapping that yields minimal communication costs.

3.5.1 Prerequisites and assumptions

Below is a list of prerequisites and assumptions we made for the mapping algorithm:

1. The input of the mapping algorithm is a DAG, where the vertices denote PEIs and the edges represent the data streams connecting them. The vertices are annotated with unit cost and location (for anchored PEIs). The edges are annotated with their data rate.

2. The semantic descriptions of all of the PEIs are retrieved from the registry (i.e. data type and data rate), and the performance related data gathered from previous enactments are retrieved from PDB (i.e. unit cost). For newly implemented PEIs, the unit cost and data rate are marked as *zero*. The PDB will accumulate more performance data from each of the enactments and will incrementally build up the knowledge about these PEIs.

3. The list of available DIVMs is retrieved from the resource catalogue stored in the gateway. For the moment, we only consider an enactment environment where all of the DIVMs are connected to a single gateway.

4. For the moment, we assume that DIVMs are homogeneous, i.e. same computational capability and communication capability, and dedicated machines, i.e. have no other workloads, to simplify the assignment of heavy and light PEIs. We will discuss in the respective sections about this assumption.

5. The performance threshold for categorising heavy and light PEIs is already set. There are two possible ways to define the threshold: learn from previous enactments or determine by experts.

The output of the mapping algorithm is a graph which has been annotated with assignment decisions, i.e. to be enacted on which DIVM. The annotated graph is passed to workflow enactment engine to create executable workflows and prepare them for enactment.

3.5.2 Assigning anchored PEIs

Anchored PEIs can be semantically defined by users or observed from the PDB. There are two possible ways for users to define anchored PEIs: upon registration of a data source into the registry or by using the `location` modifier to specify data to which they are anchored (⇝*4.2*). The anchored PEIs can be assigned to one of the DIVMs near their source data[7]. The optimiser first attempts to assign these anchored PEIs. For each PEI, in the anchored set, the optimiser discovers all of the instances of the required data, and for each of these, discovers all of the DIVMs capable of accessing that data and close to that data. It then allocates each anchored PEI to one of these DIVMs, distributing load and reducing data movement.

The identification of anchored PEIs discussed above requires human intervention. The users who have deployed the data sources know the URI and register it with the registry, which is then accessed by the optimiser. The workflow authors need to know the infrastructure layout before they can explicitly define the anchor point in the workflow request. For some PEIs that perform a filtering process on an input data stream, it is better to place it closer to the data resource. This can avoid streaming a large volume of data between DIVMs, which would incur high communication costs.

Another possible way to identify anchored PEIs is through the PDB. Each time a PEI is enacted, the performance data for PEI to execute on the assigned DIVM are recorded. Some PEIs are categorised as anchored for processing capability, instead of resource accessibility restriction. In a heterogenous enactment environment, their performance varies when running on DIVMs with different computational power. The PDB can tell which DIVM is the best location for the PEI by analysing the performance data. Thus, it can classify it as an anchored PEI that is bound to that particular set of DIVMs. This is beyond the scope of of our mapping algorithm[8], but it will be discussed in Section 4.4.

The algorithm for anchored PEIs assignment is shown as Algorithm 1. The result of the assigning algorithm is a graph of PEIs that is partially annotated with DIVM on some of the PEIs, i.e. anchored PEIs. This graph is ready for the second stage of mapping—assigning heavy PEIs.

[7] There may be more than replied of the data with which they require proximity. In which case, the `location` modifier anchors to anyone of the corresponding subset of DIVMs.

[8] We assume that the DIVMs are homogenous.

Algorithm 1 Assigning anchored PEIs

Read the graph of PEIs, $G = \{p_1, p_2, ..., p_m\}$, the number of PEIs, $m = |G|$
Read the list of DIVMs, $D = \{d_1, d_2, ..., d_n\}$, the number of DIVMs, $n = |D|$
Initialise **int** *location*[1..m] /* *location*[i] is in which DIVM p_i has been allocated */

for all p_i in G **do**
  if p_i is annotated **then**
    Find anchor d_j from D
    location[i] = d_j
  end if
end for
Update G with new allocation

3.5.3 Assigning heavy PEIs

We divide the assignment algorithm for heavy PEIs into two parts (see Algorithm 2). In the first part, the algorithm traverses the graph of PEIs to find the heavy PEIs, i.e. those with unit cost greater than the performance threshold, and constructs a new list. The list is then sorted in descending order, i.e. largest cost first, and goes into the second part of the algorithm, where the assignment takes place. The second part of the algorithm assigns the list of heavy PEIs based on a round-robin with sorted unit cost. To avoid putting largest PEIs with the largest, we reverse the direction of the scan in the end of each round.

There are many ways of assigning the heavy PEIs and Algorithm 2 is one of them. However, the main drawback is that communication costs are not taken into consideration. PEIs connected with large volume data stream may be split across separate DIVMs. The second problem is the number of DIVMs used for the enactment. Given a large number of DIVMs for the enactment of a small workflow, it is likely that all of the PEIs will be assigned separately, one on each DIVM. This will cause a high deployment and communication overhead.

We are trying to minimise workflow makespan, in the first instance. This may be limited by contention for factors such as: CPU, RAM, disk I/O and network bandwidth. The optimised placement of the set of heavy PEIs (H) has therefore to avoid whichever of these is the limiting factors. Ergo, if we can characterise each p_i in terms of its use of each of these limiting factors and we can characterise each d_i for its capacity to deliver CPU, RAM and disk I/O, and each connection ($ds_{i,j}$) for its bandwidth, then we can assign each p_i to d_i until it would overload that p_i, or some $ds_{i,k}$ that is used by

Algorithm 2 Assigning heavy PEIs

Read the graph of PEIs, $G = \{p_1, p_2, ..., p_m\}$, the number of PEIs, $m = |G|$
Read the list of DIVMs, $D = \{d_1, d_2, ..., d_n\}$, the number of DIVMs, $n = |D|$
Read the threshold value, *threshold*
Initialise **int** *location*[1..m] /* *location*[i] is in which DIVM p_i has been allocated */
Initialise an empty list of heavy PEIs, H

/* construct a sorted list of heavy PEIs in descending order */
for all p_i in G **do**
  if p_i is not annotated **then** /* p_i already assigned in Algorithm 1 */
    if $t_{cost_i} > threshold$ **then**
      Insert p_i into H
    end if
  end if
end for
Sort H in descending order of t_{cost_i}

/* assign heavy PEIs */
Initialise $index = 1$
Initialise $direction = 1$
while H is not empty **do**
  Remove p_j from H
  $location[j] = d_{index}$
  $index = index + direction$
  /* reverse the direction when the scan read the end of a round */
  if $index > n$ **then**
    $index = n$
    $direction = -1$
  else if $index < 1$ **then**
    $index = n$
    $direction = 1$
  end if
end while
Update G with new allocation

the candidate assignee p_i. We can use any heuristic or a complete analysis, to find a good candidate assignment of H to D. As there are research issues in deciding possible bottleneck should be considered, in determining the unit cost and node/connection capacity in each of these dimensions, and in choosing a method of assignment with constraints. This is a substantial research undertaking in its own right, made more complex by some of the parameters and variables not being independent. Further, we also need to explore the choice of threshold bounding H and the amount we then need to reserve for light PEIs being assigned in co-location with H. This is therefore postponed for later consideration (⇝*6.2*). We now proceed to the third stage of the mapping algorithm.

3.5.4 Assigning light PEIs

At this stage, the anchored and heavy PEIs have already been allocated to respective DIVMs.

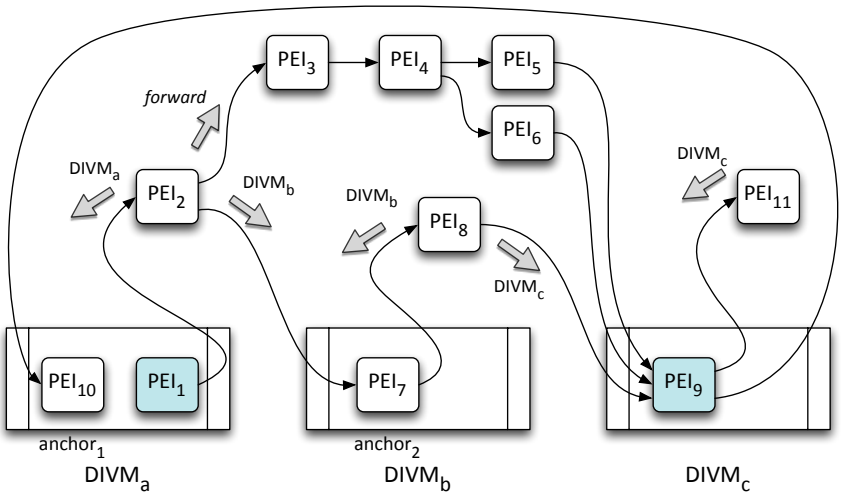


Figure 3.15: Partially assigned graph with the remaining light PEIs.

Figure 3.15 shows a partially assigned graph of PEIs. $\mathsf{PEI_7}$ and $\mathsf{PEI_{10}}$ are anchored PEIs and have been assigned, and so have the heavy PEIs (coloured in blue). There are many ways to assign the remaining light PEIs because the PE may have arbitrary numbers of inputs and outputs. $\mathsf{PEI_{11}}$ is only connected to $\mathsf{PEI_9}$. Thus, it should obviously be assigned to $\mathsf{DIVM_c}$. The decision for $\mathsf{PEI_8}$ is quite straight forward because it has single predecessor and single successor. The assignment is dependent on the relative volume of the input data stream and the output data stream, i.e. assigned to $\mathsf{DIVM_b}$ if the volume of the input data stream is the larger and to $\mathsf{DIVM_c}$, if it is the smaller.

The assignment for $\mathsf{PEI_2}$ is difficult. We have to look at all of its data streams in making the decision. It will be allocated to $\mathsf{DIVM_a}$, if $\mathsf{vol_{1,2}}$ >the other two data streams, or to $\mathsf{DIVM_b}$, if $\mathsf{vol_{2,7}}$ is the largest. If $\mathsf{ds_{b,c}}$ is the largest among the three, then decision falls on $\mathsf{PEI_3}$ and maybe the rest of the subgraph connecting with $\mathsf{PEI_9}$.

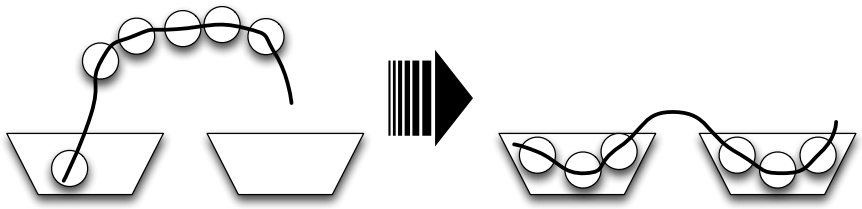


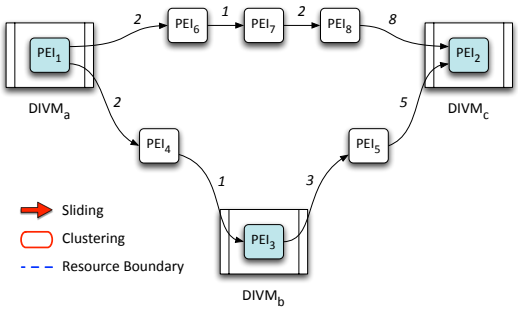
Figure 3.16: Beads and bowls as an analogy of PEIs and Workflows.

Conceptually, we can see the graph as *beads* connected by strings that correspond to a sequence of their data-flow interconnections, and the DIVMs as the *bowls* to store the beads, as illustrated in Figure 3.16. The assignment problem is the task of allocating beads into bowls. For each bead, we have to decide whether it should slide down into the left bowl or to the other direction into the right bowl. For *sliding* PEIs into DIVMs, the decision is affected by two criteria: *the volume of data flow* along each stream connecting the PEI to its predecessors and successors, and *the workload* of each of the DIVMs. If there are multiple connections between these beads (i.e. PEI with more than one input/output), there may be several strings pulling a PEI in different directions. In such a situation, the string which has the biggest pull will be chosen because it's transferring most data. We also need to take into account the summation of the strings, $\sum vol_{i,j}$ for all of the connections from $\mathsf{PEI_i}$ towards $\mathsf{DIVM_k}$.

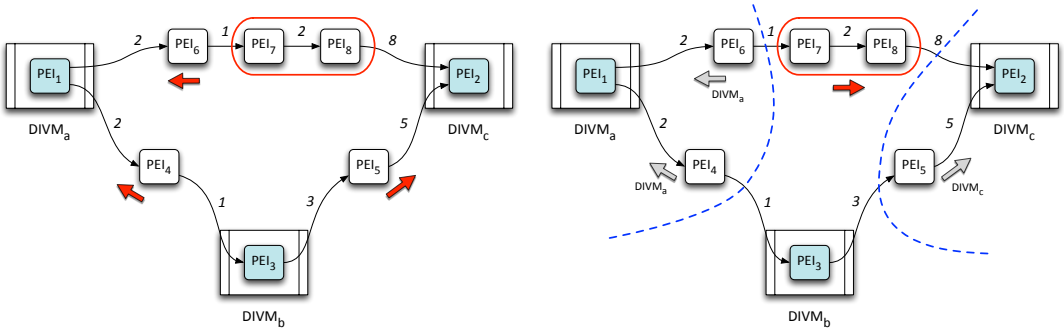
The main objective of the beads-sliding algorithm is to find the *weakest link* on each of the strings to perform the cut. This can be done with a brute-force method to find the assignment of a set of light PEIs on a set of DIVMs that yields the minimum communication cost. However, the brute-force method incurs high optimisation overhead. To reduce the delay before and variance of delay before workflows start, we have chosen a less exhaustive method to solve the cutting problem with a heuristic algorithm, shown as Algorithm 3. The algorithm performs two tasks: clustering and sliding. It clusters a light PEI to its neighbour with the strongest pulling force, i.e. the volume of data flow $vol_{i,j}$, into beads. If the strongest link of the beads is connected to a PEI that has been allocated, the beads are slid into that corresponding DIVM. We will illustrate this algorithm with three examples.

Algorithm 3 Assigning light PEIs

1: Read the graph of PEIs, $G = \{p_1, p_2, ..., p_m\}$, the number of PEIs, $m = |G|$
2: Read the list of DIVMs, $D = \{d_1, d_2, ..., d_n\}$, the number of DIVMs, $n = |D|$
3: Initialise **bool** *placed*[1..m] /* *placed*[i] = *true* if the p_i is already assigned */
4: Initialise **int** *location*[1..m] /* *location*[i] in which DIVM p_i has been allocated */
5: Initialise empty list of bead, B /* each bead, c_i consists of ≥ 1 PEIs */
6:
7: /* Set up PEIs for prior to allocation */
8: **for all** p_i in G **do**
9: **if** p_i is annotated **then** /* p_i already assigned in Algorithm 1 and 2 */
10: *placed*[i] = *true*
11: *location*[i] = *annotation*
12: **else**
13: *placed*[i] = *false*
14: Create new bead, b_j
15: Insert b_j into B
16: **end if**
17: **end for**
18:
19: /* Slide light PEIs */
20: **while** B is not empty **do**
21: Remove bead, b_i from B
22: Find adjacent PEI connected to b_i, p_j with $max(vol_{i,j})$
23: **if** *placed*[j] **then** /* p_j has been allocated */
24: /* Assign all PEIs in the bead to the same DIVM with p_j */
25: **for all** p_k in bead, b_i **do**
26: *placed*[k] = *true*
27: *location*[k] = *location*[j]
28: **end for**
29: **else**
30: Remove bead, b_l, from B that comprises p_j
31: Merge b_i and b_l to form new bead, b_m
32: Insert b_m to B
33: **end if**
34: **end while**
35: Update G with new allocation



(a) Partially assigned graph



(b) First sliding

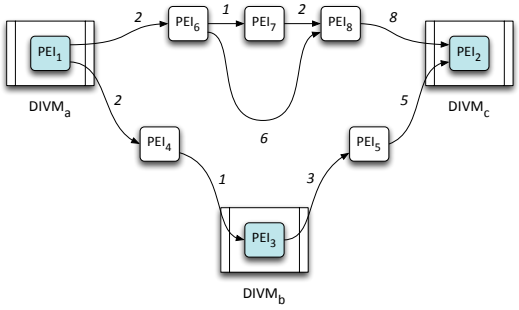
(c) Second sliding

Figure 3.17: Example light PEIs sliding.

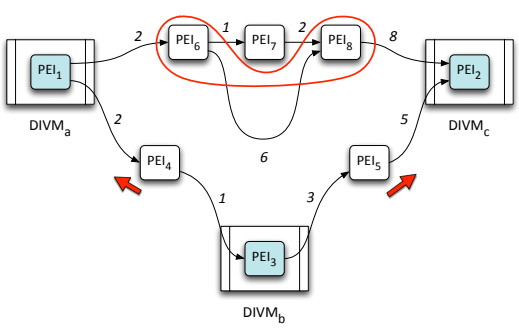
Figure 3.17(a) shows a simple graph of light PEIs that consists of three "strings", and the weight on the edges denote the volume of the data streams. The algorithm first constructs a list of beads[9], $B = \{\{PEI_4\}, \{PEI_5\}, \{PEI_6\}, \{PEI_7\}, \{PEI_8\}\}$, from all of the unallocated PEIs (i.e. lines 8 to 17 of Algorithm 3). Each time whenever a bead has been removed (line 21), the algorithm decides whether a sliding (i.e. the if block—lines 25 to 28) or clustering (i.e. the else block—lines 30 to 32) should be performed. PEI_4 is connected to PEI_1 and PEI_3. The pulling force towards PEI_1 is the strongest, i.e. $vol_{1,4} > vol_{4,3}$, and apparently it has been allocated to DIVM_a. This causing the algorithm goes into the if block (line 23), and places PEI_4 into DIVM_a. The algorithm then removes another bead from B, and uses the same method to slide PEI_5 and PEI_6, as the strongest links of both PEIs are connected to a respective PEI, that has been allocated (see Figure 3.17(b)) Next, the algorithm picks up $\{PEI_7\}$ from B, and discovers that the strongest link is connected with PEI_8, which is not yet been allocated. So, the algorithm removes $\{PEI_8\}$ from B (line 30), combines it with $\{PEI_7\}$ to form a new bead $\{PEI_7, PEI_8\}$ (line 31), and inserts the bead back into B (line 32). At this stage, three light PEIs have been allocated and one bead remains in the list of unassigned PEIs (see Figure 3.17(c)). The last bead, i.e. $\{PEI_7, PEI_8\}$ is strongly

[9]We use "bead" to refer to the cluster of PEIs in B.

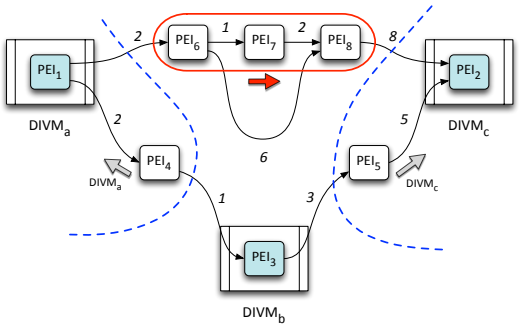
pulled towards PEI_2, which has been allocated to DIVM_c. Thus, $\{PEI_7, PEI_8\}$ is slid into DIVM_c. The result of the mapping is $\{PEI_1, PEI_4, PEI_6\}$ to DIVM_a; $\{PEI_3\}$ to DIVM_b; and $\{PEI_2, PEI_5, PEI_7, PEI_8\}$ to DIVM_c. The total data volume across DIVMs is $1 + 1 + 3 = 5$.



(a) Partially assigned graph



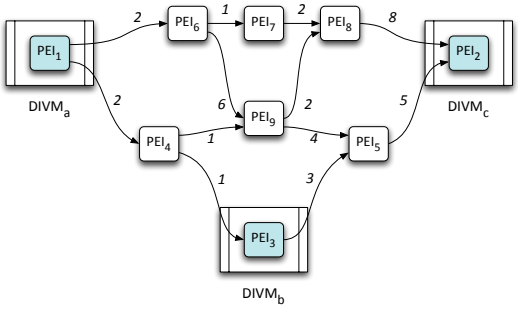
(b) First sliding



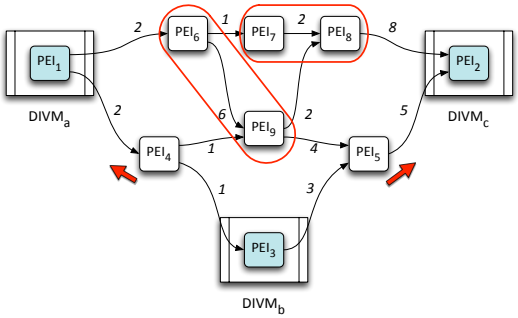
(c) Second sliding

Figure 3.18: PEIs sliding with parallel data streams (modified from Figure 3.17).

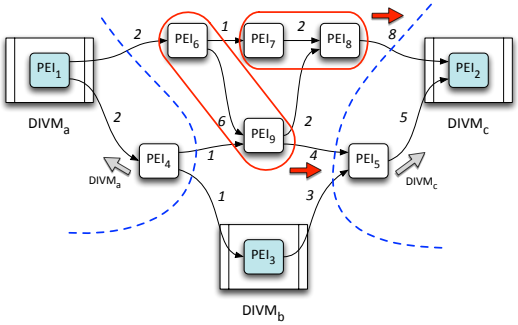
We slightly modify the graph by adding an additional data stream connecting PEI_6 to PEI_8 and leaving the rest unchanged, as shown in Figure 3.18(a). The sliding of PEI_4 and PEI_5 remains the same in previous example (see Figure 3.18(b)). When the algorithm has removed $\{PEI_6\}$ from B and found that the largest data volume is $vol_{6,8}$, it removes $\{PEI_8\}$ from B, forms a new bead with $\{PEI_6\}$ and inserts back to B, i.e. $B = \{\{PEI_7\}, \{PEI_6, PEI_8\}\}$. The algorithm continues to remove $\{PEI_7\}$ from B. $\{PEI_7\}$ has the maximum data volume connected to PEI_8. So, the algorithm remove the bead that comprise PEI_8, i.e. $\{PEI_6, PEI_8\}$, to form a new bead $\{PEI_7, PEI_6, PEI_8\}$ (see Figure 3.18(c)). Lastly, the algorithm remove this bead from B, calculates the forces, and slides it into DIVM_c. The result of the mapping is $\{PEI_1, PEI_4\}$ to DIVM_a; $\{PEI_3\}$ to DIVM_b; and $\{PEI_2, PEI_5, PEI_6, PEI_7, PEI_8\}$ to DIVM_c. The total data volume across DIVMs is $1 + 2 + 3 = 6$. An important remark is that by adding $ds_{6,8}$ to the graph, the pulling force of PEI_6 towards DIVM_c has been increased and causing PEI_6 to be assigned to DIVM_c.



(a) Partially assigned graph



(b) First sliding

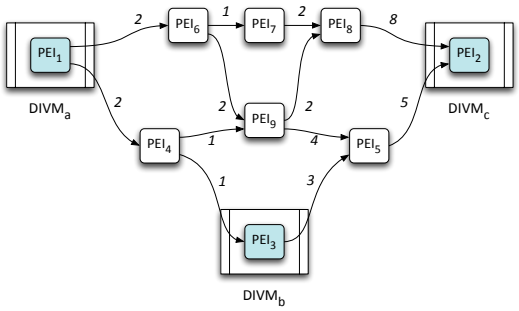


(c) Second sliding

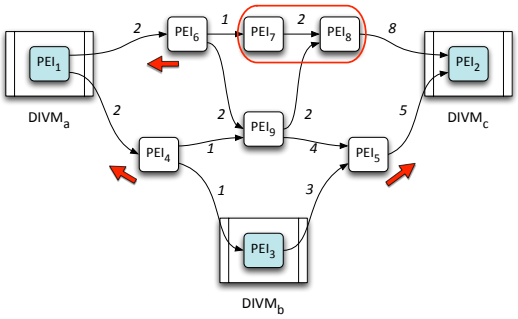
Figure 3.19: PEIs sliding with multiple input and output (modified from Figure 3.18).

We have demonstrated the bead-sliding algorithm with two rather simple examples, where each of the PEIs is pulled from two directions. We further modify the graph by adding $\mathsf{PEI_9}$ into the graph, which is connected to $\mathsf{PEI_4}$, $\mathsf{PEI_5}$, $\mathsf{PEI_6}$ and $\mathsf{PEI_8}$, as shown in Figure 3.19(a). Even though new forces have been added to $\mathsf{PEI_4}$ and $\mathsf{PEI_5}$, but these forces are smaller than their existing pulling forces. Figure 3.19(b) shows the allocations of $\mathsf{PEI_4}$ and $\mathsf{PEI_5}$, and the forming of two new beads, $\{PEI_6, PEI_9\}$ and $\{PEI_7, PEI_8\}$. The adjacent PEIs of $\{PEI_6, PEI_9\}$ that has the largest data volume is $\mathsf{PEI_5}$, which has been assigned to $\mathsf{DIVM_c}$. So $\{PEI_6, PEI_9\}$ is slid into $\mathsf{DIVM_c}$. Figure 3.19(c) shows the result of the sliding, with $1 + 1 + 2 + 3 = 7$, total data volume across DIVMs.

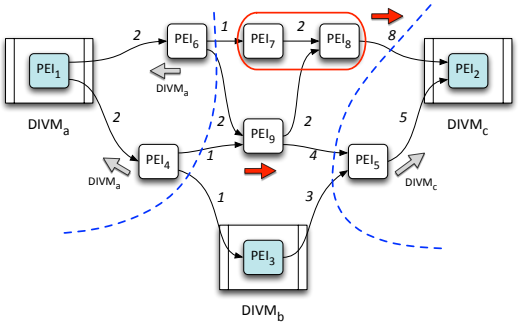
All of the three assignments that we have demonstrated in this section have shown a successful mapping where the cut yields the lowest communication cost. We now look at a scenario where the heuristic gets it wrong, i.e. the cut does not reach the lowest data volume across DIVMs. We modify Figure 3.19(a) by changing $vol_{6,9}$ to 2, to create two data streams – that are connecting $\mathsf{PEI_6}$ to its adjacent nodes – with equal data volume (see Figure 3.20(a)). The assignments of $\mathsf{PEI_4}$ and $\mathsf{PEI_5}$ remain unchanged. However, the decision for assigning $\mathsf{PEI_6}$ is now getting complicated, as there are two data streams with maximum data volume, i.e. $ds_{1,6}$ and $ds_{6,9}$. Now we have two options: clustering $\mathsf{PEI_6}$ with $\mathsf{PEI_9}$ or sliding $\mathsf{PEI_6}$ to $\mathsf{DIVM_a}$. Assume that the latter option has



(a) Partially assigned graph



(b) First sliding



(c) Second sliding

Figure 3.20: PEIs sliding with non-optimal cut (modified from Figure 3.19).

been chosen, as shown in Figure 3.20(b). The algorithm then processes $\{PEI_7\}$ and clusters it with $\{PEI_8\}$, i.e. $B = \{\{PEI_7, PEI_8\}, \{PEI_9\}\}$. Lastly, the algorithm slides both beads into $\mathsf{DIVM_c}$ (see Figure 3.20(c)). The total data volume across DIVMs is $1+1+2+1+3 = 8$. However, the optimal cut would be clustering $\mathsf{PEI_6}$ with $\mathsf{PEI_9}$, and sliding this bead into $\mathsf{DIVM_c}$ (same as the assignment shown in Figure 3.19, where the total data volume across DIVMs is 7). The algorithm does not consider the aggregated forces that are pulling $\mathsf{PEI_9}$ towards $\mathsf{DIVM_c}$. This is a limitation of the beads-sliding algorithm that can be improved in the future.

When the beads-sliding has run, all of the PEIs will have been assigned to their respective DIVMs, and this information is annotated onto the workflow graph. The workflow enactor will take over from here, to create concrete workflows, and prepare for the execution.

3.6 Summary of optimisation model

This chapter has introduced a mapping algorithm for optimising the enactment of streaming workflows. It has demonstrated the use of performance data gathered from previous enactments in optimisation. We have clearly defined:

- the streaming workflow model in our context, including the definition of PEs and data streams,
- an abstraction of the execution environment as a DIVM,
- two classes of optimisation problems: graph transformation and DIVM allocation,
- a definition of unit cost for streaming workflow,
- a classification of PEIs based on their behaviour, and
- a three-stage mapping algorithm.

In this thesis, we focus on minimising the workflow makespan as our optimisation goal. This is a time-based optimisation that looks at the application perspective. There are other optimisation goals that focus on the resource perspective. For instance, optimising the system throughput to get the most out of the data-intensive platform, reducing disk storage to accommodate more applications, or minimising energy consumption. The exploration of these optimisation goals is beyond our research scope.

We have briefly discussed two classes of optimisation problems: graph transformation and DIVM allocation. Our work focuses on the latter. We have proposed a conceptual model that looks into various forces that affect the assignment decision and suggested that the assignment can be done in stages. We use time as the approximation of all of the factors that influence the enactment.

There are a large body of optimisation work on job scheduling based on heuristic or mathematical approaches, such as queuing theory [25] and graph partitioning [38, 97]. Our DIVM allocation problem is fundamentally different from the model used in queuing theory that is applied to an unbound job queue. We are scheduling a finite set of PEIs prior to their execution and use the concepts of job-shop scheduling to explain the notion of time used in streaming models. These do not necessarily reach equilibrium conditions needed for the application of queuing theory. In addition queuing theory usually applies to independent unpredictable arrivals whereas the arrival of values in streams are computationally correlated and therefore approximately predictable.

The way we partition our PEIs is also different that the graph partitioning approach that attempts to find cut on a graph, which results a balance distribution workload. We are categorising the PEIs into two distinct partitions, i.e. classes, and apply different techniques to assign them to the DIVMs. We hypothesise that our allocation of the heavy workload PEIs followed by a migration of the lighter PEIs to cluster around them, is, in some sense equivalent to graph partitioning. This requires further analysis and experiment to discover the differences in workload distribution as optimisation cost. The exploration of other optimisation techniques is a potential area for future research (⇝*6.2*).

In the next chapter, we will discuss the underlying architecture that supports the optimisation model.

CHAPTER 4

Data-intensive architecture

This chapter describes the underlying architecture that enables the implementation of the optimisation model proposed in Chapter 3. We will attempt to answer the following research questions:

1. How do we achieve separation of concerns in dealing with scientific workflows? (⇝*4.1*)
2. How do we describe scientific workflows? (⇝*4.2*)
3. How do we enact scientific workflows? (⇝*4.2.2*)
4. How do we capture performance data during workflow enactment? (⇝*4.3*)
5. How do we organise performance data? (⇝*4.4*)
6. How do we access performance data? (⇝*4.4.2*)

4.1 Data-intensive architecture

The data-intensive architecture promotes the exploration and exploitation of distributed and heterogeneous data and spans the complete knowledge discovery process, from data access through data preparation, data analysis and results presentation. Typically, these stages are revisited iteratively as researchers develop understanding, methods and evidences. The architecture has three levels, as shown in Figure 4.1. The upper layer (the *tool* level) supports the work of both domain experts and data analysis experts. It houses an evolving set of portals, tools and development environments, sufficient to support the diversity of both of these communities of experts. The lower layer (the *enactment* level) houses a large and dynamic community of providers who deliver data and data-intensive enactment environments as an evolving infrastructure (called the *data-intensive platform*), which supports all of the work done in the upper

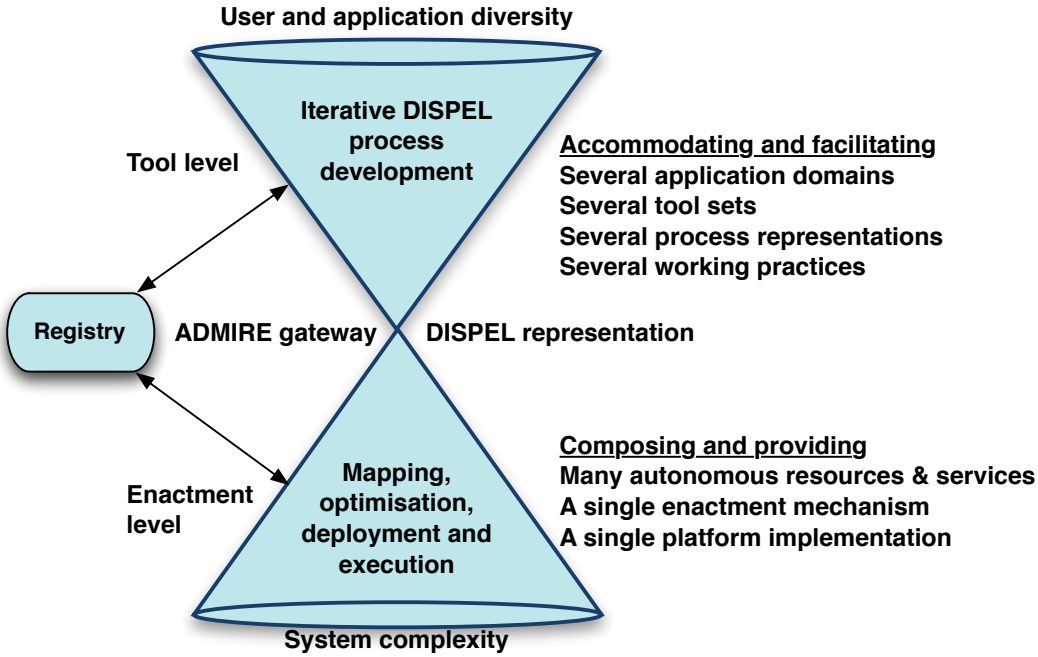


Figure 4.1: The data-intensive architecture from Atkinson *et al.* [10].

layer. Most of the work done by data-intensive engineers goes on here. Data-analysis experts can also develop generic libraries optimised for a provider's enactment environment at this level should they so desire. The crucial innovation is the neck of the hourglass, which is a tightly defined and stable interface through which the two diverse and dynamic upper and lower layers communicate. This has a minimal and simple protocol and language, ultimately controlled by standards, into which the upper and lower communities can invest, secure in the knowledge that changes to this interface will be carefully controlled.

We have explored our interface by creating a new workflow composition language, named DISPEL. The primary function of DISPEL is to express how a data-intensive application uses processing elements (e.g. that provide noise filtering algorithms and perform pair-wise cross correlation of time-series data), and how these elements communicate with each other. In other words, DISPEL is a language for expressing a directed graph, where processing elements represent the computational nodes and the flow of data between them is represented by connections. Thus, DISPEL provides an abstraction technique for a data-streaming execution model. At the lower level, DISPEL also handles validation, and provides the required model for carrying out workflow optimisations. It is designed to be comprehensible to expert humans so that it is also a medium for dialogue between experts.

4.1.1 Registry

The architecture also has its own registry which is used to store descriptions of all components available for the construction of data-intensive tasks; the registry serves to relate the entities used by the tool level to the various possible implementations of those entities at the disposal of the enactment level. Thus the semantic descriptions stored in the registry provide consistent functionality across the tool and enactment levels.

As an example of the type of information recorded in the registry, both for human and system consumption, processing elements are described with:

1. A unique name (a URI).
2. A short natural language description.
3. An ontology-based classification of their purpose.
4. A precise description of their input and output connections, including their structural and domain types, as described in the previous section.
5. The consistency and propagation rules for structural and domain types in their input and output connections.
6. Their known relationships in the sub-type hierarchy.
7. Their patterns of data consumption and production.
8. Their termination behaviour and error modes.
9. Information useful for placing instances and optimising enactment.
10. Information about version relationships that may be used by automated change adapters.

The registry is a key component of the architecture for three reasons. First, it holds and validates all of the descriptions discussed above, and expands as descriptions evolve. Second, it acts as a consistency foundation and database for all of the subsystems (tools, language processing and enactment) in the architecture. Third, it provides a foundation for sharing and cooperation using web-based tools, ontologies and information models.

4.1.2 Data-intensive platform

The lower layer of the data-intensive architecture (see Figure 4.1), the enactment level, is intended to host a large and dynamic community of providers who deliver data and data-intensive enactment environments as an evolving infrastructure, called the *"data-intensive platform"*, that supports the work of the upper layer. The DISPEL request is produced using facilities at the tool level and then sent to a gateway, which acts as the

entry point to the data-intensive platform. A data-intensive platform comprises:

- an *application development environment* (including libraries of processing elements, functions, and data types),
- a *gateway* as the entry point of enactment which accepts DISPEL request,
- a DISPEL *language processor* that compiles the DISPEL request into graph representation,
- an *enactment engine* that optimises those graphs, deploys them, executes them in a controllable framework that permits interaction with the end user, and finally terminates them and cleans up the environment,
- *execution engines* that deploy and execute workflows, and
- *data sources* that are connected and made available through this platform.

Once a DISPEL request for enactment has been received, it has to be transformed and mapped to selected parts of the data-intensive platform. This involves analysing the request and determining whether it can be run, whether it is best run on the local platform, or better delegated to another, or whether it should be partitioned and each part delegated to platforms that better matches its balance of resource requirements. Additionally, the data-intensive platform takes full responsibility for buffering and optimising the flow of values along each connection, e.g. passing by reference when possible; or serialising, compressing and encrypting long haul transmission. The system will automatically buffer, spilling to disk when this is unavoidable.

4.2 DISPEL

The Data-Intensive Systems Process Engineering Language (DISPEL) is a data-flow workflow construction and optimisation language for distributed data-intensive applications [114]. DISPEL holds the key to achieving the separation of concerns within the data-intensive architecture. It is used to describe abstract workflows for data-intensive applications. Compared to other workflow languages, e.g. Meandre's ZigZag [123], Taverna's Simple Conceptual Unified Flow Language (SCUFL) [138], Kepler's Modelling Markup Language (MoML) [115] and Swift's SwiftScript [178], DISPEL is more human readable; DISPEL is non-XML and excludes execution details. Thus, DISPEL is the right tool to facilitate the dialogues between the three group of experts in data-intensive research. DISPEL draws inspiration from other descriptive languages, i.e. database query languages and workflow languages, but it is an imperative language that constructs workflows and interacts with the architecture (e.g. by registering a new PE in the registry and submitting a workflow for enactment). DISPEL is not the language used to de-

scribe the detailed computation of PEs, but how they are connected to form a workflow. At the lower level, DISPEL also handles validation, and provides the required model for carrying out workflow optimisations. Thus, DISPEL is powerful and descriptive enough to support the communication between the components in the data-intensive platform (see the chapter on *"Data intensive thinking with DISPEL"* [9] in [10]).

A DISPEL workflow is a composition of PEs connected with data streams. PEs may be primitive (implemented in other languages, e.g. Java and Python) or composite (implemented in DISPEL). Before a PE can be used as a workflow building block, an instance of the PE, i.e. PEI, is instantiated. Data are streamed between PEIs via connections. Connections carry data from one output interface of a PEI to one or more input interfaces of other PEIs. In constructing workflows, users can reuse existing PEs (from general or domain specific libraries), or define their own. The newly defined PEs can be registered in the registry for later use and shared among other users. Thus, a DISPEL request can be a complete description of a workflow, a declaration of PEs, or a declaration of functions.

4.2.1 A simple DISPEL example

DISPEL uses a notation that is similar to Java. Figure 4.2 is a simple example of a DISPEL request that retrieves data from a database and delivers the results to the requesting client.

Line 1 is a packaging methodology similar to Java that avoids the newly registered PEs from conflicting with existing PEs that are unrelated but have similar names. Lines 2 and 3 import predefined PEs that have been registered in the registry. Lines 7 and 8 create corresponding instances for all of the PEs used in the workflow. The reason to have the explicit definition of PEI is to allow multiple instances of a PE to be instantiated in the workflow. For instance, a workflow can comprises two `SQLQuery` PE instances that are accessing two data sources separately. Line 11 defines the SQL expression to access the database—in this case, it requests all of the data about workflows submitted by Chee Sun. Lines 14 to 17 set up the data flow between the PEIs. Line 14 defines the URI for the data source by supplying is as a stream literal to one of the inputs of `query`. Line 16 connects the output stream of `query` to the input stream of `results`. Lastly, the workflow is submitted for enactment on line 21.

A full description of DISPEL and more examples of DISPEL requests can be found in DISPEL reference manual [114] and the language definition chapter [131] in [10].

```
package book.examples {
  // Import existing PEs
  use dispel.db.SQLQuery;
  use dispel.lang.Results;

  // Create instances of PEs for workflow
  SQLQuery query = new SQLQuery;
  Results results = new Results;

  // Specify query to feed into workflow
  String exp = "SELECT * FROM Workflow WHERE user_id = 'cheesun'";

  // Connect PE instances to construct the workflow
  |- "uk.ac.ed.inf.pdb" -| => query.source;    // Specify URI of data source
  |- exp -| => query.expression;               // Specify the query
  query.data => results.input;                 // Set up data flow from query to results
  |- "Workflow submitted by cheesun" -| => results.name;
                                               // Specify the name of the results for user

  // Submit workflow for enactment
  submit results;
}
```

Figure 4.2: A simple DISPEL request to retrieve data from the PDB.

A DISPEL request is submitted to a gateway for enactment. A gateway has numerous ways to implement the abstract workflow, i.e. any DISPEL request, on its computational resources.

4.2.2 DISPEL enactment

There are four stages in the enactment process of data-intensive computations, as shown in Figure 4.3. Any implementation of a Data-Intensive Platform will implement these stages in some form.

Stage 1 DISPEL *Language Processing*, which includes parsing and validating DISPEL request and interpreting it to generate the corresponding data-flow graph. The registry provides the semantic descriptions of the PEs and functions used in the request, where their implementation are kept in the repository.

Stage 2 *Optimisation*, which includes selection of PEs, transformation of the data-flow graph, substitution of PEs, identification of available resources, and the mapping of PEs to resources. The mapping algorithm proposed in the previous chapter is run in this stage. The output of the optimisation is a data-flow graph annotated with resources information.

Stage 3 *Deployment*, which includes compiling the graphical representation into platform-specific executable graphs and setting up resources and data-flow connections. Concrete workflows are created and ready for deployment.

Stage 4 *Execution and Control*, which includes instrumentation and performance measurement, failure management, delivering results and clean up. The performance data are stored in the PDB, and used for runtime monitoring and future optimisation.

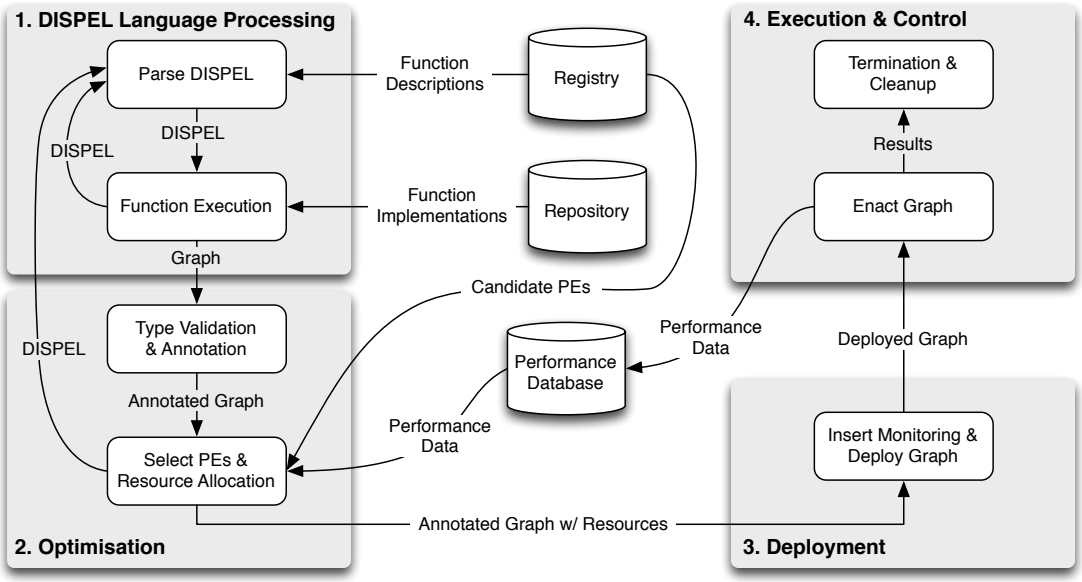


Figure 4.3: Steps involved in processing DISPEL programs.

4.2.3 DISPEL in optimisation context

DISPEL has two important features to support workflow optimisation. First, we look at *function abstraction* in DISPEL. A DISPEL function abstracts a composition of PEs and it will generate a DISPEL graph of PEIs (can be a single PE or an entire application) when processed by the language processor. The function abstraction allows the generation of re-usable workflow patterns and parameterisation of DISPEL requests. From the workflow reusability and repeatability point of view, users can execute the workflow with different sets of parameters or repeatably execute on several data sources in parallel. From the optimisation point of view, the use of functions allows dynamic expansion of the DISPEL graph based on the available computing and data resources.

We illustrate this with a common workflow pattern in data mining which is widely used across different domains. The k-fold cross validation is a workflow pattern used to train

and evaluate the accuracy of a classification algorithm. Data are randomly partitioned into k subsets, where $k-1$ subsets are used to train a new classifier, which is later tested with the remaining subset. This is to make sure that the data used for the training do not influence the accuracy of the evaluation. Wrapping the k-fold cross-validation pattern as a function enables training and testing of multiple classifiers concurrently.

```
1  package dispel.datamining {
2    // Import abstract types.
3    use dispel.datamining.Validator;
4    use dispel.datamining.TrainClassifier;
5    use dispel.datamining.ApplyClassifier;
6    use dispel.datamining.ModelEvaluator;
7    use dispel.core.DataPartitioner;
8    // Import PE constructor function.
9    use dispel.datamining.makeDataFold;
10   // Import implemented type.
11   use dispel.core.ListMerge;
12
13   // Produces a k-fold cross validation workflow pattern.
14   PE<Validator> makeCrossValidator(Integer k,
15                                    PE<TrainClassifier> Trainer,
16                                    PE<ApplyClassifier> Classifier,
17                                    PE<ModelEvaluator>  Evaluator) {
18     Connection    input;
19     // Data must be partitioned and re-combined for each fold.
20     PE<DataPartitioner> FoldData = makeDataFold(k);
21     FoldData            folder   = new FoldData;
22     ListMerge           union    = new ListMerge with inputs.length = k;
23
24     // For each fold, train a classifier then evaluate it.
25     input => folder.data;
26     for (Integer i = 0; i < k; i++) {
27       Trainer    train     = new Trainer;
28       Classifier classify  = new Classifier;
29       Evaluator  evaluator = new Evaluator;
30
31       folder.training[i] => train.data;
32         train.classifier => classify.classifier;
33           folder.test[i] => classify.data;
34         classify.result => evaluator.predicted;
35           folder.test[i] => evaluator.expected;
36         evaluator.score => union.inputs[i];
37     }
38
39     // Return cross validation pattern.
40     return PE( <Connection data = input> =>
41                <Connection results = union.output> );
42   }
43
44   // Register PE pattern generator.
45   register makeCrossValidator;
46 }
```

Figure 4.4: PE function `makeCrossValidator` from Martin and Yaikhom [131].

Figure 4.4 is a k-fold cross-validation function `makeCrossValidator`, found in the package `dispel.datamining.kdd` of the DISPEL libraries (see Appendix C of [10]). The `makeCrossValidator` function takes four parameters: `TrainClassifier` PE encapsulates a learning algorithm to build a classifier from a training dataset, `ApplyClassifier` PE takes the test dataset and a classifier, and performs a classification, `ModelEvaluator` PE takes observation data and classified result, and assigns a score that reflects the accuracy of the classification, and an `Integer`, k, that specifies the number of subsets into which the data should be partitioned. Line 20 shows the use of another function, `makeDataFold` to partition the input dataset. Lines 26 to 37 set up the data flow for training and testing k classifiers. For a 4-fold cross-validation, the input dataset will be partitioned into four subsets. Each subset will be used three times in the training of three distinct classifiers in parallel, and once for testing another classifier. The execution of this function will return four different classifiers with their evaluation scores.

The `makeCrossValidator` function is designed to support the k-fold cross-validation workflow pattern, which can be reused with different learning algorithm, different input data and different values of k. Users can define a PE that encapsulates their own learning algorithm for training the classifier for their own domain, as long as the PE is compatible with the `TrainClassifier`, i.e. takes a list of tuples as input and produces a classifier. The `ModelEvaluator` needs to be appropriate for the learning algorithm, `TrainClassifier`. We have developed three application-specific PEs for the EURExpress workflow (⇝*5.1.1*): `TrainClassifier`, `Classify`, and `Eval` in package `book.examples.eurexpress`. Figure 4.5 is the DISPEL request to run a 10-fold cross validation for training and testing classifiers for classifying humerus part of a mouse embryo image. From the optimisation perspective, the function can be expanded according to the available resources. The blue box in Figure 4.6 shows the PEIs for each "*fold*", which can be mapped onto different DIVMs for parallel execution.

The second important feature of DISPEL, from the optimisation perspective, is the three-level type system: language types, structural types and domain types. The *language type* validates the consistency of DISPEL sentences. For instance, the language processor checks all of the four parameters for calling the `makeCrossValidator` function (Line 18 of Figure 4.5), to ensure that they match the correct types defined in the function declaration.

```
1  package book.examples.eurexpress {
2    // Import PEs from dispel libraries
3    use dispel.db.SQLQuery;
4    use dispel.lang.Results;
5
6    // Import PEs from eurexpress library
7    use book.examples.eurexpress.DataProducer;
8    use book.examples.eurexpress.TrainClassifier;
9    use book.examples.eurexpress.Classify;
10   use book.examples.eurexpress.Evaluator;
11
12   // Import abstract type and constructor
13   use dispel.datamining.Validator;
14   use dispel.datamining.makeCrossValidator;
15
16   // Create a cross validator PE
17   PE<Validator> CrossValidator =
18       makeCrossValidator(10, TrainClassifier, Classify, Evaluator);
19
20   // Create instances of PEs for workflow
21   SQLQuery query = new SQLQuery;
22   DataProducer producer  = new DataProducer;
23   CrossValidator validator = new CrossValidator;
24   Results results   = new Results;
25
26   // Specify query to feed into workflow
27   String exp = "select euxassay_id, case when " +
28                "embryo_limb_forelimb_arm_upper_arm_mesenchyme_humerus " +
29                "< 5 then 0 else 1 end from annotation";
30
31   // Connect PE instances to construct the workflow
32   |- "uk.ac.ed.inf.eurexpress" -| => query.source;
33   |- exp -| => query.expression;
34   query.data => producer.source;
35   producer.data => validator.data;
36   validator.results => results.input;
37   |- "Classifier Scores" -| => results.name;
38
39   // Submit workflow for enactment
40   submit results;
41 }
```

Figure 4.5: A DISPEL request to run a 10-fold cross validation for an anatomical component classifier.

Figure 4.7 is the DISPEL request for registering `ReadFile` PE; we use it to illustrate the the two types. Line 7 that defines `ReadFile` as a type of PE, that has two input connections, namely `source` and `fPath`, and a single output connection, `data`.

The *structural type* defines the format and low-level interpretation of values in the stream connecting PEs. Lines 8 and 9 specify that both input connections of `ReadFile` are of structural type `String`. During the validation process, the language processor checks whether these connections are fed with a stream of `String` type data. In the situation where a mismatch is detected, the language processor should check whether

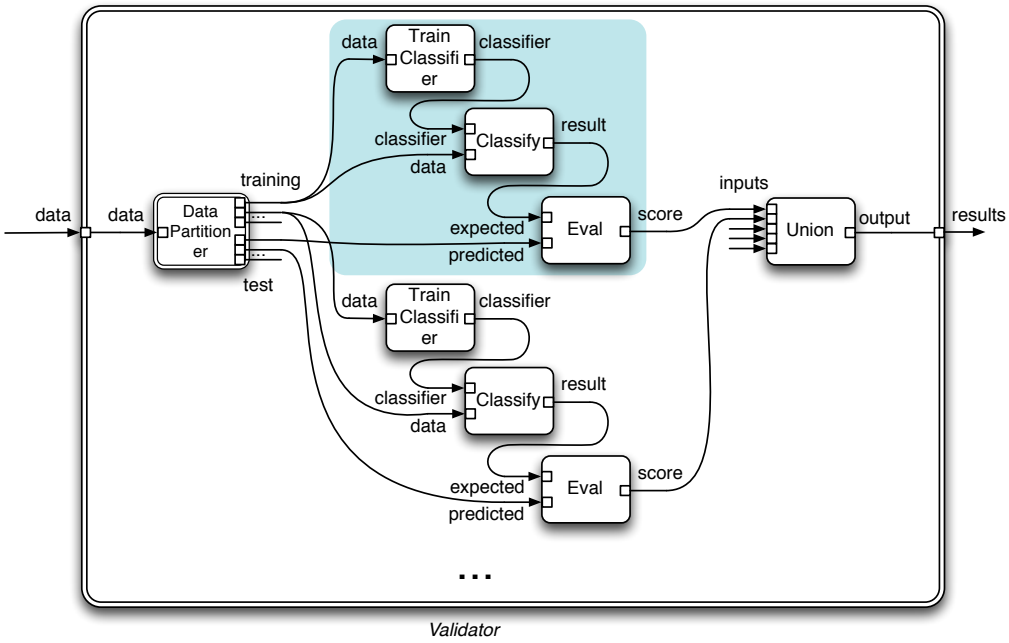


Figure 4.6: Part of the DISPEL graph generated from the DISPEL request in Figure 4.5.

```
1  package dispel.files {
2  //-------------------------------------------------------------------------
3  // The dispel.files package contains PEs supporting file manipulation.
4  //-------------------------------------------------------------------------
5    namespace db "http://dispel-lang.org/resource/dispel/db/";
6
7    Type ReadFile is PE (
8      <Connection: String:: "db:FileSystem" locator terminator source;
9       Connection: String:: "db:FilePath" terminator fPath> =>
10     <Connection: Byte[ ] data>
11   ) with lockstep(source, fPath), rate(fPath) == rate(data), @description =
12      "In response to each pair arriving on source and fPath the PE "      +
13      "delivers the contents of the referenced file on data. "
14   ;
15
16   register ReadFile;
17 }
```

Figure 4.7: The `ReadFile` PE of package `dispel.files` from Atkinson *et al.* [10].

there is a suitable converter PE is available to perform the type conversion [182]. Figure 4.8 shows a fragment of a DISPEL request that registers a new function to perform feature generation. The `makeFeatureGenerator` function has an image processing pipeline, which reads images from the file system, and performs rescaling and noise-reduction. The structural type of the `output` connection of `ImageRescale` is `BufferedImage`, but it successor, `MedianFilter` is expecting `Integer[][]` for its `input` connection. A `BufferedImage`-to-`Integer[][]` type-conversion PE can be inserted into the data stream to resolve the type conflict.

```
1  package book.examples.eurexpress {
2    ...   // Import PEs from dispel libraries
3    use dispel.files.ReadFile;
4    use book.examples.eurexpress.ImageRescale;
5    use book.examples.eurexpress.MedianFilter;
6    use book.examples.eurexpress.FeatureGeneration;
7
8    ...   // Create instances of PEs for workflow
9    ReadFile reader = New ReadFile;
10   ImageRescale rescale = new ImageRescale;
11   MedianFilter filter = new MedianFilter;
12   FeatureGeneration generator = new FeatureGeneration;
13
14   ...   // Connect PE instances to construct the workflow
15   reader.data => rescale.data;
16   |- 118 -| => rescale.width;
17   |- 190 -| => rescale.height;
18   rescale.output => filter.input;
19   |- 4 -| => filter.mask;
20   filter.output => generator.input;
21   ...
22
23   // Register new PE and function
24   register AbstractFeatureGenerator, makeFeatureGenerator;
25 }
```

Figure 4.8: A fragment of a DISPEL request to register a new feature generation function used in EURExpress workflow.

The *domain type* defines the interpretation of the permitted set of values transmitted along a connection using terms understandable to domain-experts. Each domain has its own agreed terms. The `namespace` keyword is used to refer to the existing ontology locations where the term definitions can be found. For instance, Line 5 in Figure 4.7 specifies the ontology where `db:FileSystem` can be found. The domain type describes how domain-experts interpret the data passing thought the PEs. The biologists will interpret the input data of `MedianFilter` as `MouseEmbryoImage`, while the earth scientists that use the same PE to de-noise a satellite image might name it as `SatelliteImage`.

The three-level type system provides more than just validation. These extra descriptions of the workflow components enable the exploration of optimisation techniques. The structure type exposes the low-level structure of the data streamed in the connections. For instance, the structure type of `ImageRescale` PE is a Java object, i.e. `BufferedImage`. Allocating both `ImageRescale` PE and `MedianFilter` PE on the same DIVM will enable the passing of references between PEs, instead of being serialised and sent between DIVMs. DISPEL enables the PEs designer to provide more information to the enactment engine about how to best implement the PEs. For instance, adding the `locator` (Line 8 Figure 4.7) informs the optimiser on where the instance of ReadFile

should be anchored during the enactment. Developers can use the `with` keyword to specify additional properties of the PEs. The input and output data rate of `ReadFile` is explicitly specified in Line 11. A detail discussion on the type system can be found in [10, 182] and the DISPEL reference manual [114].

4.3 Measurement framework

If you can not measure it, you can not improve it.

William Thomson.

The diversity and complexity of the scientific workflows has increased the difficulty of doing performance analysis. Truong *et al.* in [168] introduced a hierarchical abstraction for the performance analysis of workflows and identified performance metrics for different levels of abstraction in a workflow. We incorporated their model into our measurement framework.

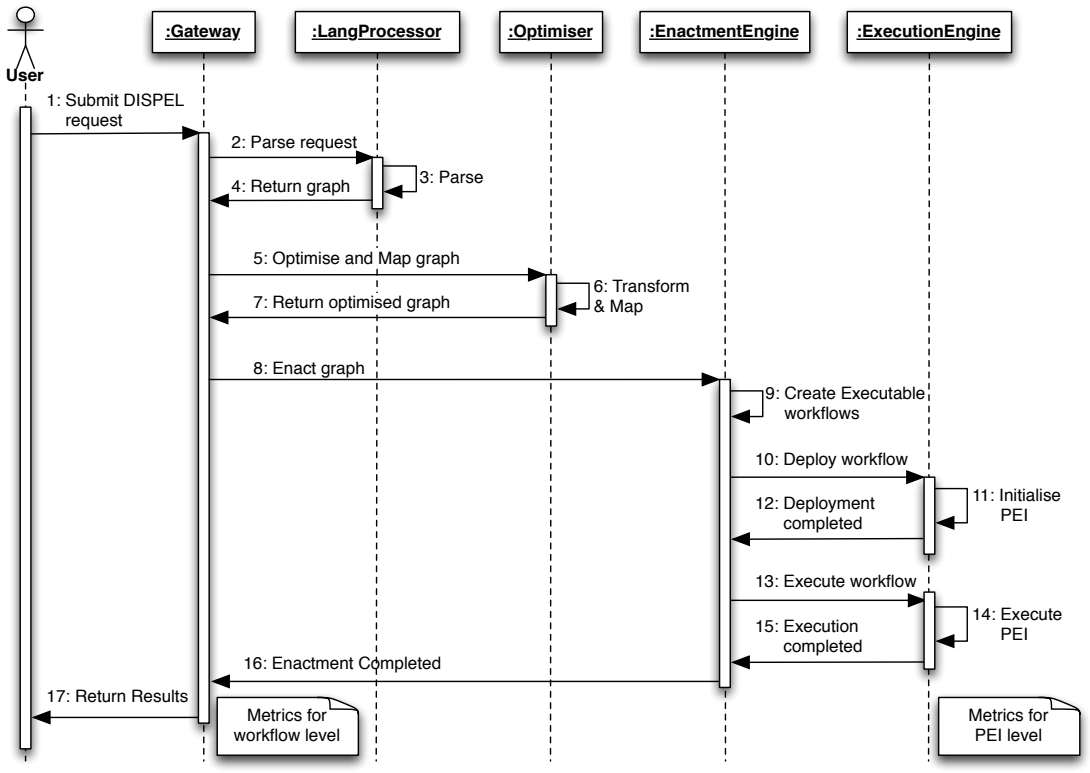
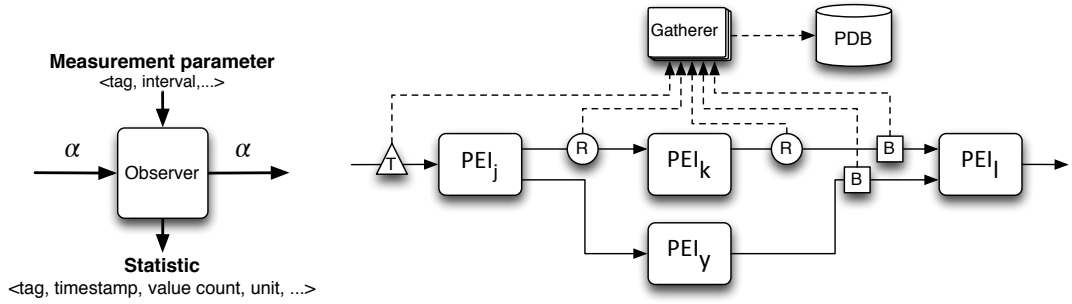


Figure 4.9: Execution model of a DISPEL request in a data-intensive architecture.

We classify the performance metrics into two levels of abstraction: *workflow level* and *PEI level*. We define different performance metrics for both levels. Figure 4.9 shows a multi-level abstraction for performance analysis during workflow enactment. At the

workflow level, we are looking at the *response time*. Workflow response time is defined as the time interval between a user request for a service and the response of the system [111]. Response time, is sometimes referred to as elapsed time and can be measured from the point of the user issuing a request and the system *starting* to respond or *finishing* its response. In our context, we refer to the period between the time a user submits a workflow to the gateway and the user receiving the execution results, which includes the overall computation time for the workflow processes, the time spent in IO and waiting in process queues, and any data delivery delays. Capturing the performance data at the workflow level is straightforward. At the gateway, the time between accepting a DISPEL request and returning results to user can be measured easily. We also observe the optimisation overhead.

At the PEI level, we are looking at both *unit cost* and *data rate*, where both metrics are used in our mapping algorithm ($\rightsquigarrow$*3*). In order to capture the performance data at this level, we have designed two measurement components:*observer* and *gatherer*.



(a) Design of Observer (b) The use of observers and a gatherer during enactment

Figure 4.10: PEI-level measurement components.

An observer receives data from input streams from a previous PE, performs a time-stamping, and outputs the data to the following PE without altering the data, as shown in Figure 4.10(a). By placing observers on the data streams, detailed enactment information can be captured and used for making appropriate optimisation decisions. In theory, we can have three types of observer, each with minimum impact on performance and a capability to capture performance data from a different perspective:

- *Type observer* is used to capture type information of the data flow on any given data stream. Together with the semantic information of the workflows, the type information may be useful in estimating the data transfer cost and determining the ability to split a data stream and process it in parallel. The type information should be collected prior to execution during DISPEL-language processing.

- *Rate observer* measures the data processing rate of data streams. When used in pairs, rate observers can capture the processing time per unit of data of a PEI. As shown in Figure 4.10(b), a rate observer is placed before PEI_k to capture its input data rate during the enactment. Together with output data rate measured by another rate observer, we can know the processing rate of PEI_k.

- *Buffer observer* is used to observe the buffer implementation of data streams. Buffers are used when there are variations in the data processing rate of any connected PEs. In Figure 4.10(b), buffer observers on the two input streams of PEI_l determine the rates at which data arrive on each stream from which we can infer the critical path of the workflow.

For the implementation of a data-intensive platform, the type observer is applied during the DISPEL language processing stage. When the DISPEL-language processor walks the generated graph verifying that source assertions and destination requirements about the structure types of values in the data stream are compatible, the input and output structural type of every PEI in the request will be recorded. Both rate observer and buffer observer are implemented during the execution stage, i.e. by observing the pipe buffer events in the data stream. During the enactment, the data producer of a data stream writes the data into the buffer, while the consumer reads data from it. Both operations will trigger different events. Another two interesting events to record are blocking from read and write.

The results collected from observers are sent to a *gatherer* which will insert these data into the PDB after the enactment is finished, as shown in Figure 4.10(b). To further reduce the overhead incurred during enactment, each gateway should have gatherers that run on separate execution engines to process the collected data on the fly, and insert the derived and hence much compressed performance data into the PDB. However, the platform should provide an option to keep all of the recorded events when data-intensive engineers are trying to trace the events that occurred for diagnostic purposes.

4.4 Performance database

This section describes the rationale behind the Performance database (PDB), and proposes a systematic way to implement it. The PDB[1] is designed to gather information at the level of PE class instances, so that we can determine how each class and data stream behaves. For instance, information collected from a previous enactment can

[1]Not to be confused with the Worldwide Protein Data Bank.

indicate whether co-locating certain PEs within the same execution engine will result in poor performance because these PEs are competing for the same resources. The use of performance data collected from previous enactments is worthwhile for two reasons. Firstly, domain experts tend to repeat similar enactment requests to iterate their understanding or to process multiple similar data samples. Secondly, there are many fundamental PEs that are used across domains and consequently appear in many enactment requests, e.g. those PEs from the DISPEL libraries.

4.4.1 Performance data life-cycle

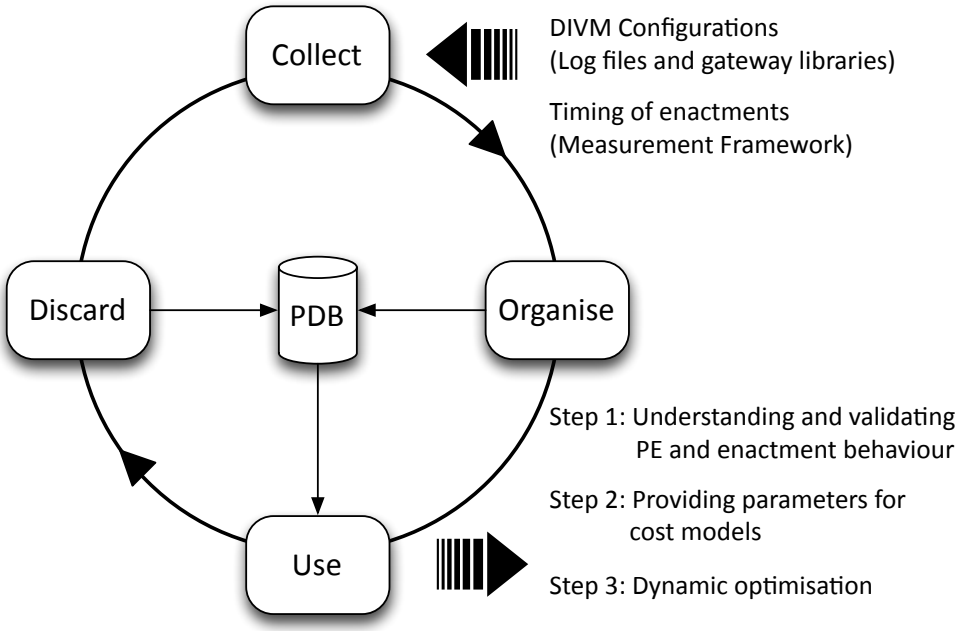


Figure 4.11: Performance data life-cycle.

We introduce a four-stage performance-data life-cycle: *collect*, *organise*, *use* and *discard*, to describe how the performance data are collected and transformed into information, as shown in Figure 4.11.

Collect stage We use *system logs* and the *measurement framework* to collect two types of data: configuration of DIVMs and timing of the actual enactments. System logs keep track of the activities involved in managing DIVM lifetimes. The data gathered from the log files allows reconstruction of the software and hardware stack at a given time. Logs are also generated to track which PEIs are running on each DIVM at a given time.

Organise stage Data collected in the first stage are organised in the PDB. The tables in the PDB are divided into three categories according to how their data are collected (see Figure 4.12). The first type of table stores data harvested from log files, e.g. DIVMInstance and DIVMInstallation. The second type of table stores data collected from measurement framework, e.g. DataStream. These two types of data are considered raw data. The final type of table stores derived data, e.g. PerfOfInstance, which are preprocessed by gatherer on all of the events recorded. These data are used in the calculation of the unit cost for any given PE on the DIVM where enactment occurred.

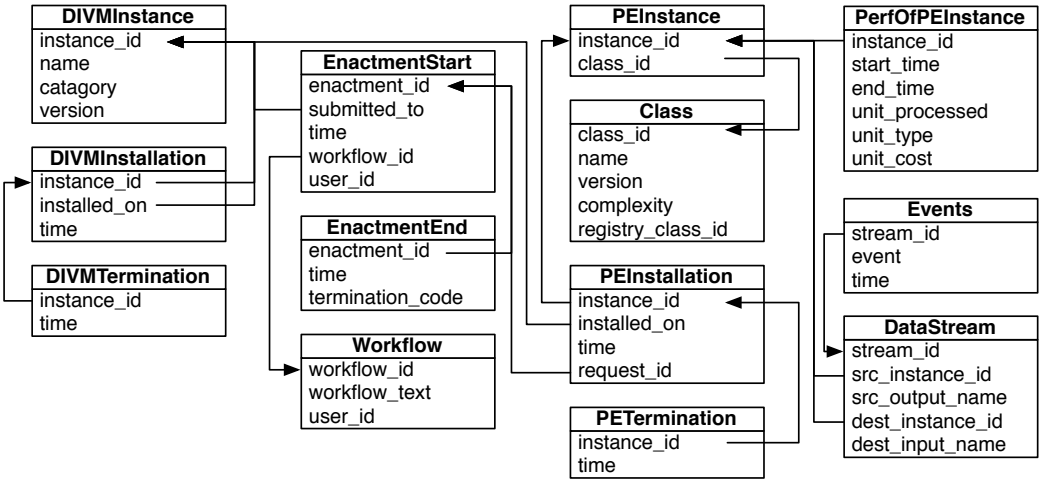


Figure 4.12: Logical content of the PDB.

Use stage The performance data gathered in the PDB are used in stages, as shown in Figure 4.11. For each stage, we formulate different sets of queries to access the PDB. The PDB data allow us to understand and validate hypotheses about PEs and their enactment behaviour, such as the type of data flow in the data stream[2] and the processing rate of PEs on different execution engines, through queries such as the following:

```
SELECT AVG(PerfOfPEInstance.unit_cost), MIN(PerfOfPEInstance.unit_cost),
       MAX(PerfOfPEInstance.unit_cost), COUNT(DIVMInstance.instance_id),
       DIVMInstance.instance_id
FROM PerfOfPEInstance, PEInstance, DIVMInstance, PEInstallation
WHERE PerfOfPEInstance.instance_id = PEInstance.instance_id
  AND PEInstallation.instance_id = PEInstance.instance_id
  AND PEInstallation.install_on = DIVMInstance.instance_id
  AND PEInstance.class_id = 'PEa'
  AND (DIVMInstance.instance_id = 'DIVM1' OR DIVMInstance.instance_id = 'DIVM2')
GROUP BY DIVMInstance.instance_id
```

[2]This may not be known *a priori* because the DISPEL sentences deliberately use Any and rest to suppress details, accommodate change and support domain specific formats. Once enactment is being considered, the optimiser needs to 'look under the hood'.

The query will retrieve all of the previous execution records of a PEI on all of the DIVM, filtered by DIVM_1 and DIVM_2, to find the more suitable DIVM on which to enact a PEI. This enables the optimiser to deploy PEs on the execution engines best able to support them in a heterogeneous environment.

Gray's Laws [162] on approaching the large-scale scientific data engineering challenge states:

1. Scientific computing is becoming increasingly data intensive.
2. The solution is in a "*scale-out*" architecture.
3. Bring computations to the data, rather than data to the computations.
4. Start the design with the "*20 queries*".
5. Go from "*working to working*".

We follow his approach and identify important questions that our PDB should answer.

Discard stage The size of the PDB is expected to grow rapidly. To sustain the performance of the PDB, a cleaning process is needed to remove out-dated or less important data. The PDB is cleaned in three ways:

1. by removing the raw data after the derivatives from those data were processed and stored,
2. by removing data associated with deprecated versions of a PE, and
3. by removing data that are obsolete (e.g. pertaining to a discontinued DIVM or after a predefine number of days).

A prototype of the PDB has been implemented in ADMIRE project [116].

4.4.2 Example queries

The "*20 queries*" informal rule formulated by Jim Gray was intended to bridge the semantic gap between the languages used by domain scientists and database engineers. However, this approach also helps in keeping the design process focused on the most important features the system must support. We follow his approach and have identified some of the basic questions for our PDB, as below:

Q1: What is the performance of instances of PE_a compared with instances of PE_b? Usage: To choose between two PEs with equivalent functionality.

Q2: Compare the performance of instances of PE_a depending on whether or not there is an instance of PE_b in the same DIVM.

Usage: To find out whether co-locating two PE instances on a same DIVM will result in loss of performance. This query can guide the decision of splitting workflow across multiple DIVMs to speed up the performance.

Q3: What is the performance of class PE_a across all DIVMs? (Do they process at a constant rate during enactment?)
Usage: To study the enactment performance of a given class PE. The query will retrieve all the previous execution records of a PE on all the DIVMs.

Q4: What is the performance of class PE_a on DIVM_1 compare with its performance on DIVM_2?
Usage: To find out the most suitable DIVM to enact a PEI. The query will retrieve all the previous execution record of a PEI on all the DIVMs. This query is similar to *Q3*, but filtered by DIVM_1 and DIVM_2.

Q5: Compare the performance of enacting instances of PE_a and instances of PE_b on different DIVMs?
Usage: To find out whether enacting 2 PE instances on a different DIVMs incur additional cost which will result in losing performance. Same as *Q2*, this query can guide the decision of splitting workflow across multiple DIVM to speed up the performance.

Q6: Find all of the PE instances that are running on a DIVM.
Usage: To find out the current workload of the particular DIVM.

Q7: Compare the performance of PEs for user A and user B.
Usage: To understand the workflows submitted by different users, for resource provisioning.

Q8: Find all of the workflow requests completed $> n$ days ago.
Usage: Used in Stage 4 to remove obsolete data.

Q9: Find all of the events that occurred during the execution of a workflow.
Usage: To count all of the blocked events in order to identify the slow processing PE instances. We will discuss this further in Section 5.3.1.

Q10: What is the performance of all of the PE instances of a workflow.
Usage: To study all of the PEs that are used to form a workflow. This information is used in the mapping algorithm and the graph transformation, i.e. identify slow processing pipeline for parallelisation.

From the list of questions above, we formulate the queries to extract information from the PDB. Below are the example queries for Question 1 and Question 2.

Q1: What is the performance of instances of PE_a compared with instances of PE_b?

```
SELECT AVG(PerfOfPEInstance.unit_cost),
       MIN(PerfOfPEInstance.unit_cost),
       MAX(PerfOfPEInstance.unit_cost),
       COUNT(PEInstance.class_id), PEInstance.class_id
FROM PerfOfPEInstance, PEInstance
WHERE PerfOfPEInstance.instance_id = PEInstance.instance_id
  AND (PEInstance.class_id = 'PEa' OR PEInstance.class_id = 'PEb')
GROUP BY PEInstance.class_id
```

Q2: Compare the performance of instances of PE_a depending on whether or not there is an instance of PE_b in the same DIVM.

If two instances are associated with the same request_id, then we can infer that they co-exist at the same time and potentially overlap in their use of a DIVM. This is achieved with queries below:

Construct a view, PEIonDIVM *that joins data from related tables.*

```
CREATE VIEW PEIonDIVM AS
SELECT Class.name AS pe, PEInstallation.instance_id AS pei,
       PEInstallation.request_id, DIVMInstance.name AS divm,
       PerfOfPEInstance.unit_cost
FROM PEInstance, PEInstallation, PerfOfPEInstance, Class, DIVMInstance
WHERE PEInstallation.instance_id = PEInstance.instance_id
  AND DIVMInstance.instance_id = PEInstallation.install_on
  AND PEInstance.class_id = Class.class_id
  AND PerfOfPEInstance.instance_id = PEInstance.instance_id;
```

Construct a view, Co_located *to compute a subset of* PEIonDIVM *where the* request_id *and* divm *are equal.*

```
CREATE VIEW Co_located AS
SELECT DISTINCT a.pe,  a.pei, a.time_per_unit, a.request_id,
                a.divm, 'TRUE' AS co_located
FROM PEIonDIVM a
INNER JOIN PEIonDIVM b
ON a.request_id=b.request_id AND a.divm=b.divm
WHERE (a.pe='PEa' AND b.pe='PEb') OR (a.pe='PEb' AND b.pe='PEa');
```

Construct another view, Not_co_located *to compute a subset of* PEIonDIVM *where the* request_id *and* divm *are not equal.*

```
CREATE VIEW Not_co_located AS
SELECT pe, pei, unit_cost, request_id, divm, 'FALSE' AS co_located
FROM PEIonDIVM
WHERE pe='PEa' AND pei NOT IN (SELECT pei from Co_located)
UNION
SELECT pe, pei, time_per_unit, request_id, divm, 'FALSE' AS co_located
FROM PEIonDIVM
WHERE pe='PEb' AND pei NOT IN (SELECT pei from Co_located);
```

At last, compute the performance from the UNION *of both views above.*

```
SELECT AVG(u.unit_cost), MIN(u.unit_cost), MAX(u.unit_cost),
       u.pe, u.co_located, COUNT(u.pe)
FROM (SELECT * FROM Co_located
      UNION
      SELECT * FROM Not_co_located) AS u
GROUP BY pe, co_located;
```

4.5 Summary of data-intensive architecture

In this chapter, we have discussed the underlying architecture to support the optimisation model proposed in Chapter 3. The data-intensive architecture is designed to support the diversity and complexity challenges in workflow enactment (⇝*2.1*). We have described all of the three crucial components of the architecture that provides the separation of concerns: a novel and powerful process engineering language (DISPEL), a registry that provides rich semantic descriptions, and an extensible and robust enactment platform that supports the data-intensive computations on distributed and heterogeneous environments. We have also presented our design of measurement framework and performance database, that are the information infrastructures supporting the optimisation of streaming workflows.

The data-intensive architecture was developed under the ADMIRE project and its prototype is available as open source[3]. The data-intensive platform is built on top of OGSA-DAI, a framework for building distributed data access and management systems [55]. PEs are implemented as OGSA-DAI activities, i.e. basic building blocks for OGSA-DAI workflows. The concrete workflows generated from DISPEL requests are OGSA-DAI XML workflows, which are then submitted to OGSA-DAI servers (i.e. the execution engines) for execution. We have designed, implemented and used the measurement framework and PDB in the course of this thesis, and adopted them in this prototype.

The gateway is developed using the Spring framework[4] by the colleagues in EPCC[5] and FLE[6]. The mapping algorithm is implemented as a servlet that is plugged into the gateway. The registry was designed and built by our collaborators from the Ontology Engineering Group in UPM[7].

[3] ADMIRE prototype: `http://sourceforge.net/projects/admire/`
[4] Spring: `http://www.springsource.org/`
[5] Edinburgh Parallel Computing Centre: `http://www.epcc.ed.ac.uk/`
[6] Fujitsu Laboratories of Europe: `http://www.fujitsu.com/emea/about/fle/`
[7] Ontology Engineering Group, Universidad Politécnica de Madrid: `http://www.oeg-upm.net/`

The DISPEL design and implementation is an ongoing work in the Data-Intensive Research Group in the School of Informatics[8]. The language parser development is in progress, with all of the basic functionalities implemented. Some of the features described in this chapter are not available in the current prototype, e.g. the automatic insertion of converters. Thus, we have made certain tweaks in the prototype for our experiments.

The next chapter will discuss the experiments that have been conducted in evaluating the architecture.

[8]Edinburgh Data-Intensive Research Group: `http://research.nesc.ac.uk/`

CHAPTER 5

Experiments and results

This chapter discusses how we plan and conduct our experiments to evaluate the model and the architecture. Section 5.1 describes the use cases from the ADMIRE project, that are used in populating the performance database (PDB), and as the test workload for our experiments. The experimental apparatus is discussed in Section 5.2. We have conducted our experiments in three phases, with each focusing on a particular aspect of our work. The first phase aims to study the behaviour of streaming workflows and examine the ability of the measurement framework in capturing performance-related data during their enactments (⇝*5.3.1*). The second phase aims to demonstrate the use of performance data in optimisation, with hand-crafted parallelisation (⇝*5.3.2*). The third phase aims to evaluate our proposed three-stage mapping algorithm and demonstrates the use of performance data in DIVM allocation (⇝*5.3.3*). We have designed a framework for evaluating the proposed algorithm with real-world workloads taken from scientific communities. However, the full exploration of this evaluation framework requires substantial time and resources is impractical to be conducted within the course of this study. Thus, we have selected a reasonable size workflow and modest computing resources for the preliminary evaluation experiments. These generated evidence that the approach is promising. The complete experimental plan for an extensive evaluation is suggested in Section 5.4.

5.1 Experiment use cases

There are two real-world use cases from different scientific domains that are used as the test workload for the experiments. The first use case is from developmental biology, a project named EURExpressII and the other is from seismology on seismic ambient

noise correlation. Both use cases are selected for two reasons: they are use cases studied in the ADMIRE project in which we have built up the understanding of the problem, and they are considered as typical scientific workflows that are suitable to demonstrate our work.

5.1.1 EURExpressII

The EURExpress-II project [92] aims to build a transcriptome-wide atlas of gene expression for the developing mouse embryo established by RNA *in situ* hybridisation. The project annotates images of the mouse embryos by tagging images with terms from the ontology for mouse anatomy development. The data consists of mouse embryo image files and an annotation database (in MySQL) that describes the images. To date, 4 TB of images have been produced and 80% of the annotation is done manually by human curators. Based on 600 MB that we have received, we will produce multiple classifiers where each classifier recognise a gene expression from a set of 1,500 anatomical components to classify the remaining 20% of images (85,824 images) automatically. The overall EURExpress-II automated annotation task is divided into 3 stages: *training*, *testing* and *deployment*. The training and testing stage are performed together in a workflow. Dataset are split into 2 parts: for training a classifier and for testing the accuracy of the trained classifier using the k-fold cross-validation pattern described earlier (⇝*4.2.3*). The classifier will then be deployed to classify the remaining data.

Figure 5.1 shows the overall data-mining workflow of the EURExpress-II use case, which can be simplified as below:

1. Read raw image files and the annotation database.
2. Image Scaling: Scale selected images to a standard size (320×200 pixels).
3. Noise Reduction: Apply median filtering to reduce the image noise.
4. Feature Generation: Using wavelet transformation, generate the image features as matrices of wavelet coefficients. 64,000 features are generated per image of 320×200 pixels.
5. Feature Selection: Reduce the features set by selecting the representative features for constructing classifiers using Fisher Ratio analysis [62], i.e. 24 most significant features are extracted from 64,000 features generated in step 4.
6. Classifier Design: Build a separate classifier for each anatomical feature which takes image features as input and output a rating of '*not detected*', '*possible*', '*weak*', '*moderate*' or '*strong*' for an anatomical features (eyes, nose, etc).

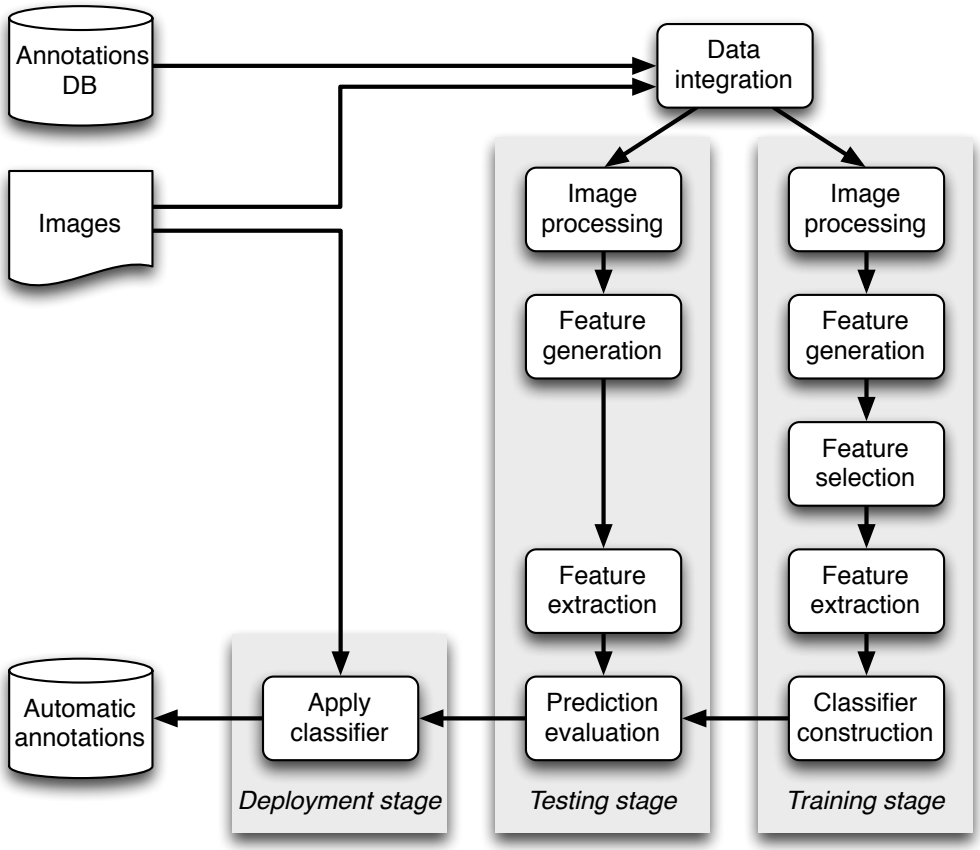


Figure 5.1: High-level EURExpress-II image annotation workflow.

7. Evaluation: Test the classifier built in step 6 against a partition of the data not used in the preceding steps but already classified.

5.1.2 Seismology

We are working closely with the seismologists from Royal Netherlands Meteorological Institute[1] in the previous ADMIRE project, and now the VERCE[2] project. This collaboration gives us the opportunity to explore the data-intensive challenge in the seismology, that deals with massive and distributed sensor data.

The seismic interferometry use case studies the interference of pairs of seismic signals, originated from earthquakes or other seismic sources. This research focuses on the *seismic ambient noise processing* and looks into automated cross-correlation and aggregation (known as stacking) of distributed seismic wave forms [44]. Seismic signals received at two geographically locations are cross-correlated to produce a Green's func-

[1]Royal Netherlands Meteorological Institute: `http://www.knmi.nl/`

[2]Virtual Earthquake and seismology Research Community e-science environment in Europe (VERCE): `http://www.verce.eu/`

tion between these receivers—a waveform that would be observed at one location if the signal source were placed at the other. The post-processing of the stacked signals used in many applications, including ambient noise tomography, which is applied to determine dispersion measurements of surface waves, to create tomographic maps [66].

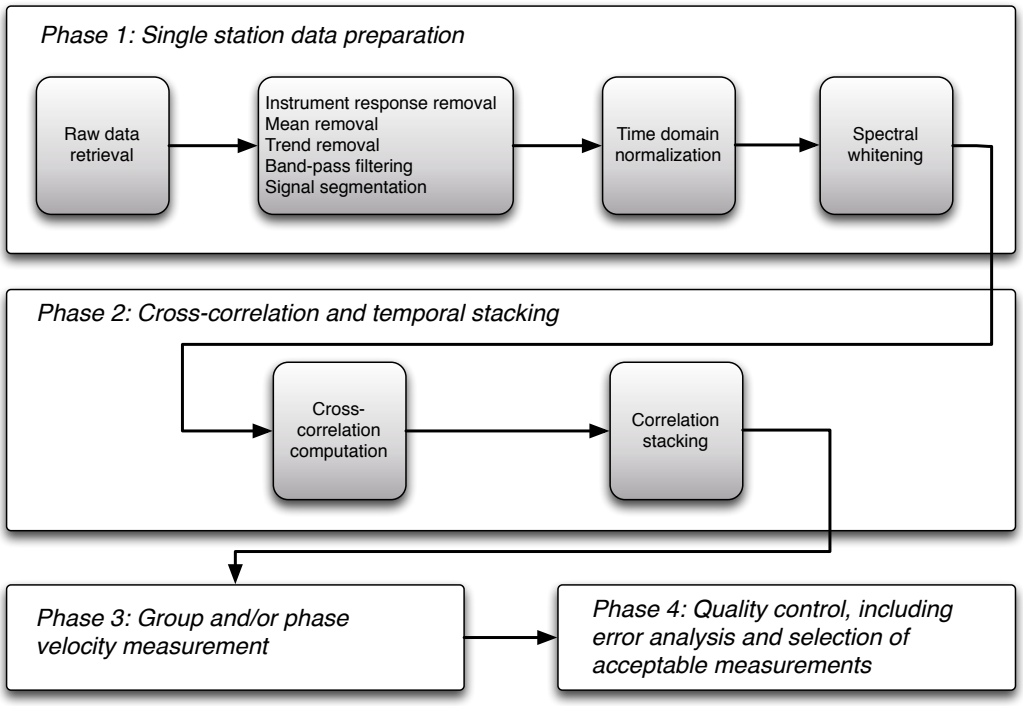


Figure 5.2: Phases of the seismic ambient noise processing procedure from Galea *et al.* [66].

Figure 5.2 shows the four phases of the ambient noise signal processing procedure that are defined by Benson *et al.* [20]:

1. *Single station data preparation*: removes other signals and irregularities that obscure the ambient noise, and applies time-domain normalisation and spectral whitening.

2. *Cross-correlation and temporal stacking data*: performs cross-correlation between all possible station pairs on a day-long segment, and stacks them to correspond to a longer time-series, e.g. from daily segments into weekly, monthly and yearly. The resulting waveform of this phase is an estimated Green function.

3. *Measurement of dispersion curves*: uses traditional frequency-time analysis to measure the group and phase speeds.

4. *Quality control*: identifies and rejects bad measurements and computes quality assurance statistics for the accepted measurements.

In the ADMIRE project, we have implemented the first two phases to exploit the data-intensive architecture. These workflows are used as the test workload for our model and architecture evaluation.

5.2 Experimental apparatus

5.2.1 Hardware

We have chosen three different types of computing resources that are available and accessible for setting up the testbed: *a*) Edinburgh Compute and Data Facility, *b*) Edinburgh Data-Intensive Machine 1, and *c*) workstations.

Edinburgh Compute and Data Facility (ECDF)[3] is a high performance computing and large-scale storage facility in the University of Edinburgh. ECDF has a high-performance cluster based on IBM's iDataplex technology, which comprises main phases. The first phase consists of 130 worker nodes with two quad-core processors and large random access memory (RAM), and connected with Gigabit Ethernet (with a 10 Gigabit Ethernet backbone). The specification of the worker nodes is shown in Table 5.1. The second phase is another 156 more powerful nodes, each equipped with two six-core processors (68 of them are also connected by Infiniband network which is suitable for MPI jobs).

Component	Worker Node	Storage Server
Processor	2 × Intel E5620 (4 core 2.4 GHz)	2 × Intel E5620 (4 core 2.4 GHz)
Memory	24 GB DDR3	48 GB DDR3
Storage	1 × 250 GB 7200 RPM HDD	

Table 5.1: Specification of ECDF nodes.

The computing cluster is connected with a HPC storage that uses IBM's General Parallel File System. The HPC storage is a three-tier storage system with a total capacity of 163 TB, as shown in Table 5.2. The cluster nodes access the parallel file system through eight storage servers.

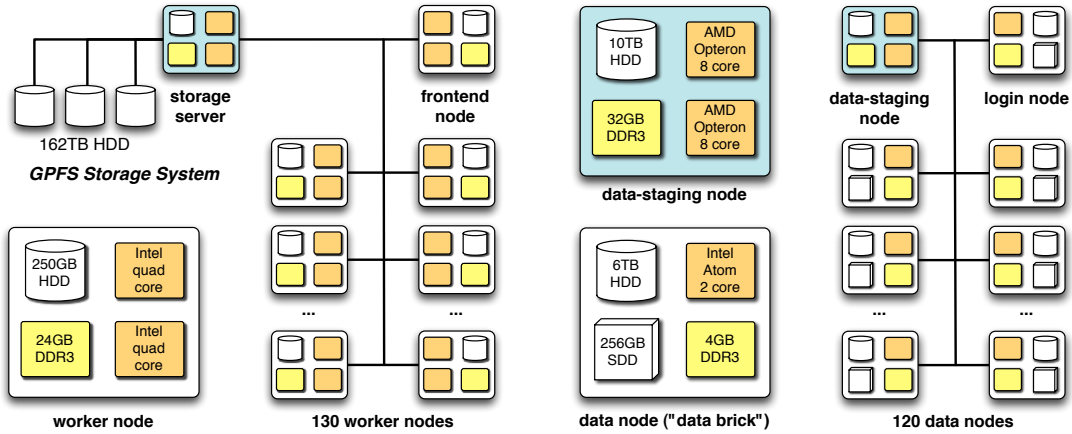
The computing machines are setup as a Linux Beowulf master-worker cluster, that supports batch-job processing. Jobs are submitted to the scheduler in the frontend node (see Figure 5.3(a)) . Each of the cluster nodes has a powerful processing capability but with limited storage space. This gives good performance for running naturally parallel

[3]ECDF: `http://www.ecdf.ed.ac.uk/`

Tier	Disk Specification	Usage
Tier 0	950 GB SSD	Metadata, e.g. filesystem structure and folders
Tier 1	74 TB 15000 RPM HDD	Live data, i.e. new files
Tier 2	88 TB 7200 RPM HDD	Aged files and large sequential files

Table 5.2: ECDF three-tier storage system.

computation, where each individual task can execute with minimal communication with others. However, this setup has raised two important issues for the data-intensive workflows that we are trying to deal with. First, the cluster is designed for task-based batch processing and not meant for service-based streaming processing model, which is central to our research[4]. Second, the storage space is sitting on a separate storage network, and data are staged into the worker nodes for computation. This is violating *Gray's Law* for data-intensive science: "bring computations to the data, rather than data to the computations"[162].



(a) ECDF phase 1 computing cluster connected with HPC storage system

(b) EDIM1

Figure 5.3: Layout comparison between ECDF and EDIM1.

Jim Gray suggested the idea of building a data store made of cheap "*data bricks*" [85], where each node has modest processing capability with significant storage. The facility can easily scale out by adding more data bricks to support application with larger data size. Edinburgh Data-Intensive Machine 1 (EDIM1) is built based on the data-bricks architecture [130]. EDIM1 comprises 120 data nodes, each has a low power consumption Intel Atom processor but a large storage space, i.e. three 2 TB hard disk drives (HDD) and a 256 GB solid-state drive (SSD). Table 5.3 lists the specification of EDIM1 nodes. The data nodes are independent from each other and connected with a

[4]Section 5.4 suggests how pilot-job mechanism can be used to solve this problem.

Component	Data-staging Node	Data Node
Processor	2 × AMD Opteron 6128 (8 core 2 GHz)	Intel Atom (2 core 1.6 GHz)
Memory	32 GB DDR3	4 GB DDR3
Storage	5 × 2 TB 7200 RPM HDD	3 × 2 TB 7200 RPM HDD 1 × 256 GB SSD
GPU	-	NVIDIA Ion

Table 5.3: Specification of EDIM1 nodes.

Gigabit Ethernet. These data nodes are organised in 3 racks of 40 nodes each, where racks are connected with a 10 Gigabit Ethernet. The 6 TB HDD can hold a large volume of data and the SSD provides excellent performance for random access to the data. If the ECDF is a HPC facility for compute-intensive applications, then EDIM1 is tailored for I/O-intensive applications.

Figure 5.3(b) shows the layout of EDIM1. The login node is the entry point to the data nodes. Besides the low-power data nodes, there is another dedicated data-staging node that has two 8-core processors and a large storage space, i.e. 10 TB, see Table 5.3. The data-staging nodes has three functions: *a*) temporarily holds the data prior to their deployment onto data nodes, *b*) hosts metadata, e.g. performance database and data catalogue, and *c*) executes compute-intensive jobs.

Besides the ECDF and EDIM1, we also use some dedicated workstations in our experiments. These workstations are connected with a Gigabit Ethernet:

- 8 × Intel Core Duo 2.4 GHz with 2 GB RAM and 320 GB HDD,
- 2 × Intel Core 2 Duo 3.0 GHz with 8 GB RAM and 2.2 TB HDD, and
- 2 × Intel Core 2 Quad 2.66 GHz with 8 GB RAM and 2.2 TB HDD.
- 1 × Intel Core 2 Quad 3.0 GHz with 16 GB RAM and 1 TB HDD.
- 3 × Intel Xeon Quad 2.2 GHz with 16 GB RAM and 8 TB HDD.

5.2.2 Software

Table 5.4 is a summary of the major software elements used in the experiments.

There are other software and tools pre-installed on the hardware we use. This includes cluster management software (Rocks Open-source Toolkit), cluster monitoring software (Ganglia), parallel file system (IBM GPFS) and job scheduler (Sun Grid Engine version 6.2 update 5). The list of software does not include the application-specific software for the use cases, such as NumPy, SciPy and Automatically Tuned Linear Algebra Software that are used in the seismology experiments.

Software	Usage
ADMIRE platform and tools	For DISPEL request submission (gateway client and ADMIRE workbench), DISPEL request processing and enactment (gateway service), DISPEL parsing (DISPEL language parser) and semantic information store (registry).
OGSA-DAI server (version 3.1, 4.0, 4.1 and 4.2)	Used as the execution engine for data-intensive architecture.
Java Development Kit (version 1.6)	For developing concrete implementation of PEs and the prototype architecture.
Jakarta Tomcat (version 5.5)	For hosting the ADMIRE web services, i.e. gateway and execution engines.
DBMS	For hosting performance database (PostgresSQL version 9.0) and storing use-case data (MySQL version 5.0).
Linux OS	The operating system for ECDF cluster (Scientific Linux version 5), EDIM1 (CentOS version 5.5), and workstations (Ubuntu version 11.04).

Table 5.4: Essential software used for conducting the experiments.

5.3 Experiments

The experiments are conducted in three phases. We will discuss how these phases are being conducted in separate sections. In each section, we describe:

- the hypothesis,
- the experiment use cases,
- the evaluation method, including how we setup and measure the experiment,
- the experimental procedure, including a step-by-step list of everything to be performed and how many times it is repeated, and
- the results and our observations.

5.3.1 Streaming processing model and measurement framework

We have conducted two experiments in this phase to study the behaviour of the streaming processing model and to evaluate the capability of the measurement framework in capturing these fine-grained performance data.

Hypothesis: Our measurement framework is capable to record fine-grained measurement-related data to support the optimisation model.

5.3.1.1 Experiment 1

Use cases: EURExpress-II classifier training and testing workflow.

Evaluation method: We built the observer and gatherer (⇝*4.3*) as independent PEs which are inserted into the workflow to measure enactment performance. We designed the EURExpressII workflow, shown in Figure 5.4, to be executed on a single OGSA-DAI server. Nine PEs are identified to be the measurement target (marked are PE_1 to PE_9 in the figure). The observer will capture the precise time when a data item is read by each of the PEs, and when a unit of output data is produced. Both raw mouse embryo images and the annotation database are deployed on the node prior to the experiment.

Experimental procedure:

Set experiment parameter, $param = \{800, 1600, 3200, 4800, 6400, 12800, 19200\}$.
for all image sizes defined in *param* **do**
  Launch OGSA-DAI server on a workstation.
  Using a client machine, submit the EURExpress-II workflow.
  Store the performance data into individual text file.
end for

The experiment started with 800 images as the training and test dataset. The number of images is changed for each experiment iteration, i.e. 800, 1600, 3200, 4800, 6400, 12800 and 19200.

Results and observations:

The observer PEs have captured the time when a data item is writen into the input data stream of a targeted PE, and when it is read from its output data stream. Figure 5.5 shows the trace of the processing of every single data item by the PEs. For instance, the PE4 is captured by a rate observer placed at the output of ImageRescaleActivity.

The data recorded by the observers are useful to understand the behaviour of the PEs. The graph shows that PE_1, PE_2 and PE_3 are relatively fast-processing PEs, as compared with the PE_4, PE_5, PE_6 and PE_8. The latter type of PEs should be first considered if parallelisation is applied—in the subsequent experiments, we perform data parallelisation by having multiple [PE_4, PE_5, PE_6] pipelines running in parallel.

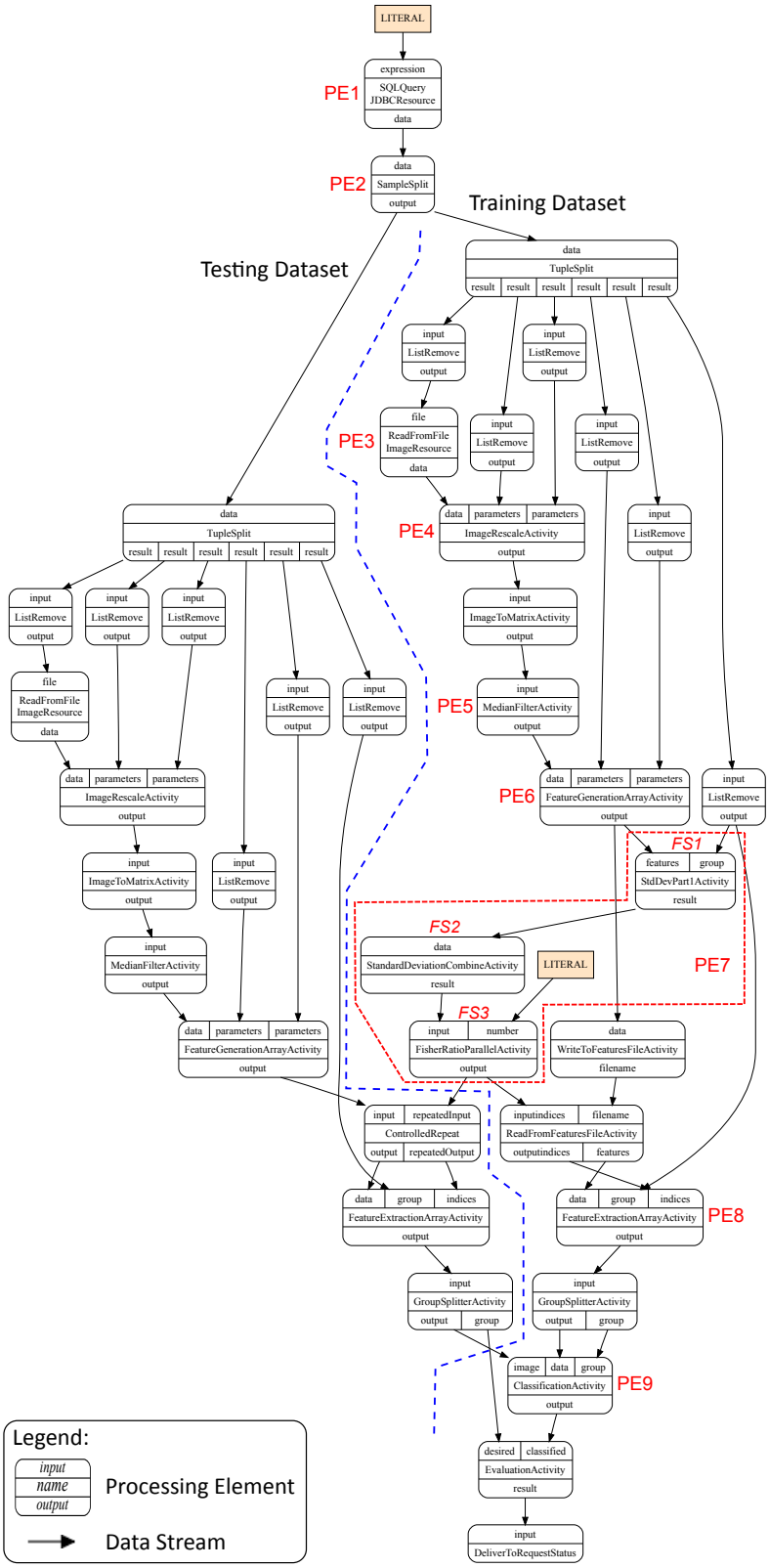


Figure 5.4: Implementation of PEs in EURExpress-II workflow as OGSA-DAI activities.

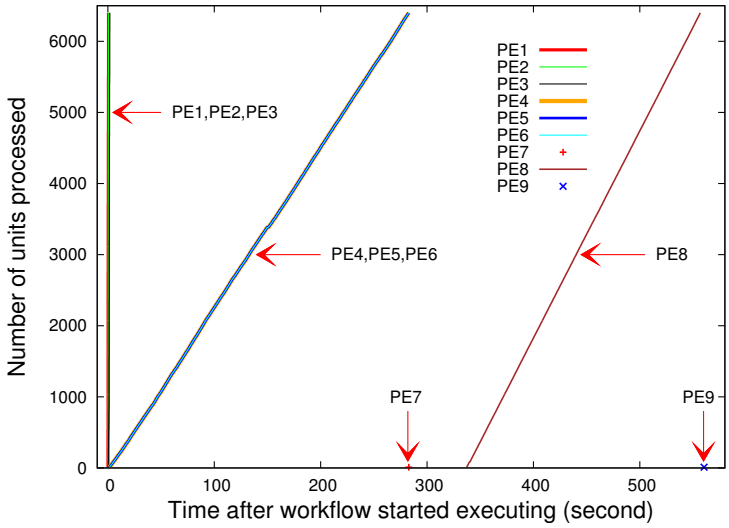


Figure 5.5: Trace of PEs execution in single machine with 6400 images.

Another important observation is the trace of PE_7 and PE_9. Both PEs mark a single entry on the graph because of their aggregative behaviour. PE_7 reads all of the generated features to construct a matrix, before applying the Fisher Ratio calculation to extract the most significant features set, which is an array of Integer values showing the indices of these features. Another aggregative PE, PE_9, reads all of the training data and produces a predicted classification for any given test image. PE_9 produces classification results in an Integer array. PE_7 and PE_9 have broken the streaming processing model as they read a single input data item and produce a single output data item. These two PEs require the system to accumulate all of the input data item before they start producing a result. Thus, in an heterogenous computational environments, these PEs have to be mapped on the fastest machines with large memory.

5.3.1.2 Experiment 2

Use cases: General data transformation workflow used in the various applications of the ADMIRE project.

Evaluation method: We have deployed the measurement probes (⇝*4.3*) in several ADMIRE gateways run by the members of the ADMIRE project, to collect performance-related data for the enactment of real-world workflows. We embedded the observer in the stream implementation of the execution engines to record the events occurring in the stream buffer (⇝*3.4.1*). We have developed the gatherer as a web service. The gatherer is deployed on the gateway to gather all of the performance data and populate them to the PDB (⇝*4.4*).

Experimental Procedure:

This experiment involves several testbeds used in the ADMIRE project. For each of the testbeds, we conduct the following procedure:

Deploy the execution engines with embedded buffer observer.
Start a gateway service connecting all of these execution engines.
Start a gatherer service to gather all of the collected data.
for all DISPEL requests that are submitted for enactment **do**
  for all PEIs in each of the DISPEL request **do**
    Capture the buffer events occur during the execution.
    Send the recorded data to gatherer.
  end for
  The gatherer service insert the data to the PDB.
end for

Results and observations:

We have collected the performance data for the enactment of workflows from seismology, functional genetics, astronomy and hydrology domain. In this discussion, we have selected a data integration and transformation workflow that is frequently found across domains; a simple example is shown in Figure 5.6.

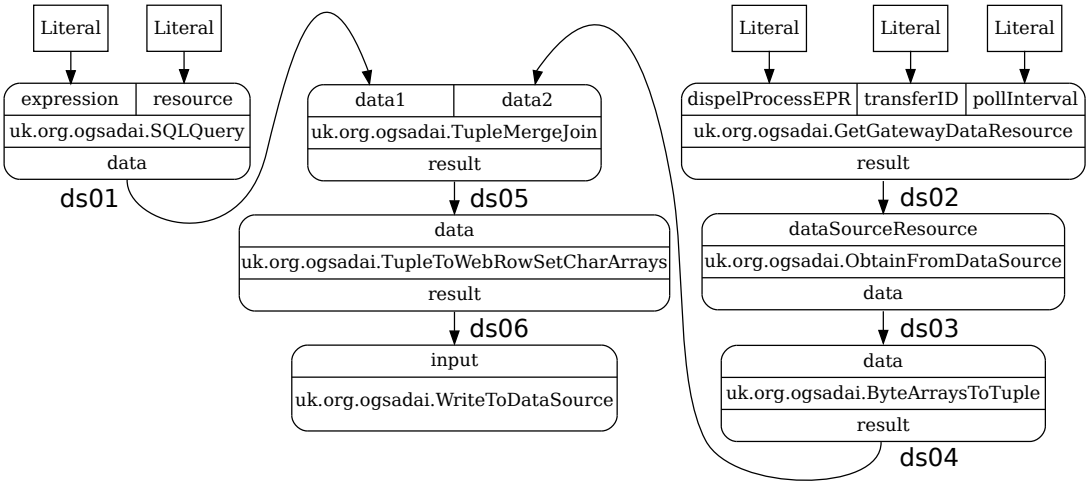
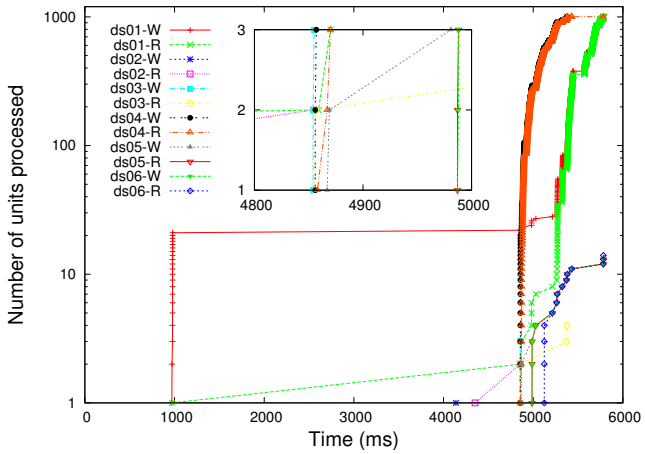


Figure 5.6: Common data integration workflow.

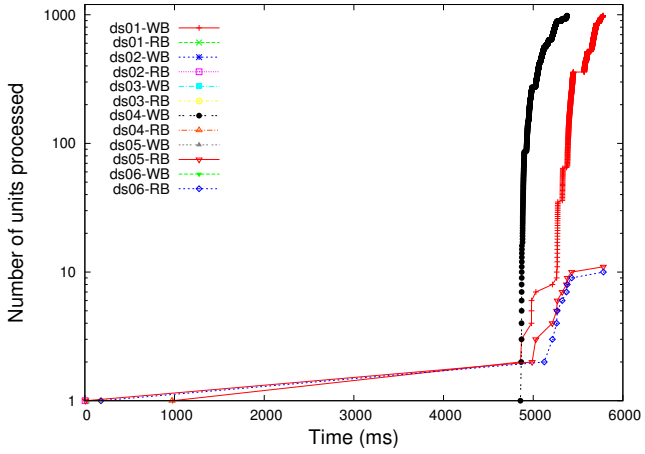
This workflow is a typical distributed, generalised query workflow. This pattern of a tree of paths selecting and integrating data is a common subgraph in the scientific workflows, e.g. integrating signal data from all seismic stations. It involves:

- *retrieving* data from distributed and heterogeneous data sources (Retrieve data from a database using SQLQuery and streams of data from a remote data source using ObtainFromDataSource),
- *merging* two data streams (Join two data streams using TupleMergeJoin),
- *transforming* the data streams (Transforming tuples output into XML WebRowSet format using TupleToWebRowSetCharArrays), and
- *delivering* results (Write the result into a data source using WriteToDataSource).

The data collected by all of the buffer observers are sent to a gatherer service sitting on the gateway, and populated into the PDB. Graphs in Figure 5.7 are the event trace during the enactment which is extracted from querying these tables (note that the vertical scale is logarithmic).



(a) Write and Read events



(b) WriteBlocked and ReadBlocked events

Figure 5.7: Events trace for data streams used in the workflow.

Figure 5.7(a) shows the trace of read and write events. When SQLQuery has executed and has started to write data into its output data stream (i.e. ds01-W), which is later read by TupleMergeJoin in input data1 (i.e. ds01-R). After performing 21 writes, SQL-Query is blocked from writing (the default buffer size is 20 blocks of data). The blocking continues because TupleMergeJoin is waiting the data from from the other input data2, which is produced by another pipeline. GetGatewayDataResource produces the first result 4138 milliseconds after workflow is started. TupleMergeJoin finally read a data item at input data2 (i.e. ds04-R) at 4856^{th} millisecond, and produce the first merging result at 4867^{th} millisecond (i.e. ds05-W). Figure 5.7(b) shows the trace of the blocked events. Both of the figures have demonstrated the fidelity of the performance data that are captured by the measurement framework.

This is a simple example of a typical optimisation challenge, which is to identify which branch in the graph (subDAG) is causing a delay. One of the ways to identify this problem is by looking at the *blocked events*. We executed the query below to count the events that occurred during the enactment to plot the graph shown in Figure 5.8 (note that the vertical scale is logarithmic).

```
SELECT PEInstance.class, Events.event, count(Events.event) AS NumOccurences
FROM PEInstance, Events WHERE stream_id IN
     (SELECT DISTINCT DataStream.stream_id
      FROM DataStream, PEInstance, PEInstallation
      WHERE (DataStream.src_instance_id = PEInstance.instance_id
          OR DataStream.dest_instance_id = PEInstance.instance_id)
        AND PEInstance.instance_id = PEInstallation.instance_id
        AND PEInstallation.request_id = 'request_id')
GROUP BY Events.event;
```

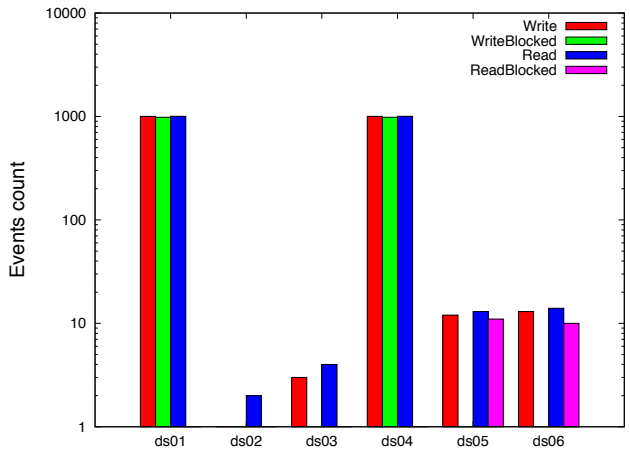


Figure 5.8: Events count for data streams used in the workflow.

When we traced the events that occur in all of the data streams, we find that Tuple-

MergeJoin received data from two input streams and there is clearly a long wait for the arrival of data from one of its streams. Observe that the two data streams into TupleMergeJoin suffer a large number of write blocks and that the remaining streams suffer negligible read blocks. This is because the data processing rate of TupleMergeJoin is slower than both of its predecessors. Thus, we can conclude that TupleMergeJoin has a relatively large unit cost because of the WB events recorded by its predecessors (SQLQuery and ByteArraysToTuple) and the RB events recorded by its successor (TupleToWebRowSetCharArrays). This example has demonstrated the value of monitoring buffer events to locate the parts of an abstract workflow that contribute most the response time.

5.3.2 Optimisation of workflow enactment with parallelism

In Section 3.2.1, we have discussed possible graph transformations for optimising the enactment. In this phase, we conduct experiment to investigate parallelism as an optimisation strategy for streaming workflows.

Hypothesis: Splitting a data-streaming workflow into parallel streams and enacting them on multiple machines yields linear speedup.

Use cases: EURExpress-II classifier training and testing workflow.

Evaluation method: We have redesigned the workflow showed in Figure 5.4 for parallel execution. This is one fold of the 8-fold cross-validation, where the dataset is divided into 8 subsets, as shown in Figure 5.9. The training pipeline, [PE_3, PE_4, PE_5, PE_6 and FS_1], is split and executed on multiple workstations. The results of all of the FS_1 instances are merged by FS_2 before proceeding with the Fisher Ratio calculation on FS_3. OGSA-DAI server does not provide the function to split and execute parallel workflows. Thus, we have performed the split manually, and created two sets of workflows. This is a simulation of the kind of graph transformation discussed earlier.

Figure 5.10 illustrates the execution plan of the workflow on the eight DIVMs. 7 DIVMs (M1–M7) are used to process the training dataset. Each DIVM will receive a subset of the training data, read the selected image files, re-scale and de-noise the retrieved images, generate features, and perform partial feature selection (FS1). The result of FS1 will be sent through the network to M8, and combined with result from other DIVMs for the remaining feature selection calculation (FS2 and FS3). The indices of the selected features are sent back to M1–M7 for feature extraction. Finally, M8 will receive and combine (C) all of the extracted features of the training dataset and perform classification on its testing dataset.

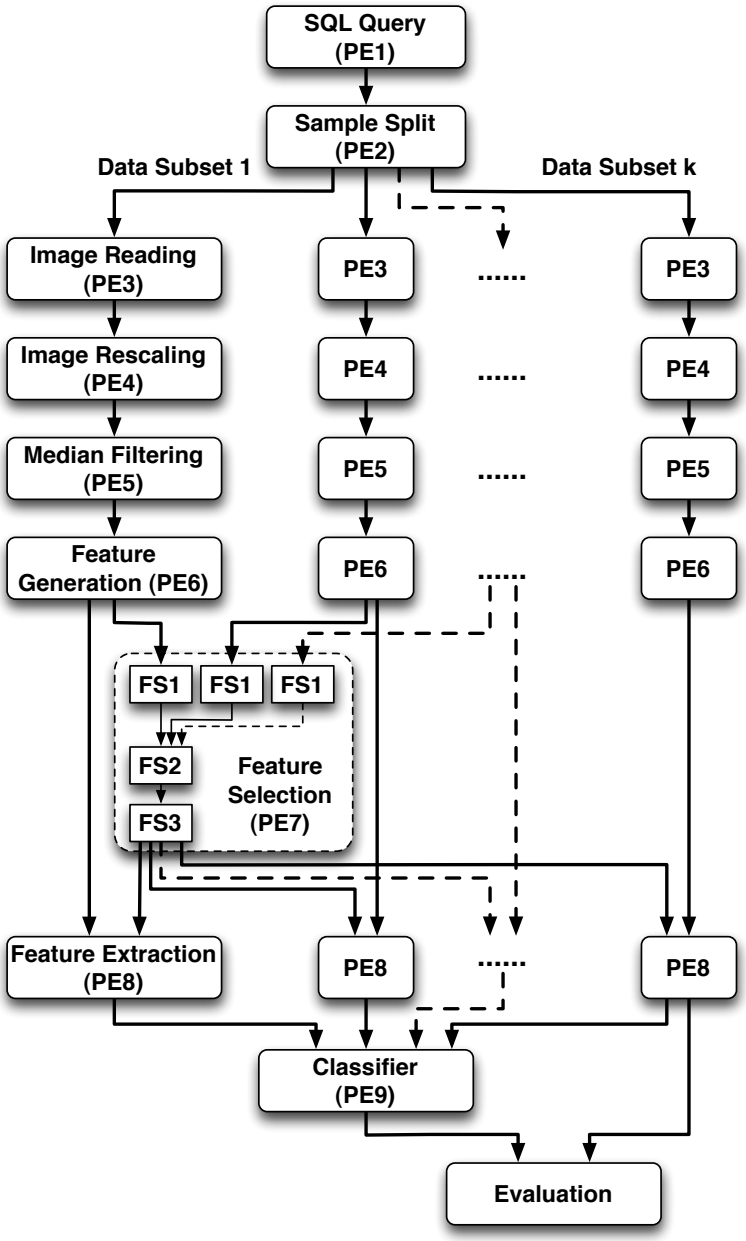


Figure 5.9: EURExpress-II workflow for parallel execution.

Data movement between the DIVMs is implemented using two OGSA-DAI activities, namely DeliverToTCPActivity (shown as S) and ObtainFromTCPActivity (shown as R). The DeliverToTCPActivity creates a TCP socket and sends data to another TCP host implemented in the ObtainFromTCPActivity. Data can be sent as a primitive data type, such as integer and double, or as resizable char array. These two activities are designed to allow fine-tuning on the transmission process by adjusting char array size and socket buffer size, and to support a large volume of data movement with multithreads.

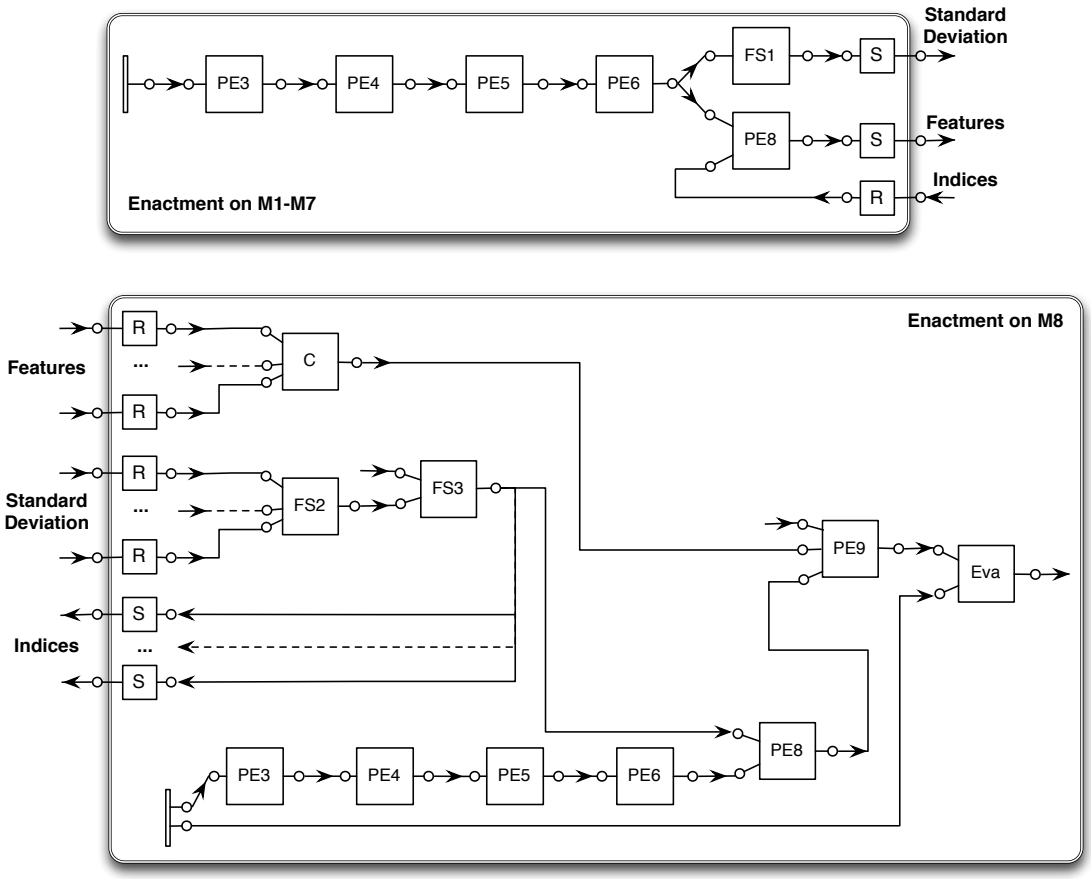


Figure 5.10: Execution plan for running on eight DIVMs.

These DIVMs are dedicated workstations connected on a LAN with other workstations. Prior to the execution, the raw data are distributed across all of the DIVMs, thus, PE_3 is read from the local file system. A MacBook running a submission client[5] is used to submit the workflows to all of the DIVMs. The result of the evaluation, i.e. an Integer array, is sent back to the submission client by M_8 at the end of the execution.

Experimental Procedure:

The experiment started with executing on a single workstation, where both training and testing workflows are sent to this workstation for execution. The workflow will training and evaluating a classifier with 800 images, and will be repeated for ten times to get the average performance. Then, the number of images is increased to 1600, then 3200 and up to 19200. The experiment is then move on to execution on multiple machines. The experimental procedure is shown below:

[5] A customised OGSA-DAI client that measured the workflow-level metric, i.e. the total makespan. It performs a time-stamping prior to submission, and another one immediately after receiving the last result item.

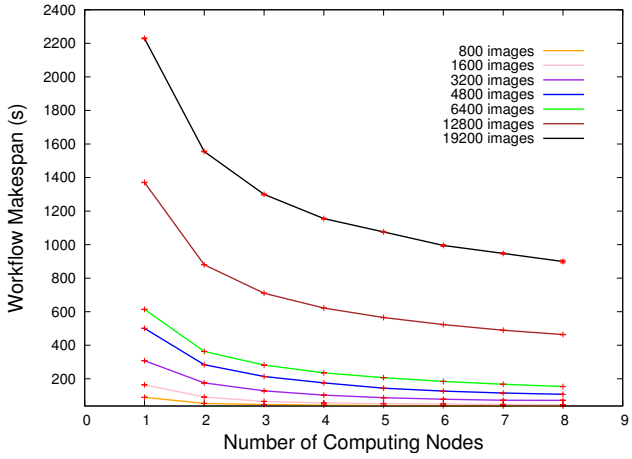
Set experiment parameter, $param = \{800, 1600, 3200, 4800, 6400, 12800, 19200\}$.
The experiment started with a single workstation.
for all image sizes defined in *param* **do**
  for *iteration* = 1 to 10 **do**
    Launch OGSA-DAI server on a workstation.
    Using a client machine, submit the EURExpress-II workflow.
    Store the performance data into PDB.
    Shutdown the OGSA-DAI servers and clear system cache.
  end for
end for

Proceed to enactment on multiple machines.
for *total* = 1 to 7 **do**
  for all image sizes defined in *param* **do**
    for *iteration* = 1 to 10 **do**
      Launch OGSA-DAI server on M_8.
      Launch OGSA-DAI servers on *total* machines.
      the classifier testing workflow to M_8.
      Submit the classifier training workflow to *total* machines.
      Store the performance data into PDB.
      Shutdown the OGSA-DAI servers and clear system cache.
    end for
  end for
end for

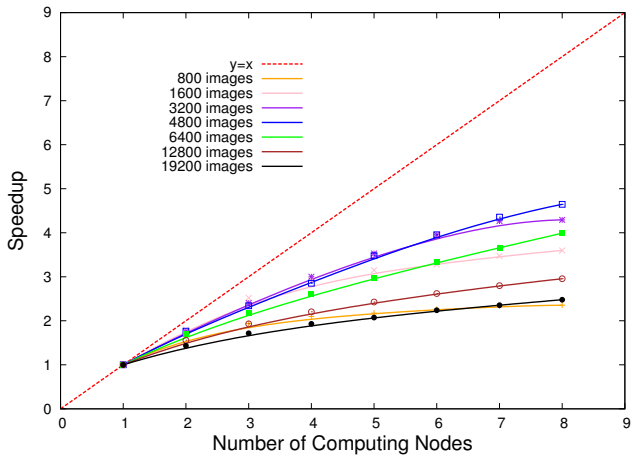
Results and observations:

The results of our first manual optimisation attempt are shown in Figure 5.11. We manage to achieve a small speed up on the parallel execution (The error bar shown in Figure 5.11(a) is so small that it appears as a red cross on the graph). To identify the reason of the poor speed up, we look into the PE-level fine-grained execution trace. The coloured histogram in Figure 5.12 shows the activity processing time of the PEs for 3 data samples (i.e. 6400, 12800 and 19200 images). The execution time for all of the PEs decreases gradually when more machines are added, except for PE_9. We then examined the implementation to discover the cause.

The problem is caused by the implementation of the testing workflow, which is executed on M_8. When more machines are added for each iteration, the training dataset



(a) Execution time with 99% confidence intervals



(b) Speedup of the parallel execution

Figure 5.11: Parallelisation of EURExpress-II workflow.

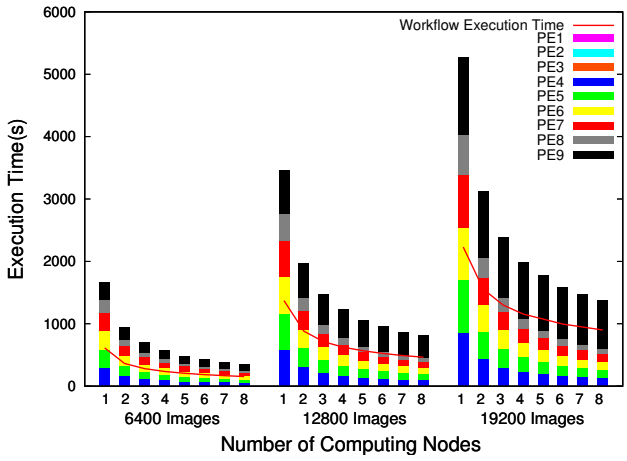
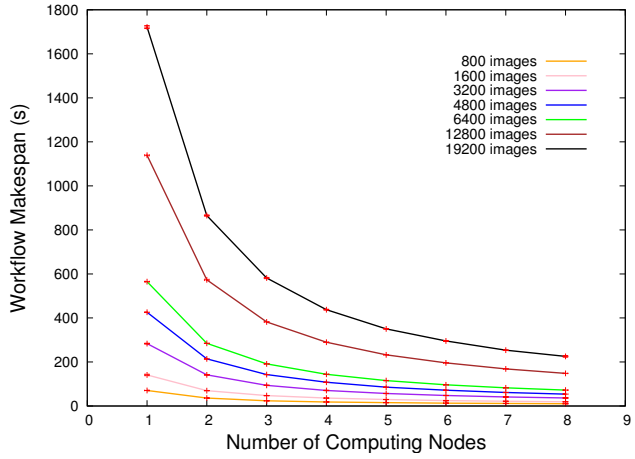
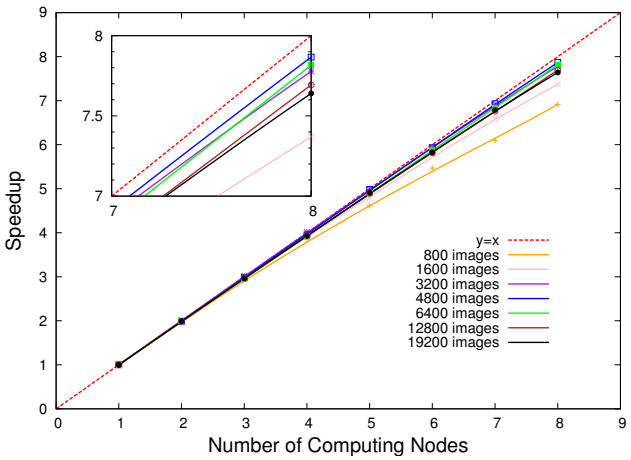


Figure 5.12: PEs execution time.



(a) Execution time with 99% confidence intervals



(b) Speedup of the parallel execution

Figure 5.13: Parallelisation of EURExpress-II workflow (second attempt).

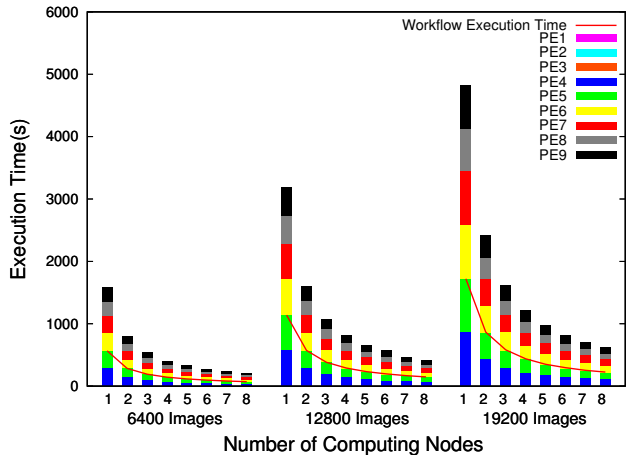


Figure 5.14: PEs execution time (second attempt).

gets smaller. However, the size of the test dataset remains the same in each iteration. PE_9 uses the testing dataset as the input for the classification process, thus, its execution time does not benefit from the parallelisation of the training pipeline. To further improve the execution, we have redesigned both workflows to achieve a balance workload—one of the optimisation approach to reduce overall makespan suggestion before (⇝*3.3*).

The result of the improvement is shown in Figure 5.13(a). The workflow makespan has been reduced with the increasing of number of computing nodes. Figure 5.13 shows near linear speedup with the number of computing nodes used. The total processing time of the PEs is more than the workflow makespan. For instance, the average total processing time of the PEs for executing 19200 images on a single machine is 4821 seconds, while the real workflow processing time (indicated as red line in Figure 5.14) is 1721 seconds. In the streaming model, a PE will output the result to the next PE in the workflow as soon as it finishes the computation on a portion of its data stream. The difference between 4821 seconds and 1721 seconds shows the advantage of the streaming execution model. In a pipeline of PEs, each consumes the results of the predecessor as soon as a granule of work is available. Thus they overlap except for the time taken by the predecessor to produce the first granule and for the successor to handle the final granule.

We examine the PE level metrics in Figure 5.14. As before, PE_1, PE_2 and PE_3 still show no significant impact on the overall execution. The processing of PE_9 is reduced proportional to the increase of execution machines.

5.3.3 Evaluation of three-stage mapping algorithm

We have conducted two experiments for evaluating our proposed three-stage mapping algorithm. The first experiment is designed to investigate the sensitivity of the threshold value in affecting the overall mapping result. The second experiment aims to compare our proposed algorithm with simple resource allocation algorithms.

5.3.3.1 Experiment 1

Hypothesis: Selection of threshold value has a big influence on the mapping result.

Use cases: EURExpress-II image preprocessing and feature selection workflow.

Evaluation method: We have reimplemented all of the PEs used in the EURExpress-II classifier training and testing workflow and written the DISPEL script for the enact-

ment on the ADMIRE gateway. In this experiment, we used the image preprocessing, feature generation and the feature selection process, which are the first five steps of the EURExpress-II data-mining workflow, as described in Section 5.1.1. The DISPEL script is shown below:

```
package eu.admire.EURExpress {
  // OGSADAI and ADMIRE libraries
  use uk.org.ogsadai.SQLQuery;
  use uk.org.ogsadai.Split;
  use uk.org.ogsadai.TupleSplit;
  use uk.org.ogsadai.ListRemove;
  use uk.org.ogsadai.ReadFromFile;
  use uk.org.ogsadai.DeliverToNull;
  use eu.admire.Results;

  // EURExpress PEs
  use eu.admire.EURExpress.ImageRescale;
  use eu.admire.EURExpress.BufferedImageToMatrix;
  use eu.admire.EURExpress.MedianFilter;
  use eu.admire.EURExpress.FeatureGeneration;
  use eu.admire.EURExpress.FisherRatioParallel;
  use eu.admire.EURExpress.StandardDeviationCalculation;
  use eu.admire.EURExpress.StandardDeviationCombine;
  use eu.admire.EURExpress.IntegerArrayToString;

  // DMI PE instances
  SQLQuery query = new SQLQuery;
  TupleSplit tsplit = new TupleSplit;
  ReadFromFile reader = new ReadFromFile;
  ImageRescale rescale = new ImageRescale;
  BufferedImageToMatrix converter = new BufferedImageToMatrix;
  MedianFilter filter = new MedianFilter;
  FeatureGeneration generator = new FeatureGeneration;
  StandardDeviationCalculation devCalc = new StandardDeviationCalculation;
  StandardDeviationCombine devComb = new StandardDeviationCombine;
  FisherRatioParallel ratio = new FisherRatioParallel;
  IntegerArrayToString resultToString = new IntegerArrayToString;
  Results results = new Results;

  // Utility PE instances
  ListRemove listRemoveInput = new ListRemove;
  ListRemove listRemoveClass = new ListRemove;

  // Declaring variables
  String anatomicalComponent  =
    "embryo_limb_forelimb_arm_upper_arm_mesenchyme_humerus";
  String expression =
    "select CONCAT('e', CAST(euxassay_ID as CHAR), '_1.jpg'), " +
    "case when " + anatomicalComponent +
    " < 5 then 0 else 1 end from annotation limit 100";
  String annotationDbResource = "DbAdmireResource";
  String imageFileResource = "FileAdmireResource";
  Integer width = 2048;
  Integer height = 3584;
  Integer dblevel = 2;
  Integer dbname = 3;
  Integer mask = 25;
  Integer featureNumber = 100;

```

```
55    |- expression -| => query.expression;
56    |- annotationDbResource -| => query.resource;
57    query.data => tsplit.data;
58
59    tsplit.result[0] => listRemoveInput.input;
60    listRemoveInput.output => reader.file;
61    |- imageFileResource -| => reader.resource;
62
63    reader.data => rescale.data;
64    |- repeat enough of width -| => rescale.width;
65    |- repeat enough of height -| => rescale.height;
66
67    rescale.output => converter.input;
68    converter.output => filter.input;
69    |- repeat enough of mask -| => filter.mask;
70
71    filter.output => generator.data;
72    |- repeat enough of dbname -| => generator.dbname;
73    |- repeat enough of dblevel -| => generator.dblevel;
74
75    generator.output =>  devCalc.data;
76    tsplit.result[1] => listRemoveClass.input;
77    listRemoveClass.output => devCalc.classType;
78
79    devCalc.result => devComb.data;
80    devComb.result =>  ratio.data;
81    |- featureNumber -| => ratio.featureNumber;
82
83    ratio.output => resultToString.input;
84    |- repeat enough of "," -| => resultToString.separator;
85
86    |- "Selected Feature Indices" -| => results.name;
87    resultToString.output => results.input;
88
89    submit results;
90  }
```

Lines 2 to 9 import the PEs from the OGSA-DAI and ADMIRE libraries, and lines 11 to 19 import the EURExpress application-specific PEs that we have developed. Lines 40 to 53 define the parameters used for the workflow, e.g. the chosen anatomical component, the raw image data source, the relational database that stores the annotation metadata, and the image processing parameters. The result produced by the `FisherRatioParallel` PE is an array of `Integer`. An `IntegerArrayToString` PE is used to convert it into `String`, which is written to a data source at the end of the execution.

We have set up an ADMIRE testbed on four DIVMs. These DIVMs are dedicated workstations, namely `eScience5`, `eScience6`, `eScience7`and `eScience8` that are connected on a Gigabit Ethernet LAN. Table 5.5 summarises the specification of these workstations. Each workstation hosts an execution engine and one of them, i.e. `eScience8`, hosts a gateway that performs the enactment. The measurement framework and PDB are deployed to collect the performance data. Figure 5.15 shows the layout of the testbed. The DISPEL request is enacted for 10 times on a single execution engine to populate the

Component	escience5/6/7	escience8
Processor	Intel Xeon Quad 2.2 GHz	Intel Core 2 Quad 3.0 GHz
Memory	16 GB DDR3	16 GB DDR3
Storage	4 × 2 TB 7200 RPM HDD	2 × 500 GB 7200 RPM HDD

Table 5.5: Specification of the workstations used for the experiments.

PDB. The raw images used for the experiment are in higher resolution compared with the images used in the previous experiments, i.e. average of 2790 × 3990 pixels with the file size around 600KB. The input data size should generated sufficient workload for evaluating the algorithms.

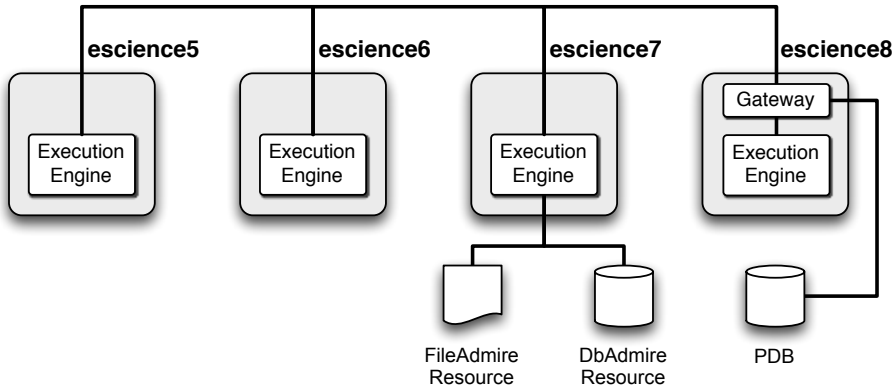


Figure 5.15: The testbed set up for the experiment.

Based on the performance data and our experience from previous experiment (⇝*5.3.2*), we manually parallelise the workflow into two processing pipelines. This is achieved by adding a `TupleArithmeticSample` PE after the `SQLQuery` PE to split the results obtained from the annotation database. The `StandardDeviationCombine` PE merges both pipelines prior to the Fisher Ratio calculation. The DISPEL request is processed by the gateway to produce a fully-expanded graph that consists of 35 PEIs (including multiple instances of a same PE). Figure 5.16 illustrates the DISPEL graph of the workflow that is annotated with performance data. The number shown in the blue boxes is the unit cost of the PEI measured from the initial enactment, while the number shown in the red ovals is the relative communication cost. These information are used for the mapping algorithm.

A major optimisation decision is the selection of the *threshold value*, which is used to partition the PEIs prior to the DIVM allocation (⇝*3.4.2*). Before we decide the threshold value, we first look at the performance data gathered in the PDB for this workflow on these DIVMs. Figure 5.17 shows the average unit cost of all of the PEs used in this workflow, sorted in descending order from the left to the right.

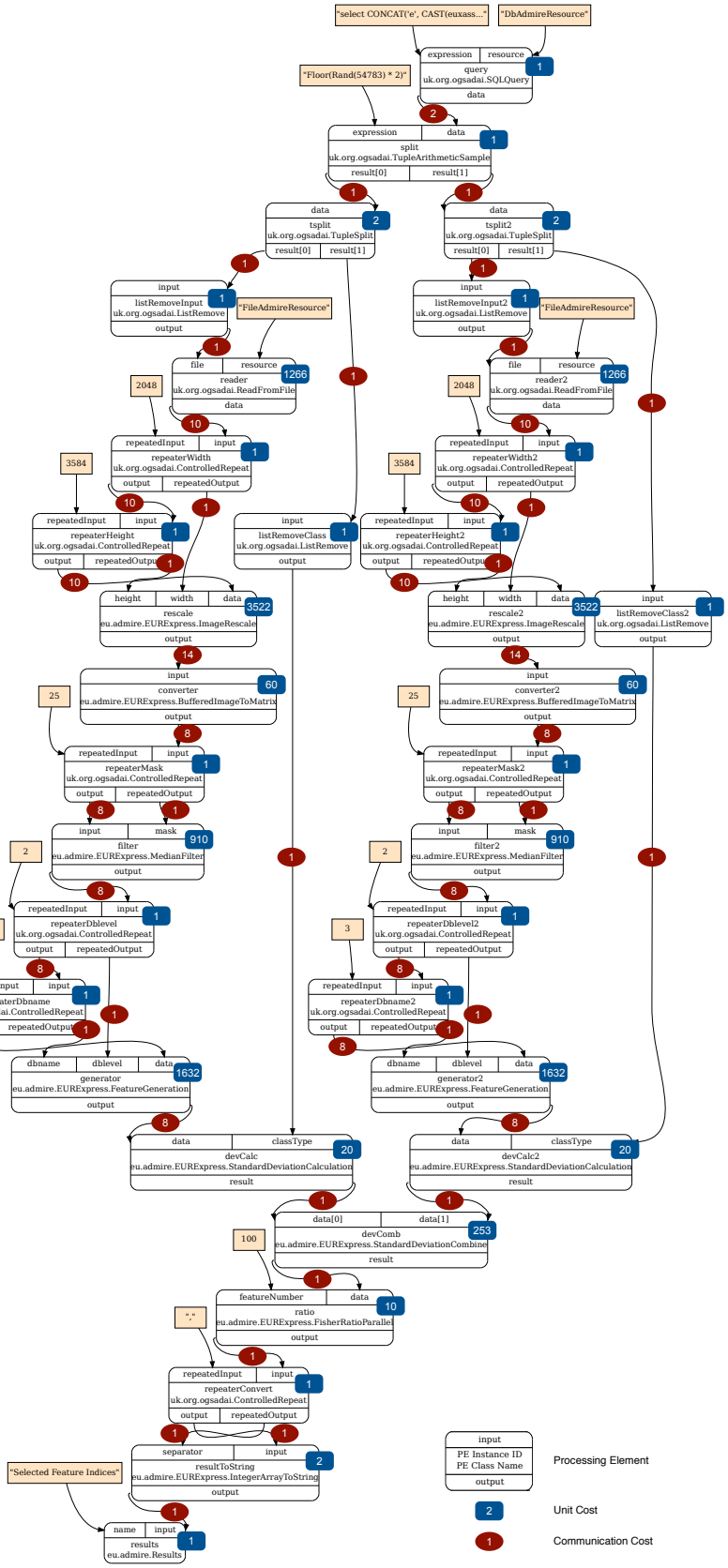


Figure 5.16: EURExpress-II workflow annotated with performance data.

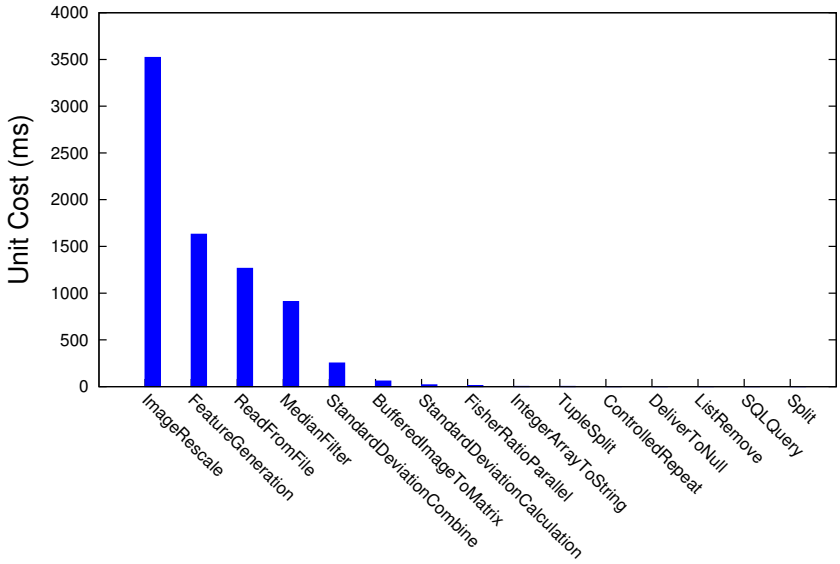


Figure 5.17: The unit cost for all of the PEs used in the workflow (listed in Table 5.6).

In general, there are few high unit cost PEs, where the unit cost is greater than 1000 milliseconds. Most of the PEs have very low unit cost, i.e. 1 or 2 milliseconds. Figure 5.18 shows the distribution of the PEs' unit cost, in a logarithmic scale. We have selected four threshold values for this experiment (in milliseconds): 0, 100, 1500 and 4000. The first threshold value will categorise all of the PEs in the heavy PEs, as the minimum unit cost among the PEs is 1 millisecond. The last threshold value, i.e. 4000 will place all of them in as light PEs, because the highest unit cost is recorded as 3522 milliseconds (`ImageRescale` PE).

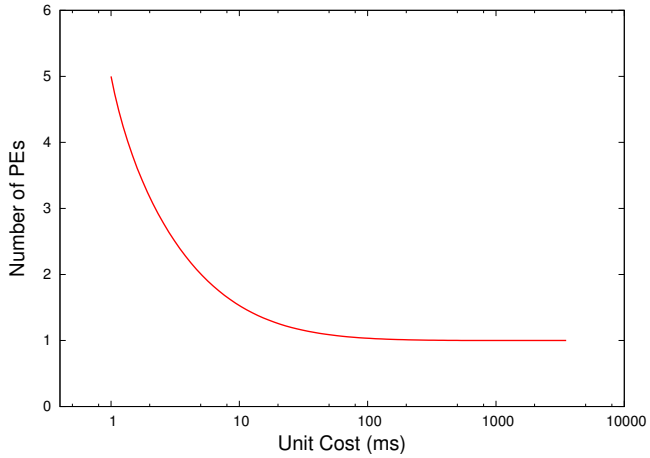


Figure 5.18: The distribution of PEs' unit cost.

The embedding of the optimiser plugin in the ADMIRE prototype is still ongoing. Thus, we have manually translated the output from the three-stage mapping algorithm to generate the concrete workflows. This is achieved by annotating the DISPEL request

PE	Unit Cost
ImageRescale	3522
FeatureGeneration	1631
ReadFromFile	1266
MedianFilter	910
StandardDeviationCombine	253
BufferedImageToMatrix	60
StandardDeviationCalculation	20
FisherRatioParallel	10
TupleSplit	2
IntegerArrayToString	2
SQLQuery	1
Split	1
DeliverToNull	1
ListRemove	1
ControlledRepeat	1

Table 5.6: Average unit cost of the PEs retrieved from the PDB.

with the mapping generated by the algorithm, using each of the threshold values defined above. We will evaluate the result of the threshold selection based on the overall workflow makespan for the execution on one, two, three and four execution engines. The concrete workflows are enacted by the gateway on the execution engines according to the mapping results. The data movement across execution engines is automatically handled by the enactor. The enactor will create a data source at the sender side that allow the PEIs to stream data into it. On the receiver side, the PEIs can read the data stream by sending a request to the data source[6].

Experimental Procedure:

The experiment started with executing on a single execution engine. As one execution engine is used for the mapping, the choice of threshold value would make no different (all of the PEIs will be mapped onto this execution engine). This execution serves as a benchmark for the comparison of the rest of the execution. The workflow selects 30 images, preprocesses and generates feature values for each of them. These feature values are used in the later stage for the feature selection process. Both filesystem resource (raw images) and relational database (annotation metadata) are deployed on `escience7`. The experiment is then move on to execution on multiple execution engines. Each execution has been repeated 10 times, and the average makespan is used for the evaluation. The experimental procedure is shown below:

[6]OGSA-DAI: `http://sourceforge.net/apps/trac/ogsa-dai/wiki/UserDocumentation`.

The experiment started with a single execution engine.
for *iteration* = 1 to 10 **do**
  Using a gateway client, submit the DISPEL request to the gateway.
  Store the performance data into PDB.
end for

Set experiment threshold value, *threshold* = {0, 100, 1500, 4000}.
Proceed to enactment on multiple execution engines.
for all threshold values defined in *threshold* **do**
  for *total* = 2 to 4 **do**
    Apply the three-stage mapping on the DISPEL graph
    Annotate the DISPEL request with the mapping result
    for *iteration* = 1 to 10 **do**
      Using a gateway client, submit the DISPEL request to the gateway.
      Store the performance data into PDB.
    end for
  end for
end for

Results and observations:

The algorithm first assigns the anchored PEIs (⇝*3.5.2*), i.e. `SQLQuery` and `ReadFromFile`. `SQLQuery` PEI retrieves data from a relational database resource hosted on `escience7` and both `ReadFromFile` PEIs reads files stored on the same execution engine. Thus, they are assigned to `escience7`. The algorithm then moves on to partitioning the PEIs into heavy or light category based on the selected threshold value.

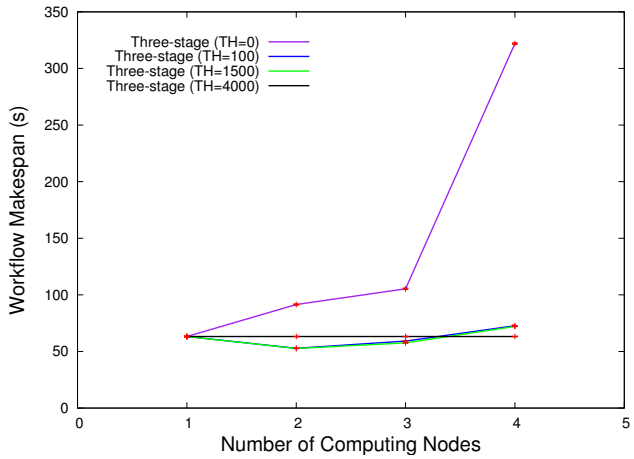


Figure 5.19: Workflow execution time with 99% confidence intervals.

Figure 5.19 shows the result of the experiment. Setting the threshold value to 0 milliseconds causing all of the PEIs are categorised as heavy PEIs and are assigned using the algorithm defined in Section 3.5.3. All of the PEIs are assigned based on their unit cost to achieve a balance computation workload among the execution engines, without considering their data communication cost. Thus, the makespan is increase exponentially when more execution engines are used. On the other hand, setting a very high unit cost, i.e. 4000 milliseconds, causes the algorithm treating all of the PEIs as light PEIs. In this workflow, all of the anchored PEIs are assigned to `escience7`. The algorithm for assigning light PEIs (⇝*3.5.4*) attempts to minimise the communication cost and assign all of the PEIs on `escience7`. As a consequence of the assignment, the algorithm only used one execution engine all the time. This is showed as a horizontal line on the graph.

Threshold	**Heavy PEI**	**Light PEI**
0	32	0
100	7	25
1500	4	28
4000	0	32

Table 5.7: Number of light and heavy PEIs for each threshold value.

The choice of using 100 milliseconds and 1500 milliseconds as the threshold value shows a very close performance difference. With 1500 milliseconds threshold value, only 4 PEIs are categorised as heavy PEIs, i.e. `ImageRescale` PEIs and `FeatureGeneration` PEIs on both pipelines. as a result, these very high unit cost PEIs are separated from the remaining PEIs, and their computation workloads are distributed among the two execution engines. The bead-sliding algorithm for assigning the light PEIs manage to minimise the communication cost by finding the weakest links in the pipeline to perform the cut. Thus, this assignment yields a better performance. However, when four execution engine are used, the makespan is longer than the execution on a single machine. This shows that the communication costs occur has compensated the performance yields from the workloads distribution. Similar result is observed for threshold value 1500 milliseconds.

We use the results to plot another graph to show the influence of the threshold values over the number of execution engines, shown as Figure 5.20. For the execution on a single machine, the makespan is not affected by the threshold value. As for the execution on multiple machines, the selection of threshold values determines the partitioning of the PEIs, and directly affects the overall workflow makespan. In this particular workflow, the ideal threshold value falls between 1500 milliseconds to 3500 milliseconds.

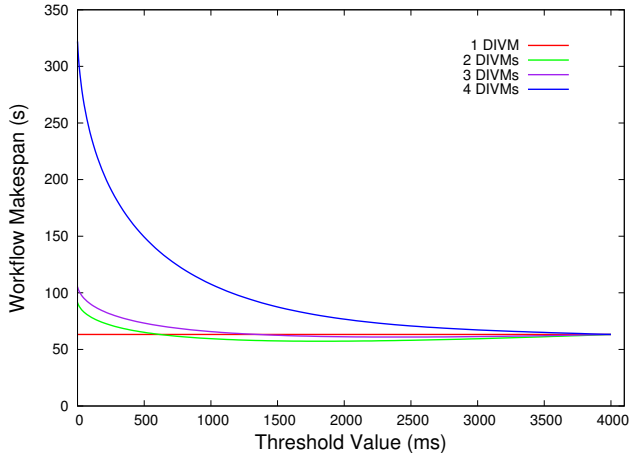


Figure 5.20: The influence of threshold values over the number of execution engines.

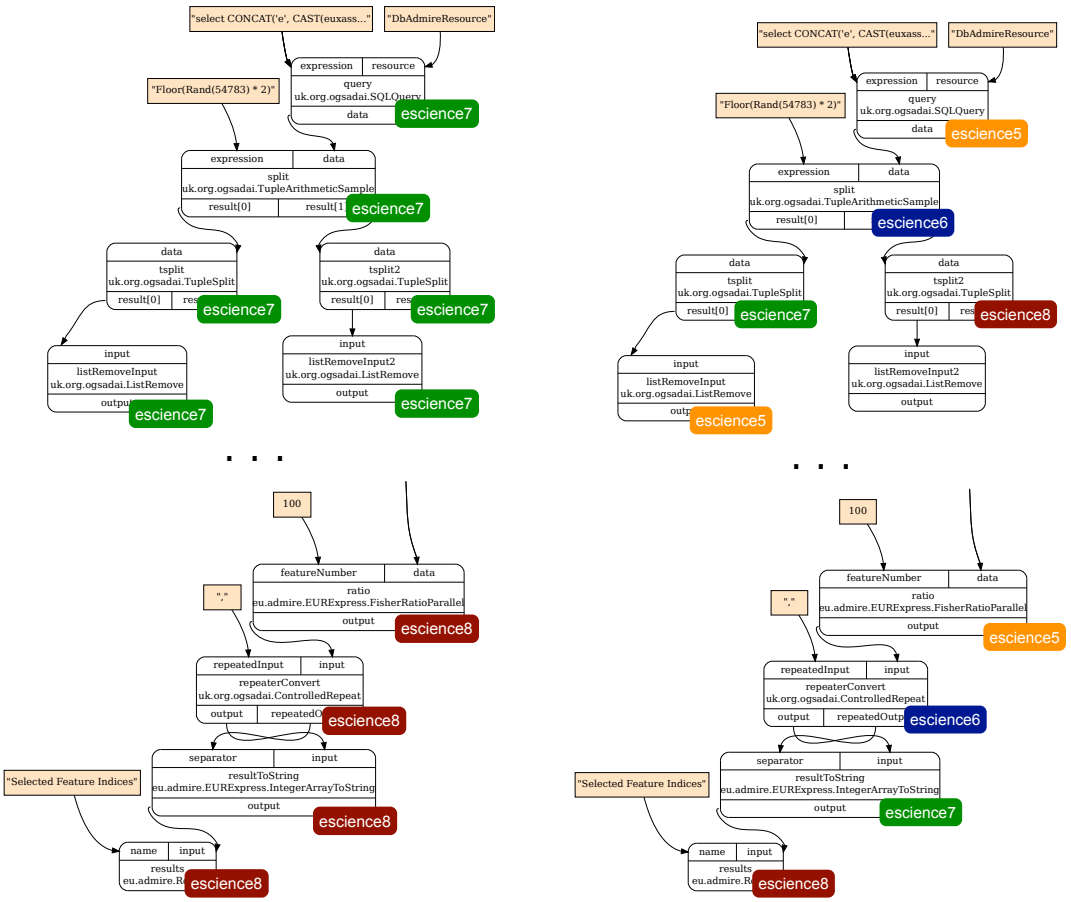
However, we would not generalise this observation as this threshold value may not be applicable to other workflows, as discussed in Section 3.4.2. At the same time, this experiment has shown that the performance threshold value can be determined from the performance data, as we have suggested in the earlier section. And in this case, there is not a significant difference in makespan when the threshold moves between two plausible values. In the future, when we experiment with a representative set of workflows, we will explore basing the performance threshold on a proportional split, e.g. 5% heavy 95% light, 10% heavy 90% light and 20% heavy 80% light. It is also necessary to choose a threshold which has more heavy PEIs than the planned number of DIVMs.

5.3.3.2 Experiment 2

Hypothesis: Our three-stage mapping algorithm performs better than simple allocation algorithms.

Use cases: EURExpress-II image preprocessing and feature selection workflow.

Evaluation method: We used the same workflow and setup as for Experiment 1, but added new mapping algorithms for the evaluation. We wanted to compare our three-stage mapping algorithm with two simple allocation algorithms: linear allocation and round-robin assignment. In the linear allocation method, the optimiser will deploy anchored PEIs before and distribute the total number of remaining PEIs equally among the available execution engines, i.e. this is based on a simple count. An alternative summing the unit costs and partitioning those linearly could be investigated in future experiments. The optimiser will traverse the DISPEL graph using breadth-first



(a) Linear Allocation (b) Round-robin Assignment

Figure 5.21: Simple allocation algorithms for comparison.

search strategy, and fill up the execution engine with every PEIs that it encountered. When the execution engines is allocated with the desired number of PEIs, the optimiser moves on to the next execution engine until all of the PEIs are assigned, as illustrated in Figure 5.21(a). For the Round-robin allocation algorithm, the optimiser also traverses the graph using breadth-first search strategy, but it assigns the PEIs to the execution engines in a Round-robin fashion, as illustrated in Figure 5.21(b). The workflow makespan resulted from the mapping of these algorithms are used to compare with the results from Experiment 1.

Experimental Procedure:

We have measured the performance of executing on a single execution engine in Experiment 1. Thus, this experiment only involves the enactment on multiple execution engines. The experimental procedure is shown below:

Set candidate algorithms, $algorithm = \{LinearAllocation, RoundRobin\}$.
for all candidate algorithms defined in *algorithm* **do**
  for $total = 2$ to 4 **do**
    Apply the *algorithm* mapping on the DISPEL graph
    Annotate the DISPEL request with the mapping result
    for $iteration = 1$ to 10 **do**
      Using a gateway client, submit the DISPEL request to the gateway.
      Store the performance data into PDB.
    end for
  end for
end for

Results and observations:

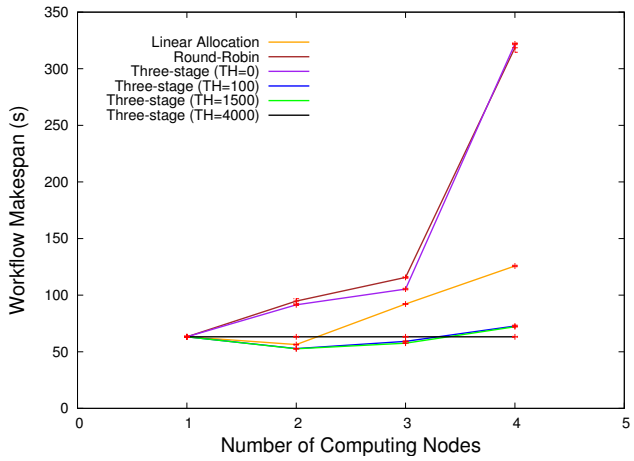


Figure 5.22: Comparison of different algorithm running on distributed execution.

The execution results are combined with the data from Figure 5.19 and shown as Figure 5.22. The Linear Allocation shows a good performance for the execution on 2 execution engines as the algorithm assigned the first half of the graph that comprises both `ImageRescale` PEIs on `escience7` and the remaining PEIs (including `FeatureGeneration`) on `escience6`. However, the performance is getting worse with more execution engines. There are two factors that contribute to the increase of workflow execution time. First, the workloads are not balance among the execution engines. All of the heavy PEIs are assigned to `escience7` and `escience8`, leaving `escience5` and `escience6` idle most of the time. Second, most of the data streams with high data volume are spread across machines, i.e. between `ReadFromFile` (on `escience7`) and `ImageRescale` (on `escience6`), and between `MedianFilter` (on `escience5`) and `FeatureGeneration` (on `escience8`). This has incurred additional cost to serialise and de-serialise the data for the network transfer.

The allocation using Round-Robin fashion is the worst. Even though the time of overlapping execution is the highest among the candidate algorithms, but it pays a very high data movement cost. Every output is written to a data source while every input is read from a separate execution machine[7]. The results from this experiment show that with a suitable performance threshold, the three-stage mapping algorithm performs better than the simple allocation. However, these are clearly preliminary and tentative conclusions. To confirm them it is necessary to investigate other competitors, such as depth-first packing and to measure a number of workflows in a number of contexts.

There are two important issues arose from the two experiments: serialisation and stream buffer implementation. To support the enactment across distributed execution engines, each of the output data types should be serialisable. For instance, the output of `ImageRescale` is a Java `BufferedImage` object that is not serialisable. We have to implement our own Java class to serialise and de-serialise it for the network transfer. Furthermore, some objects require specific serialiser to achieve a better performance, e.g. a two dimensional array of `Integer`. We can treat it as an `int[][]` object or a list of `int[]` that can be streamed across the network and reassembled on the receiver side. Both methods would result a different performance.

The second issue is regarding the size of stream buffer of the execution platform, i.e. OGSA-DAI. A smaller buffer size may cause deadlock in the streaming execution, but a bigger buffer size may easily fill up the memory space. Each output produced by the `BufferedImageToMatrix` PE is a matrix with the size of $2048 \times 3584 \times 4$ bytes = 29MB. In a Java Virtual Machine with 16GB memory, the stream buffer can hold up to 550 data items (in the experiment, we have used 12GB memory). This limitation has restricted the use of large-scale data in the experiments. We believe that scalability can be achieved by implementing a dynamic stream buffer, that will: *a*) dynamically adjust the buffer size according the execution demand and the memory capacity, and *b*) automatically spill over to hard disk drive if the execution has used up all of the memory space. Clearly, the quality of the inter-machine communication affects the tradeoff between computation and data transmission. This in turn may affect the impact of optimiser choice.

It is pleasing to note, that at least for this one workflow, our hypothesis that some PEs have dominant computation cost is supported by the measured distribution of costs shown in Figure 5.17 and Figure 5.18. A longer study is needed.

[7] We are not suggesting Round Robin is a sensible candidate mapping as it clearly pessimises data movement costs.

5.4 Experimental plan for evaluating mapping algorithm

We have demonstrated the use of performance data in the DIVM allocation with a candidate mapping algorithm and evaluated the algorithm with a modest workflow in the previous section. To further examine the efficiency of the three-stage mapping algorithm and validate the stability of the performance data, the evaluation of the mapping algorithm requires real workloads from different use cases to be enacted on the data-intensive architecture prototype. However, this requires considerable elapsed time to design the workflows with users, and then run the experiments. We offer a preliminary design of an experimental plan for a more through evaluation.

Use cases: The workflows used for the experiments should: *a*) involve large data size (gigabyte to terabyte), *b*) have multiple data sources with anchored potential, *c*) be sufficiently complex graphs that consist of PEs with different unit costs and input/output streams, and *d*) not require human intervention (e.g. prompt for user input) in the processing pipeline. We have identified the use cases for this experiment, as follows:

1. EURExpress-II.

We use the classifier training and testing workflows and represent the test workloads as DISPEL scripts. Altogether three DISPEL requests were created: data preparation, feature selection, and k-fold cross validation (training and testing a classifier). Figure 5.23 shows these workflows and the relevant resource infrastructures.

The raw mouse embryo images include both natural and technical variations, and each image is expected to be annotated with multiple anatomical terms (the same image will be used multiple times for training different classifiers, one per anatomical term). Thus, the first DISPEL request is for *data preparation*, which comprises the first three subtasks described in Section 5.1.1: image integration, image preprocessing and feature generation. The generated features are stored in a file repository, while the metadata are stored in a features database. Images can be preprocessed independently, thus allowing parallel execution; the dataset is divided into multiple subsets based on the number of available execution engines.

At this point, all of the images are preprocessed and features are generated. The second DISPEL request, *feature selection*, is submitted to select the most significant subset of the features for constructing a particular anatomical classifier, defined by one of the parameters. We have two implementations for the `FisherRatio` PE: sequential and

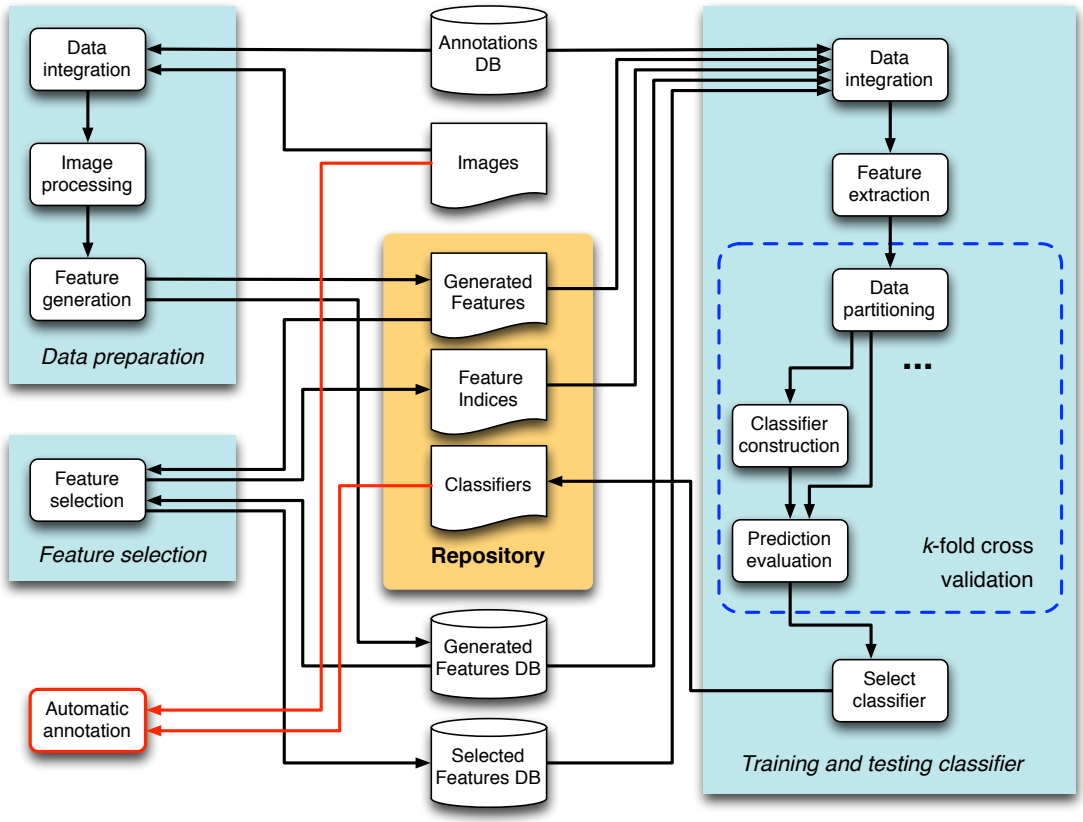


Figure 5.23: EURExpress-II annotation workflow.

parallel. The parallel implementation splits the computation into two parts. The first part computes the median and standard deviation calculation on a given subset of the dataset (retrieved from the repository) using a `StandardDeviationCalculation` PE on each of the execution engines. The second part then combines all of these calculations using a `StandardDeviationCombine` PE, and completes the Fisher Ratio computation using `FisherRatioParallel` PE. The result is returned to the repository.

The final DISPEL request extracts the features, and trains and tests classifiers using an off-the-shelf k-fold cross-validation function. To use this function, we developed three application specific PEs: `TrainClassifier`, `Classify`, `Eval` from the package `book.examples.eurexpress`.

2. Seismic ambient noise processing.

Our experiments focus on the first two phases of the seismic ambient noise processing, which involve the integration of large-scale seismic data from distributed seismic archives, and correlation processes. The high-level view of the main processes in these phases is shown in Figure 5.24.

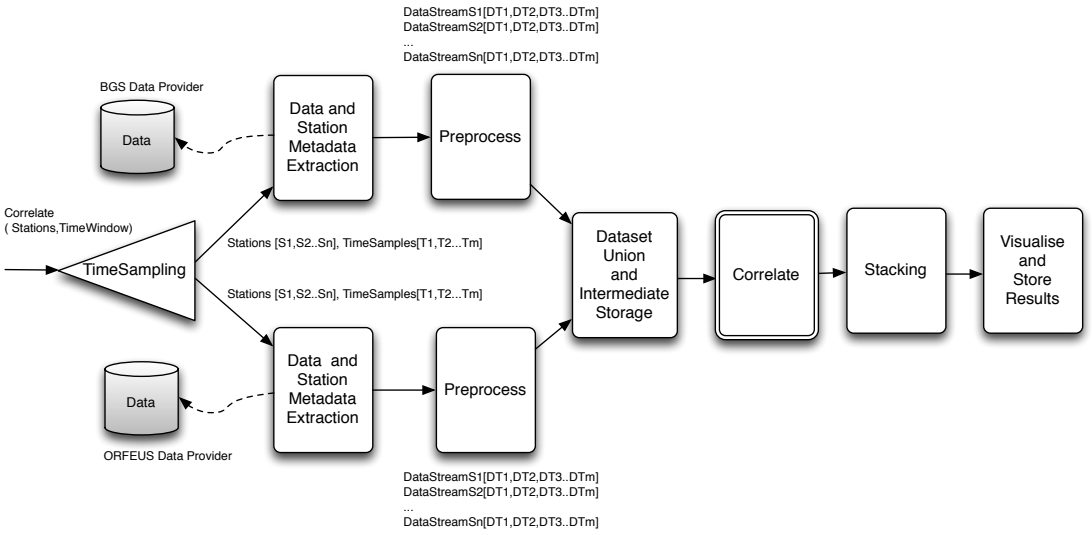


Figure 5.24: Ambient-noise workflow using two seismic archives from Galea *et al.* [66].

The first step is to identify the seismic stations and the time window that segments the time samples, and pass it to the extractor PEs. The extraction is done in parallel from the archives storing historical data for a region's network of seismic stations. Each archive may use different storage mechanisms and protocols. The extracted data are sent through a series of preprocessing stages and are shaped into the input streams that are suitable for further processing. The data streams from all of the seismic stations are then merged by a `StreamHarvester` PE, that performs a selection on the best samples (to eliminate duplicates) for the pair-wise cross-correlation. The intermediate results of the cross-correlation are then stacked to assemble the overall duration of interest, and finally stored in persistent storage.

All of the DISPEL requests have been developed and tested on the ADMIRE testbed.

Evaluation Method:

The evaluation should comprise four experiments, with both use cases on EDIM1 and ECDF cluster respectively: *1*) EURExpress-II on EDIM1, *2*) Seismic ambient noise processing on EDIM1, *3*) EURExpress-II on ECDF cluster, and *4*) Seismic ambient noise processing on ECDF cluster. EDIM1 is under our direct control which allows dedicated experiments isolated from unpredictable competing workloads. It is also designed to be well suited for this class of work. The ECDF cluster is a typical type of computing resource accessible by scientific users. The results in such a context will be informative for many typical application contexts. By using two real use cases on a variety of platforms, it will be possible to test the relevance and effectiveness of a range of optimisation methods based on performance data.

For each experiment, we try to evaluate the overall performance of the three-stage mapping algorithm, and each step individually. We will compare the performance of mapping all PEIs of the workflow on:

1. placing all of the PEIs on a single DIVM,
2. linear allocation of all of the PEIs across n DIVMs,
3. random distribution of all of the PEIs across n DIVMs,
4. three-stage mapping of all of the PEIs across n DIVMs,
5. linear allocation of all of the PEIs across n DIVMs with consideration of anchored and heavy PEIs,
6. linear allocation of all of the PEIs across n DIVMs with consideration of anchored and light PEIs, and
7. hand-crafted mapping of all of the PEIs across n DIVMs by data-intensive engineers.

The first setup works as a normalisation benchmark for the workflows. Setups 2, 3 and 4 are for comparison of the proposed algorithm with simple allocations. For the linear allocation, we first calculated the number of PEIs to be enacted on a DIVM, i.e. numbers of PEIs divided by numbers of DIVM. Then, we travel the graph to assign the PEIs to the first DIVM, and move on to the next DIVM once we have filled up the first one, without considering their unit cost and data volume of the streams. Setups 5 and 6 are conducted to investigate the mapping effect that only considers either the computational cost or the communication cost in deciding the allocations. The last setup is to find out how these automatic optimisations perform against manual optimisation.

The gateway and the PDB are installed on the data-staging node on EDIM1. The data nodes will host a single execution engine each, which is connected to the gateway. We use two different methods to load the data from the data-staging node to each of the data nodes. For the EURExpress-II workflow, the annotations DB and raw images are kept in the data-staging node. The data preparation workflow will read the raw data from the data-staging node, and distribute the data to data nodes for execution. The results, i.e. generated features, are stored locally on each data nodes. This is followed by the feature selection workflow to determine the most significant features for training classifier. The classifier training and testing workflow will be used as the test workload for the experiments above. As for the seismic ambient noise processing use case, the data are meant to be pre-distributed over the data nodes, that represent the seismic stations. This is done with a loading DISPEL request that moves the data from the data-staging node to the data nodes, prior to the experiments.

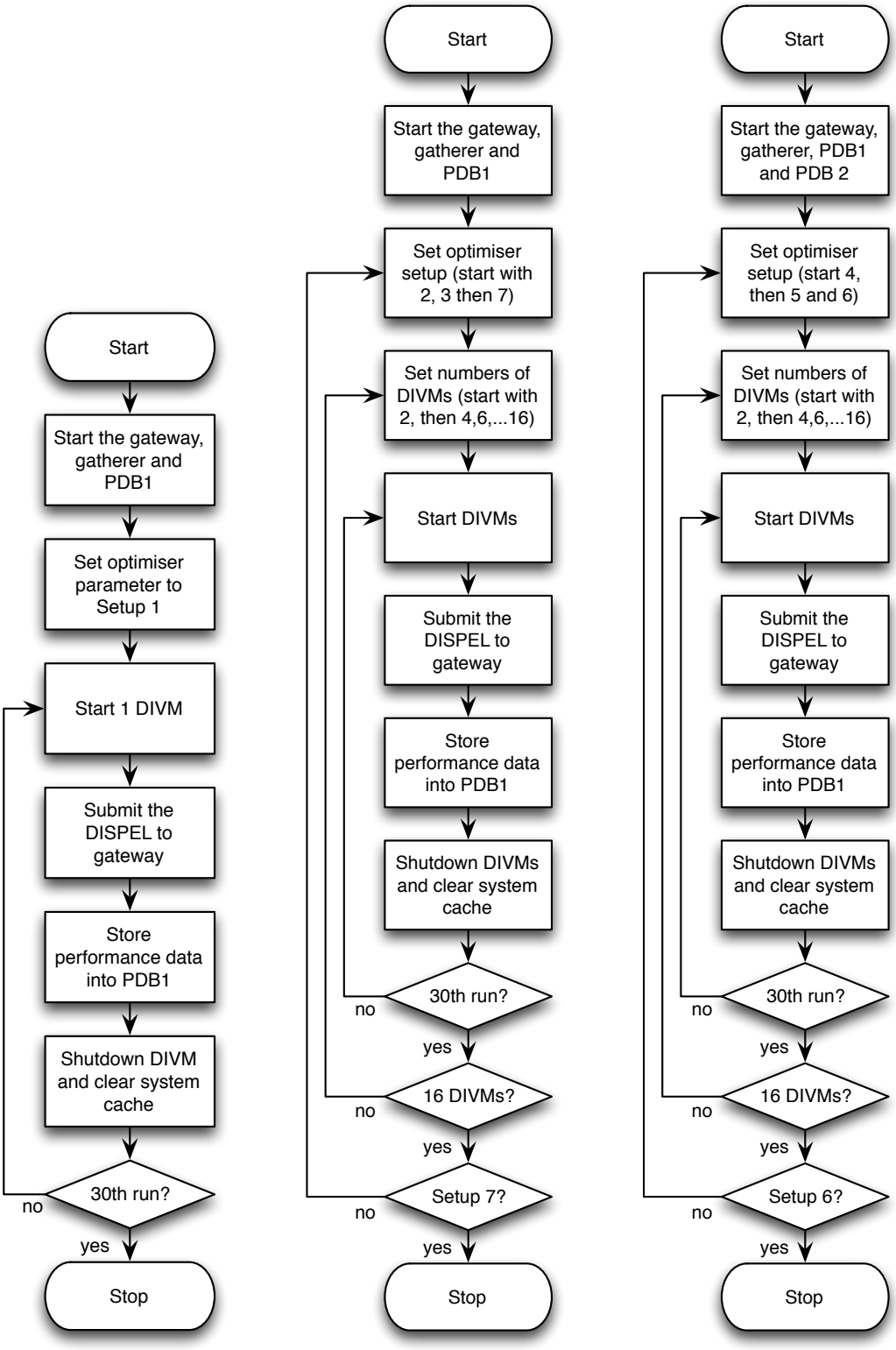
The third and fourth experiments are designed to compare the performance of the three-stage mapping algorithm on two different hardware infrastructures. The main obstacle is to run the streaming processing model on a Linux cluster that does batch-job processing. One of the possible solutions is using the pilot-job mechanism, such as Condor Glidein [149] and SAGA BigJob [124]. We will submit tasks to deploy execution engines on the cluster nodes. These long-running execution engines will be connected with a gateway (another pilot job), that accepts and enacts DISPEL requests. We will conduct further a feasibility study for these two experiments.

Experimental Procedure:

Figure 5.25 shows the experimental procedure for all of the seven setups, that will be tested in all of the four phases. The experiment starts with the first setup, i.e. single DIVM. An optimiser plug-in that allocates all of the PEIs on a single DIVMs is implemented and deployed on the gateway. The experiment procedure is illustrated in Figure 5.25(a) and is repeated thirty times—i.e. means and errors will be calculated from 30 similar runs. The performance data captured by the measurement framework are sent to a gatherer, which are then processed and populated into PDB_1. Figure 5.25(b) shows the procedure for the experiments with the second, the third and the seventh setup. Instead of running ten times on a single DIVM, the optimiser uses two DIVMs, and increases the number of DIVMs by 2 in each cycle, to the maximum of sixteen. For the seventh setup, data-intensive engineers will manually determine the allocation and hard-code it in the optimiser plug-in.

For each DISPEL request, we calculate the performance for each experiment setup by averaging all of ten enactments. Each run will eventually improve the optimiser in understanding the performance and behaviour of the PEs. Thus, for the fourth, fifth and sixth setup, the optimiser uses the secondary PDB[8], i.e. PDB_2, which is the same for each iteration in this experiment, and inserts the performance data gathered from these enactments into PDB_1 for future analysis. One of the worthwhile analysis would be comparing the efficiency of the mapping algorithm using the primary and second PDB respectively. This should help the investigation on the size of the PDB in affecting the performance of the mapping algorithm. Another important remark is the experiments with first, second, third and seventh setups do not use the PDB in making the mapping decision. Thus, this give us the opportunity to build up the initial PDB that comprises the performance data for all of the PEs.

[8]The secondary PDB is populated with performance data stored in the primary PDB prior to these experiments.



(a) Setup 1 (b) Setup 2, 3 and 7 (c) Setup 4, 5 and 6

Figure 5.25: Experimental procedure for all of the setup.

There are 4 DISPEL requests (i.e. 3 from EURExpress-II and 1 from seismic ambient noise processing) to be executed on 7 mapping setups with different numbers of DIVMs. Each enactment will be repeated 10 times for the means and errors calculation. The experiments will be conducted on both EDIM1 and ECDF cluster, i.e. 2 hardware setups. Table 5.8 summarises the number of executions involved. For the first setup, 1 execution engine is used for each of the enactment. So the total numbers of executions for the first setup will be $1 \times 30 \times 4 \times 2 = 80$ executions. As for the remaining setups, more executions are involved per run, e.g. the enactment on 16 DIVMs requires 16 executions. Thus, the whole evaluation is expecting to have 103,920 executions.

Setup	**PDB used**	**PDB pop.**	**# DIVMs**	**# rep.**	**# DISPEL request**	**# h/w**	**Total exe.**
1	-	PDB_1	1	10	4	2	80
2	-	PDB_1	2,4,6,...,16	10	4	2	5760
3	-	PDB_1	2,4,6,...,16	10	4	2	5760
4	PDB_2	PDB_1	2,4,6,...,16	10	4	2	5760
5	PDB_2	PDB_1	2,4,6,...,16	10	4	2	5760
6	PDB_2	PDB_1	2,4,6,...,16	10	4	2	5760
7	-	PDB_1	2,4,6,...,16	10	4	2	5760

Table 5.8: Summary of the experiments for evaluating mapping algorithm.

We expect to collect both workflow-level and PE-level performance metrics (⇝*4.3*) from these experiments. We would like to observe the workflow makespan and the efficiency of the algorithm, similar to Figure 5.11 with six lines that represent each of the setups. The fine-grained PE-level comparison, similar to Figure 5.12, would be useful for the study of their execution time and communication cost, influenced by the mapping strategies. The last thing we would like to compare is the overhead of the mapping algorithms.

The experimental plan could be used to explore other interesting research questions, such as: *a*) to study the parameters for the DISPEL requests, e.g. the number of the n-fold cross validation as affected by the number of DIVMs, and *b*) to explore the use of hybrid DIVMs, e.g. by mapping the data preprocessing and filtering PEs onto EDIM1 nodes that host the seismic station archives and mapping the cross-correlation and stacking PEs onto ECDF cluster.

5.5 Summary and evaluation

A major advantage of being involved in the ADMIRE project is the exposure to real-world scientific use cases, and their associated data, workloads and user behaviour, in different domains, e.g. astronomy, cosmology, developmental biology, hydrology and seismology, that can be used to evaluate our architecture and model. We have chosen two of them as the test workload for our experiments. We do not claim that these workflows cover the whole spectrum of scientific experiments, but we believe they are representative of the fundamental characteristics that are commonly found across the domains.

The hardware chosen for the experiments also shows the diversity of the computational resources. The workstations are the modest resource commonly accessible by the scientific community. The tightly-coupled, high-performance cluster is preferred by the users running compute-intensive workflows. Following the rise of data-intensive science, the EDIM1 that complies to the "*data bricks*" architecture, is believed to be a scale-out solution and is a prototype of the architectures that will handle the data deluge of next generation of scientific discovery.

We have conducted all of the phases of our experiments: streaming processing model and measurement framework (⇝*5.3.1*), optimisation of workflow enactment with parallelism (⇝*5.3.2*) and the evaluation of our three-stage mapping algorithm (⇝*5.3.3*). In the first phase, we have demonstrated that our proposed measurement framework is capable of obtaining fine-grained performance data from the enactment of data streaming work. We have shown how these data can be used to support the mapping algorithm. The initial prototype of the measurement framework and the PDB were implemented and published [116]. Improvement is still in progress, and will be discussed later in the future work Section 6.2.

In the second phase, we have explored the potential of parallelism in optimising data streaming workflows. Even though we have clarified since the beginning of this thesis that graph transformation is not our main focus, we believe that this manual-optimisation experiment can help to strengthen our understanding in designing the architecture and model. These experiments have been reported in [89, 91, 117], and the preliminary evidence from these supported the hypothesis that data-intensive research can be made scalable by using data-streaming workflows and the their enactments can be made efficient by exploiting performance data gathered during previous enactments.

In the third phase, we have investigates the sensitivity of the performance threshold in affecting the mapping results. We have shown that a proper selection of the threshold value would yield a good execution performance, and we also have shown that this threshold value can be determined from the PDB. We have conduct another experiment to compare our proposed three-stage mapping algorithm with another two simple allocation algorithms. These experiments are conducted with real-world scientific workflows with modest data and complexity. However, though the show that the categorisation of PEs into heavy and light computations and three-stage mapping algorithm is promising, they must be treated with caution until more complete experiments are performed.

We suggest that an extensive evaluation should be conducted to test the optimisation model and architecture. We have designed an experimental plan for the full evaluation of the architecture. Section 5.4 describes the chosen use cases that will be used for the test workloads for the evaluation. The evaluation method and the experimental procedure are presented, and we have discussed in detail how to setup and run the experiments. When these experiments are conducted, it is inevitable that they will lead to refinements in the optimisation model and algorithms. However, we believe that there is ample evidence to support the hypothesis that the three-stage mapping strategy will be a good foundation for these further development.

The next chapter concludes our research and highlights the future research opportunities that are enabled by this thesis.

CHAPTER 6

Conclusions and future work

This chapter concludes our work by summarising what we have done and emphasising the impact and contribution of our research in Section 6.1. It then suggests possible future research directions, in Section 6.2. Our final remarks are presented in the last section of this thesis, Section 6.3.

6.1 Achievements

The rapidly growing wealth of data is shaping the landscape of scientific discovery. The *data deluge* or *data bonanza*, *data tsunami* and *data explosion* coined by different researchers across domains has posed a new challenge in enacting scientific workflows, alongside the heterogeneity and complexity of the applications and execution environments. This thesis does not claim to be the ultimate survivor guide for the emergence of data-intensive science, but it reports our understanding and approaches to deal with these challenges.

We have discussed these challenges in Chapter 2. The heterogeneity and complexity cannot be solved by unification of technologies due to the existing investments of the communities, the socio-economy power of identity and the independent evolution of technology. Instead of finding a silver bullet to solve the diversity challenge, we should let a hundred flowers blossom, and invest in integration technologies.

We have conducted a thorough survey on the related work. We have reviewed the workflow life-cycle and identified the characteristics of workflow management that have not been discussed in existing taxonomies or reviews, and used this conceptual framework to compare five prevalent systems, i.e. Pegasus, Swift, Kepler, Taverna, and Meandre.

Based on our observation, we conclude that: *a*) abstraction is the key to support the diversity of data, applications, and execution environments, *b*) a workflow language that allows the separation of concerns is needed, *c*) the optimisation and enactment of scientific workflows should be automated, *d*) learning from previous enactments is important to improve future runs, and *e*) stream processing methods have the potential to execute data-intensive workflows efficiently.

We postulate the heterogeneity and complexity challenges in running scientific experiments can be solved by an architecture that separates the concerns between the process development and its enactment. A new data-intensive architecture originated from the ADMIRE project, that we were instrumental in its design, as is presented in Chapter 4. The data-intensive architecture supports the creation of diversified applications, that involve a broad variety of users and data across domains, and their enactment on diversified execution environments, with a standard interface, i.e. DISPEL. The data-streaming model central to this architecture provides a scalable approach to large-scale data exploration and exploitation. The well-designed data-intensive platform supports the exploratory nature of scientific experiments that require fast prototyping, reusable workflows and components, and easy to reproduce data-driven experiments. These experiments typically involve extensive data handling as well as analysis and presentation.

We argue that the enactment of scientific workflows can be made efficient using optimisation techniques in the data-intensive architecture, and above all, that the optimisation should be automated. We argue that scientific users tend to repeat similar requests over similar data as they iterate their understanding or process various samples in the exploration of variants and experimentation settings. Thus in Chapter 3, we have proposed the idea of using performance data gathered from previous enactments to optimise the mapping process. These performance data are associated with individual components in the workflows; this retains information as the workflows are revised and to some extent insulates the performance data from the workflow variations. To realise this idea, we have:

- defined the streaming workflow model,
- conducted preliminary experiments to study the enactment behaviour of streaming workflows and to explore parallelism in their execution,
- defined the conceptual model for optimising the mapping process,
- created a new fine-grained measurement framework to capture performance-related data during workflow enactment,
- created a new performance database to organise the performance data systematically, and

- demonstrated how it can be queried to determine parameters that shape the optimisation of streaming workflows.

We have conducted a series of experiments to evaluate the optimisation model and architecture in Chapter 5. We have demonstrated that optimisation of data-streaming workflows can be automated by exploiting performance data. The experiments have been conducted in phases. The results show the feasibility of our idea, and have been published. We have conducted a preliminary experiment to evaluate the three-stage mapping algorithm. We have presented a detail experimental plan together with the analyses and evidence requirements for further evaluating the efficiency of the three-phase mapping algorithm. This is future work and the experiment preparation is ongoing. It needs to be conducted with real workloads and real users to to test the validity of the assumption that the performance data have sufficient stability.

The contributions of this thesis can be summarised as:

1. Influence on the design and implementation of a new data-intensive architecture.
2. Influence on the design of a new language for describing data-intensive workflows.
3. An affordable fine-grained measurement framework to collect performance-related data from the enactment of data-streaming workflows.
4. A systematic way to organise and manipulate performance-related data with a performance database.
5. A demonstration of how to obtain critical optimisation parameters by querying the performance database.
6. A novel three-stage mapping algorithm that maps PEs of data-streaming workflows onto execution engines.
7. A demonstration by hand optimisation of data-streaming workflows using performance data.

6.2 Future work

In the course of the work, we have come across many potential research directions that can be build on the work in this thesis. We will discuss them from two aspects: architecture and algorithms. The architecture aspect includes the data-intensive architecture (⇝*4.1*), the DISPEL language (⇝*4.2*), the measurement framework (⇝*4.3*) and the performance database (⇝*4.4*). The algorithm aspect should explain how the mapping algorithm (⇝*3.5*) can be extended in the future.

Architecture aspect

1. *Dynamic optimisation.* The optimiser makes all of the decisions before the concrete workflows are deployed. From the deployment stage onwards, the optimiser does not interfere with the execution. The design of the data-intensive architecture is intended to enable the enactment of DISPEL on different data-intensive platforms, and not every platform allows the alteration of PE instances during the execution. However, given the ability to control the state of PE instances and the data streams on the execution engines, dynamic optimisation, which looks at load-balancing during runtime, is achievable. We illustrate this with the EURExpress-II cross-validation workflow discussed in Section 5.3.2. When the buffer observers placed in all of the input streams of `Classifier` PE (i.e. PE_9) observed a slow input data rate on one of the input streams, the enactor could have halted the processing pipeline for the testing dataset, and duplicated a parallel pipeline on another DIVM. It would then have reconnected the data streams and resumed the execution. This is a kind of context-aware delayed-execution model where the optimiser will introduce parallelism based on the resource availability and the size of data streams at runtime. A corollary to this is to explore context-aware pattern generation. Recall that patterns are represented as functions and that specific implementations of a pattern are generated by parameterised function application. If the closure representing the function application and its parameters are passed to the execution context, then the pattern can be generated easily when the resulting graph is first needed. In which case, the function could then perform context-dependent expansion, e.g. generating a form which matches the number of CPUs available.

2. *Dynamic insertion of observers.* In our prototype, the use of observers is determined by a data-intensive engineer prior to the execution. There are two options available, either enabling the use of buffer observers on all of the data streams, or disabling all of them. The ideal measurement framework should handle this decision automatically. One of the possible ways is to design a hierarchical measurement framework that comprises observers with different measurement granularities. For each enactment, coarse-grained observers will be placed at the major inputs and outputs of a sub-graph, e.g. the processing pipeline for a training dataset. During the execution, the measurement framework will monitor the performance data collected by these observers, and insert fine-grained observers to capture the events on every PEIs of suspected pipeline. A more practical way is using the performance database to determine the need to use observer on

the PEIs, where only fresh or newly introduced PEIs (or revised ones) will be measured.

3. *Real-time processing of performance data.* The performance data is captured by the measurement framework and stored in the performance database. In the current implementation, the gatherer holds all of the performance data sent by observers in main memory, and populates the performance database after the enactment. This implementation has two major defects. First, it is not scalable because the gatherer will not be able to process the enactment of large-scale workflows when it hits the memory wall. The second problem is the discard phase in the performance data life-cycle (⇝*4.4*) needs to be run very frequently to clear the space. A possible enhancement is to process the performance data on the fly. The gatherer can perform incremental statistical analysis upon receiving the data from observers, and only populate the performance database with derived data. The main challenge is to ensure that the gatherer will understand well the cost model, and not remove any important information from the raw data, as the discarded raw data is not retrievable.

4. *Dynamic buffering implementation.* The data-intensive architecture uses data streams to connect PEs so as to construct workflows. PEs may have several inputs (⇝*3.1*) and consequently buffering may be needed within a data stream to handle different production and consumption rates. Thus, the overall workload requires arbitrary sized buffers due to mismatched processing speeds along different branches of a workflow graph. The challenge is to show how a set of implementations of data streaming can be designed so that they have efficient access patterns for the various scales of buffering required. A possible future research direction is to formulate a model to predict streaming performance corresponding to buffering strategy and then optimise data streaming by dynamically inserting appropriate buffers when the default simple buffers that allow fast, low cost buffering would be overwhelmed.

5. *Spanning onto other execution models.* The current research is built on the streaming processing model. The underlying enactment technology is OGSA-DAI. To support a wide range of enactment technologies, the architecture would have to incorporate other execution models, which might involve a hybrid system. For instance, a DISPEL request for the seismic ambient noise processing may be compiled into two concrete workflows, one to be enacted on OGSA-DAI server to perform data preprocessing, and the other is sent to DAGMan to process the cross-correlation as batch jobs. As another example, extracts of geospatial

is often needed in combination with the result data for presentation. But the geospatial standards use BPEL to derive data, so that part part of the workflow would needs to be compiled to BPEL. The main research question is how the architecture, including the gateway, the optimiser, the enactment engine, the measurement framework and the performance database can can be adapted to such a hybrid model without succumbing to excessive complexity.

Algorithm aspect

1. *Conducting an extensive evaluation on the three-phases mapping algorithm.* We have proposed an experimental plan that extends the experiments in third phase for a thorough evaluation of the mapping algorithm (⇝*5.4*). An immediate future work will be to conduct the experiment. In the experimental plan, we proposed to use two real-world scientific applications from two different domains to generate the test workload, and enact it on different executing environments. This experiment will not only evaluate the efficiency of the algorithm, but also examine the capability of the data-intensive machine, as compared with a cluster machine that are accessible to most of the scientific communities. For a thorough evaluation, we suggest to add more applications from other domains to assess the proficiency of the algorithm in supporting a broad spectrum of data-intensive applications. In each case, it is necessary to work with real users, in our case biologists and seismologists, so that the way in which they repeat and change their work can be properly accommodated by the optimisation algorithm, A repetition of a carefully logged sequence of runs allows experimental exploration.

2. *Exploring different heuristic techniques for mapping PEIs.* We treat the mapping of heavy PEIs as a DAG scheduling problem. Existing heuristic techniques are available and their use in scheduling optimisation has been explored. This give the opportunity to investigate the possibility of adapting these heuristic techniques for streaming workflows. Potentially, the current mapping algorithms can be further improved in many ways. For instance, by considering the use of overflow DIVMs when sliding light PEIs and by adding communication costs into the calculation when assigning heavy PEIs. Research is needed to discover whether the benefits justify the added complexity.

3. *Exploring the use of machine-learning algorithm for exploring the performance threshold.* The threshold plays an important role in the mapping algorithm to partition the PEIs prior to the mapping process. Identifying the threshold re-

mains a main challenge as it determine the efficiency of the mapping. We have argued that this value should be domain and site specific (⇝*3.4.2*). Thus, the identification process should be automated. Future work should explore a systematic way to automate the calculation of this threshold using a machine-learning algorithm.

4. *Exploring graph partitioning techniques for DIVM allocation.* The graph representing a data-intensive workflow can be annotated with performance data extracted from the performance database to form a weighted graph. The weights on the vertices represent unit costs and the weights on edges represent data movement costs. The DAG is expected to be split into multiple DAGs and enacted on separate DIVMs. This is similar to a graph partitioning problem. An interesting future research is to use graph partitioning algorithms off the shelf to cut the DAG into n partitions for the execution on n DIVMs. This could be extended to exploit DIVM heterogeneity.

5. *Exploring the DIVM allocation as a knapsack problem formulated as a constraint satisfaction combinatorial optimisation problem.* In our research, we use time as an approximation for all of the limiting factors in allocating DIVMs, e.g. CPU, memory, disk I/O and the network bandwidth. This is a multiple-objective knapsack problem, where the PEIs can be denoted as a set of items with different computational workloads, memory consumptions, I/O rates and data movements, that are to be packed into a set of DIVMs, that are symbolised as the knapsacks. To further complicate the problem, the knapsacks are different in terms of their computational and communicational capacity. Solving the DIVMs allocation requires a substantial work to understand each of these limiting factors, derive a cost model and solve it with optimisation algorithms to produce near-optimal solutions. This can also remove the biggest assumption that we have made to reduce the complexity in designing the mapping algorithm—homogenous execution environment. This would reveal whether the extra complexity and optimisation costs would yield commensurate benefits.

6.3 Final thoughts

Our research focuses on a subclass of scientific workflows, where the data are passed in a streaming manner. We have proved the feasibility of data-stream processing with real-world scientific use cases running on our data-intensive architecture. The streaming processing model does not replace the batch-job processing model for enacting work-

flows, but is an alternative applicable in a significant number of circumstances. The potential of a hybrid model that is a combination of the two should be investigated.

In our research, we used workflows from the scientific domains in the design of the optimisation model and architecture. However, this should not limit their usage to other activities that are using data, e.g. analysis of tweets social-networks research, and deriving information or evidence to support decision making and policy formation in the business domain. We believe that our work could be applicable in a broader context with certain adaptations.

The data-intensive architecture is being used by several projects, and the development is still ongoing—including the enactment on other execution platforms and the exploration of other optimisation techniques. DISPEL has been proven in the ADMIRE project, to be precise and suitable language for data-intensive computing, even though its prototype does not include all of the features and functionalities that have been designed. The work on DISPEL is continuing to support the research community.

Up to this point, our work has been tested on several domains with gigabyte scale data. How far we can cope with peta-scale and beyond? What level of efficiency can we achieve when handling applications that are outside our current domains? Can our work survive when we encounter the next major digital technology transaction? These all remain as open questions. The end of this thesis does not mark the end of our work, but an opportunity to address these questions in depth.

Bibliography

[1] David Abramson, Colin Enticott, and Ilkay Altinas, "Nimrod/K: towards massively parallel dynamic grid workflows," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pp. 24:1–24:11, IEEE Press, November 2008. [pp. 25 and 42.]

[2] Bernie Ács, Xavier Llorà, Loretta Auvil, Boris Capitanu, David Tcheng, Mike Haberman, Limin Dong, Tim Wentling, and Michael Welge, "A general approach to data-intensive computing using the Meandre component-based framework," in *Proceedings of the 1st International Workshop on Workflow Approaches to New Data-centric Science*, WANDS '10, pp. 8:1–8:12, ACM, 2010. [pp. 48.]

[3] Aashish N. Adhikari, Jian Peng, Michael Wilde, Jinbo Xu, Karl F. Freed, and Tobin R. Sosnick, "Modeling large regions in proteins: applications to loops, termini, and folding," *Protein Science*, 21(1):107–121, January 2012. [pp. 37.]

[4] Rakesh Agrawal, Anastasia Ailamaki, Philip A. Bernstein, Eric A. Brewer, Michael J. Carey, Surajit Chaudhuri, Anhai Doan, Daniela Florescu, Michael J. Franklin, Hector Garcia-Molina, Johannes Gehrke, Le Gruenwald, Laura M. Haas, Alon Y. Halevy, Joseph M. Hellerstein, Yannis E. Ioannidis, Hank F. Korth, Donald Kossmann, Samuel Madden, Roger Magoulas, Beng Chin Ooi, Tim O'Reilly, Raghu Ramakrishnan, Sunita Sarawagi, Michael Stonebraker, Alexander S. Szalay, and Gerhard Weikum, "The Claremont report on database research," *Communications of the ACM*, 52(6):56–65, June 2009. [pp. 2.]

[5] Chris Allan, Jean-Marie Burel, Josh Moore, Colin Blackburn, Melissa Linkert, Scott Loynton, Donald MacDonald, William J Moore, Carlos Neves, Andrew Patterson, Michael Porter, Aleksandra Tarkowska, Brian Loranger, Jerome Avondo, Ingvar Lagerstedt, Luca Lianas, Simone Leo, Katherine Hands, Ron T Hay, Ardan Patwardhan, Christoph Best, Gerard J Kleywegt, Gianluigi Zanetti, and Jason R Swedlow, "OMERO: flexible, model-driven data management for experimental biology," *Nature Methods*, 9(3):245–253, March 2012. [pp. 4.]

[6] Ilkay Altintas, Oscar Barney, and Efrat Jaeger-Frank, "Provenance collection support in the Kepler scientific workflow system," in *Provenance and Annotation of Data* (Luc

Moreau and Ian Foster, eds.), vol. 4145 of *Lecture Notes in Computer Science*, pp. 118–132, Springer Berlin / Heidelberg, 2006. [pp. 25 and 42.]

[7] Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Philippe Selo, Yoonho Park, and Chitra Venkatramani, "SPC: a distributed, scalable platform for data mining," in *Proceedings of the 4th international workshop on Data mining standards, services and platforms*, DMSSP '06, pp. 27–37, ACM, 2006. [pp. 61.]

[8] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia, "A view of cloud computing," *Communications of the ACM*, 53(4):50–58, April 2010. [pp. 20.]

[9] Malcolm P. Atkinson, "Data-intensive thinking with DISPEL," in Atkinson *et al.* [10]. [pp. 101.]

[10] Malcolm P. Atkinson, Rob Baxter, Paolo Besana, Michelle Galea, Mark Parsons, Peter Brezany, Oscar Corcho, Jano van Hemert, and David Snelling, *The DATA Bonanza – Improving Knowledge Discovery for Science, Engineering and Business.* Wiley, to be published in 2012. [pp. 8, 11, 98, 101, 105, 107, 109, 170, 175, 178, and 180.]

[11] Malcolm P. Atkinson, Chee Sun Liew, Michelle Galea, Paul Martin, Amrey Krause, Adrian Mouat, Oscar Corcho, and David Snelling, "Data-intensive architecture for scientific knowledge discovery," *Distributed and Parallel Databases*, 30(5):307–324, October 2012. [pp. 7.]

[12] Malcolm P. Atkinson, Jano I. van Hemert, Liangxiu Han, Ally Hume, and Chee Sun Liew, "A distributed architecture for data mining and integration," in *Proceedings of the second international workshop on Data-aware distributed computing*, DADC '09, pp. 11–20, ACM, 2009. [pp. 7.]

[13] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom, "Models and issues in data stream systems," in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '02, pp. 1–16, ACM, June 2002. [pp. 32 and 61.]

[14] Roger Barga and Dennis Gannon, "Scientific versus business workflows," in Taylor *et al.* [164], pp. 9–16. [pp. 26.]

[15] Roger Barga, Jared Jackson, Nelson Araujo, Dean Guo, Nitin Gautam, and Yogesh Simmhan, "The Trident scientific workflow workbench," in *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, e-Science '08, pp. 317–318, IEEE Computer Society, December 2008. [pp. 25.]

[16] Adam Barker, Christopher D. Walton, and David Robertson, "Choreographing web services," *IEEE Transactions on Services Computing*, 2(2):152–166, April-June 2009. [pp. 32.]

[17] Adam Barker, Jon B. Weissman, and Jano van Hemert, "Orchestrating data-centric workflows," in *8th IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '08, pp. 210–217, May 2008. [pp. 32.]

[18] Jörg Becker, Michael zur Muehlen, and Marc Gille, "Workflow application architectures: Classification and characteristics of workflow-based information systems," in *Workflow Handbook 2002* (Layna Fischer, ed.), pp. 39–50, Future Strategies, Lighthouse Point, FL, 2002. [pp. 29.]

[19] Gordon Bell, Jim Gray, and Alex Szalay, "Petascale computational systems," *Computer*, 39(1):110–112, January 2006. [pp. 2.]

[20] G. D. Bensen, M. H. Ritzwoller, M. P. Barmin, A. L. Levshin, F. Lin, M. P. Moschetti, N. M. Shapiro, and Y. Yang, "Processing seismic ambient noise data to obtain reliable broad-band surface wave dispersion measurements," *Geophysical Journal International*, 169(3):1239–1260, June 2007. [pp. 122.]

[21] G. Bruce Berriman, Ewa Deelman, John Good, Joseph C. Jacob, Daniel S. Katz, Anastasia C. Laity, Thomas A. Prince, Gurmeet Singh, and Mei-Hui Su, "Generating complex astronomy workflows," in Taylor *et al.* [164], pp. 19–38. [pp. 2.]

[22] G. Bruce Berriman, Ewa Deelman, Paul T. Groth, and Gideon Juve, "The application of cloud computing to the creation of image mosaics and management of their provenance," in *Software and Cyberinfrastructure for Astronomy* (Nicole M. Radziwill and Alan Bridger, eds.), vol. 7740, p. 77401F, SPIE, June 2010. [pp. 34.]

[23] G. Bruce Berriman and Steven L. Groom, "How will astronomy archives survive the data tsunami?," *Communications of the ACM*, 54(12):52–56, December 2011. [pp. 2 and 17.]

[24] Shishir Bharathi, Ann Chervenak, Ewa Deelman, Gaurang Mehta, Mei-Hui Su, and Karan Vahi, "Characterization of scientific workflows," in *Proceedings of the 3rd workshop on Workflows in support of large-scale science*, WORKS '08, pp. 1–10, IEEE Computer Society, November 2008. [pp. 2 and 22.]

[25] Ernst W. Biersack, Bianca Schroeder, and Guillaume Urvoy-Keller, "Scheduling in practice," *SIGMETRICS Perform. Eval. Rev.*, 34(4):21–28, Mar. 2007. [pp. 95.]

[26] Biörn Biörnstad, Cesare Pautasso, and Gustavo Alonso, "Control the flow: How to safely compose streaming services into business processes," in *Proceedings of the IEEE International Conference on Services Computing*, SCC '06, pp. 206–213, IEEE Computer Society, September 2006. [pp. 61.]

[27] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold, "Breaking the memory wall in MonetDB," *Communications of the ACM*, 51(12):77–85, December 2008. [pp. 17.]

[28] Shawn Bowers and Bertram Ludäscher, "Actor-oriented design of scientific workflows," in *Conceptual Modeling – ER 2005* (Lois Delcambre, Christian Kop, Heinrich Mayr, John Mylopoulos, and Oscar Pastor, eds.), vol. 3716 of *Lecture Notes in Computer Science*, pp. 369–384, Springer Berlin / Heidelberg, 2005. [pp. 6 and 40.]

[29] Shawn Bowers, Timothy McPhillips, Martin Wu, and Bertram Ludäscher, "Project histories: Managing data provenance across collection-oriented scientific workflow runs," in *Data Integration in the Life Sciences* (Sarah Cohen-Boulakia and Val Tannen, eds.), vol. 4544 of *Lecture Notes in Computer Science*, pp. 122–138, Springer Berlin / Heidelberg, 2007. [pp. 25 and 42.]

[30] Tracy D. Braun, Howard Jay Siegel, Noah Beck, Ladislau L. Bölöni, Muthucumaru Maheswaran, Albert I. Reuther, James P. Robertson, Mitchell D. Theys, Bin Yao, Debra Hensgen, and Richard F. Freund, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *Journal of Parallel and Distributed Computing*, 61(6):810–837, June 2001. [pp. 72.]

[31] Gert Brettlecker, Heiko Schuldt, and Hans-Jörg Schek, "Towards Reliable Data Stream Processing with OSIRIS-SE," in *BTW* (Gottfried Vossen, Frank Leymann, Peter C. Lockemann, and Wolffried Stucky, eds.), vol. 65 of *LNI*, pp. 405–414, GI, March 2005. [pp. 61.]

[32] Christopher Brooks, Edward A. Lee, Xiaojun Liu, Stephen Neuendorffer, Yang Zhao, and Haiyang Zheng, "Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II)," Tech. Rep. UCB/EECS-2007-7, EECS Department, University of California, Berkeley, January 2007. [pp. 42.]

[33] Duncan A. Brown, Patrick R. Brady, Alexander Dietz, Junwei Cao, Ben Johnson, and John McNabb, "A case study on the use of workflow technologies for scientific analysis: Gravitational wave data analysis," in Taylor *et al.* [164], pp. 39–59. [pp. 2.]

[34] Erik Brynjolfsson, Paul Hofmann, and John Jordan, "Cloud computing and electricity: beyond the utility model," *Communications of the ACM*, 53(5):32–34, May 2010. [pp. 20.]

[35] Scott Callaghan, Ewa Deelman, Dan Gunter, Gideon Juve, Philip Maechling, Christopher Brooks, Karan Vahi, Kevin Milner, Robert Graves, Edward Field, David Okaya, and Thomas Jordan, "Scaling up workflow-based applications," *Journal of Computer and System Sciences*, 76(6):428–446, 2010. [pp. 3, 27, and 36.]

[36] Steven P. Callahan, Juliana Freire, Emanuele Santos, Carlos E. Scheidegger, Cláudio T. Silva, and Huy T. Vo, "Managing the evolution of dataflows with VisTrails," in *Proceedings of the 22nd International Conference on Data Engineering Workshops*, ICDEW '06, p. 71, IEEE Computer Society, 2006. [pp. 4.]

[37] Thomas L. Casavant and Jon G. Kuhl, "A taxonomy of scheduling in general-purpose distributed computing systems," *IEEE Transactions on Software Engineering*, 14(2):141–154, February 1988. [pp. 72.]

[38] Bradford L. Chamberlain, "Graph partitioning algorithms for distributing workloads of parallel computations," Tech. Rep. UW-CSE-98-10-03, University of Washington, October 1998. [pp. 95.]

[39] Surajit Chaudhuri, "An overview of query optimization in relational systems," in *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '98, pp. 34–43, ACM, 1998. [pp. 59.]

[40] Jinjun Chen and Yun Yang, "A taxonomy of grid workflow verification and validation," *Concurrency and Computation: Practice and Experience*, 20(4):347–360, March 2008. [pp. 29.]

[41] Weiwei Chen and Ewa Deelman, "Workflow overhead analysis and optimizations," in *Proceedings of the 6th workshop on Workflows in support of large-scale science*, WORKS '11, pp. 11–20, ACM, November 2011. [pp. 36.]

[42] Peter Couvares, Tevfik Kosar, Alain Roy, Jeff Weber, and Kent Wenger, "Workflow Management in Condor," in Taylor *et al.* [164], pp. 357–375. [pp. 1 and 24.]

[43] Vasa Curcin and Moustafa Ghanem, "Scientific workflow systems - can one size fit all?," in *Cairo International Biomedical Engineering Conference*, CIBEC '08, pp. 1–9, December 2008. [pp. 1.]

[44] Andrew Curtis, Peter Gerstoft, Haruo Sato, Roel Snieder, and Kees Wapenaar, "Seismic interferometry—turning noise into signal," *The Leading Edge*, 25(9):1082–1092, September 2006. [pp. 121.]

[45] Sérgio Manuel Serra da Cruz, Maria Luiza M. Campos, and Marta Mattoso, "Towards a taxonomy of provenance in scientific workflow management systems," in *Proceedings of the 2009 IEEE Congress on Services - Part I*, SERVICES '09, pp. 259–266, IEEE Computer Society, July 2009. [pp. 30.]

[46] Susan B. Davidson and Juliana Freire, "Provenance and scientific workflows: challenges and opportunities," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pp. 1345–1350, ACM, June 2008. [pp. 25.]

[47] David De Roure, Kevin R. Page, Benjamin Fields, Tim Crawford, J. Stephen Downie, and Ichiro Fujinaga, "An e-Research approach to Web-scale music analysis," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 369(1949):3300–3317, August 2011. [pp. 47.]

[48] Ewa Deelman, "Grids and clouds: Making workflow applications work in heterogeneous distributed environments," *International Journal of High Performance Computing Applications*, 24(3):284–298, August 2010. [pp. 19 and 35.]

[49] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Sonal Patil, Mei-Hui Su, Karan Vahi, and Miron Livny, "Pegasus: Mapping scientific workflows onto the grid," in *Grid Computing* (Marios Dikaiakos, ed.), vol. 3165 of *Lecture Notes in Computer Science*, pp. 11–20, Springer Berlin / Heidelberg, 2004. [pp. 24.]

[50] Ewa Deelman, Scott Callaghan, Edward Field, Hunter Francoeur, Robert Graves, Nitin Gupta, Vipin Gupta, Thomas H. Jordan, Carl Kesselman, Philip Maechling, John Mehringer, Gaurang Mehta, David Okaya, Karan Vahi, and Li Zhao, "Managing Large-Scale Workflow Execution from Resource Provisioning to Provenance Tracking: The CyberShake Example," in *Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, e-Science '06, p. 14, December 2006. [pp. 36.]

[51] Ewa Deelman, Dennis Gannon, Matthew Shields, and Ian Taylor, "Workflows and e-Science: An overview of workflow system features and capabilities," *Future Generation Computer Systems*, 25(5):528–540, May 2009. [pp. 1, 22, 23, and 29.]

[52] Ewa Deelman and Yolanda Gil, "Managing large-scale scientific workflows in distributed environments: Experiences and challenges," in *Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, e-Science '06, p. 144, December 2006. [pp. 25.]

[53] Ewa Deelman, Gaurang Mehta, Gurmeet Singh, Mei-Hui Su, and Karan Vahi, "Pegasus: Mapping large-scale workflows to distributed resources," in Taylor *et al.* [164], pp. 376–394. [pp. 56.]

[54] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia C. Laity, Joseph C. Jacob, and Daniel S. Katz, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, 13(3):219–237, 2005. [pp. 3 and 25.]

[55] Bartosz Dobrzelecki, Amy Krause, Ally Hume, Andrew Grant, Mario Antonioletti, Tilaye Alemu, Malcolm P. Atkinson, Michael Jackson, and Elias Theocharopoulos, "Integrating Distributed Data Sources with OGSA-DAI DQP and Views," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 368(1926):4133–4145, September 2010. [pp. 7, 60, and 117.]

[56] Lei Dou, Daniel Zinn, Timothy McPhillips, Sven Kohler, Sean Riddle, Shawn Bowers, and Bertram Ludäscher, "Scientific workflow design 2.0: Demonstrating streaming data collections in Kepler," in *Proceedings of the IEEE 27th International Conference on Data Engineering*, ICDE '11, pp. 1296–1299, April 2011. [pp. 52.]

[57] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong, "Taming heterogeneity - the Ptolemy approach," *Proceedings of the IEEE*, 91(1):127–144, January 2003. [pp. 40.]

[58] Erik Elmroth, Francisco Hernández, and Johan Tordsson, "Three fundamental dimensions of scientific workflow interoperability: Model of computation, language, and execution environment," *Future Generation Computer Systems*, 26(2):245–256, February 2010. [pp. 20.]

[59] Wolfgang Emmerich, Ben Butchart, Liang Chen, Bruno Wassermann, and Sarah Price, "Grid Service Orchestration Using the Business Process Execution Language (BPEL)," *Journal of Grid Computing*, 3(3-4):283–304, September 2005. [pp. 31.]

[60] Thomas Fahringer, Radu Prodan, Rubing Duan, Jüurgen Hofer, Farrukh Nadeem, Francesco Nerieri, Stefan Podlipnig, Jun Qin, Mumtaz Siddiqui, Hong-Linh Truong, Alex Villazon, and Marek Wieczorek, "ASKALON: A Development and Grid Computing Environment for Scientific Workflows," in Taylor *et al.* [164], pp. 450–471. [pp. 24 and 56.]

[61] David Fernandez-Baca, "Allocating modules to processors in a distributed system," *IEEE Transactions on Software Engineering*, 15(11):1427–1436, November 1989. [pp. 72.]

[62] Ronald A. Fisher, "The use of multiple measurements in taxonomic problems," *Annals of Human Genetics*, 7(2):179–188, September 1936. [pp. 120.]

[63] Michael J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, C-21(9):948–960, September 1972. [pp. 63.]

[64] Ruslan Fomkin and Tore Risch, "Cost-based optimization of complex scientific queries," in *Proceedings of the 19th International Conference on Scientific and Statistical Database Management*, SSBDM '07, pp. 1–1, IEEE Computer Society, July 2007. [pp. 59.]

[65] Ian Foster, Jens Vöckler, Michael Wilde, and Yong Zhao, "Chimera: a virtual data system for representing, querying, and automating data derivation," in *Proceedings of the 14th International Conference on Scientific and Statistical Database Management*, SSDBM '02, pp. 37–46, July 2002. [pp. 20.]

[66] Michelle Galea, Andreas Rietbrock, Alessandro Spinuso, and Luca Trani, "Data-intensive seismology: research horizons," in Atkinson *et al.* [10]. [pp. 122 and 154.]

[67] Dennis Gannon, "Component architectures and services: From application construction to scientific workflows," in Taylor *et al.* [164], pp. 174–189. [pp. 31.]

[68] Dennis Gannon, Beth Plale, Suresh Marru, Gopi Kandaswamy, Yogesh Simmhan, and Satoshi Shirasuna, "Dynamic, adaptive workflows for mesoscale meteorology," in Taylor *et al.* [164], pp. 126–142. [pp. 28.]

[69] Michael R. Garey and David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979. [pp. 72.]

[70] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo, "SPADE: the System S declarative stream processing engine," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pp. 1123–1134, ACM, June 2008. [pp. 62.]

[71] Yolanda Gil, Ewa Deelman, Mark Ellisman, Thomas Fahringer, Geoffrey Fox, Dennis Gannon, Carole Goble, Miron Livny, Luc Moreau, and Jim Myers, "Examining the challenges of scientific workflows," *Computer*, 40(12):24–32, December 2007. [pp. 5, 25, and 27.]

[72] Yolanda Gil, Jihie Kim, Varun Ratnakar, and Ewa Deelman, "Wings for Pegasus: A semantic approach to creating very large scientific workflows," in *Proceedings of the OWLED*06 Workshop on OWL: Experiences and Directions*, vol. 216 of *CEUR Workshop Proceedings*, CEUR-WS.org, November 2006. [pp. 34.]

[73] Yolanda Gil, Varun Ratnakar, Ewa Deelman, Gaurang Mehta, and Jihie Kim, "Wings for Pegasus: Creating large-scale scientific applications using semantic representations of computational workflows," in *Proceedings of the Nineteenth Conference on Innovative Applications of Artificial Intelligence*, IAAI '07, pp. 1767–1774, July 2007. [pp. 24.]

[74] Tristan Glatard, Johan Montagnat, David Emsellem, and Diane Lingrand, "A service-oriented architecture enabling dynamic service grouping for optimizing distributed workflow execution," *Future Generation Computer Systems*, 24(7):720–730, July 2008. [pp. 58.]

[75] Tristan Glatard, Johan Montagnat, Diane Lingrand, and Xavier Pennec, "Flexible and efficient workflow deployment of data-intensive applications on grids with MOTEUR," *International Journal of High Performance Computing Applications*, 22(3):347–360, August 2008. [pp. 58.]

[76] Tristan Glatard, Gergely Sipos, Johan Montagnat, Zoltan Farkas, and Peter Kacsuk, "Workflow-Level Parametric Study Support by MOTEUR and the P-GRADE Portal," in Taylor *et al.* [164], pp. 279–299. [pp. 54, 57, and 62.]

[77] Carole Goble and David De Roure, "The impact of workflow tools on data-centric research," in Hey *et al.* [99], pp. 137–145. [pp. 25.]

[78] Lukasz Golab, "Data stream," in Liu and Özsu [121], pp. 638–638. [pp. 61.]

[79] Michael I. Gordon, William Thies, and Saman Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pp. 151–162, ACM, 2006. [pp. 61, 62, 64, and 65.]

[80] Katharina Görlach, Mirko Sonntag, Dimka Karastoyanova, Frank Leymann, and Michael Reiter, "Conventional workflow technology for scientific simulation," in *Guide to e-Science* (Xiaoyu Yang, Lizhe Wang, and Wei Jie, eds.), Computer Communications and Networks, pp. 323–352, Springer London, 2011. [pp. 1, 22, and 25.]

[81] Ian Gorton, Paul Greenfield, Alex Szalay, and Roy Williams, "Data-intensive computing in the 21st century," *Computer*, 41(4):30–32, April 2008. [pp. 17.]

[82] Torsten Grabs, Roman Schindlauer, Ramkumar Krishnan, Jonathan Goldstein, and Rafael Fernández, "Introducing Microsoft StreamInsight," tech. rep., Microsoft Corporation, 2009. [pp. 61.]

[83] Goetz Graefe, "Query evaluation techniques for large databases," *ACM Computing Surveys*, 25(2):73–169, June 1993. [pp. 59.]

[84] Robert Graves, Thomas Jordan, Scott Callaghan, Ewa Deelman, Edward Field, Gideon Juve, Carl Kesselman, Philip Maechling, Gaurang Mehta, Kevin Milner, David Okaya, Patrick Small, and Karan Vahi, "CyberShake: A Physics-Based Seismic Hazard Model for Southern California," *Pure and Applied Geophysics*, 168(3-4):367–381, March 2011. [pp. 3.]

[85] Jim Gray, "Jim Gray on eScience: A Transformed Scientific Method," in Hey *et al.* [99], pp. xix–xxxiii. [pp. 2, 17, and 124.]

[86] Paul Grefen and Jochem Vonk, "A taxonomy of transactional workflow support," *International Journal of Cooperative Information Systems*, 15(1):87–118, March 2006. [pp. 29.]

[87] Yunhong Gu and Robert L. Grossman, "Sector and Sphere: the design and implementation of a high-performance data cloud," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 367(1897):2429–2445, June 2009. [pp. 19.]

[88] Jayanth Gummaraju, Joel Coburn, Yoshio Turner, and Mendel Rosenblum, "Streamware: programming general-purpose multicore processors using streams," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pp. 297–307, ACM, 2008. [pp. 62.]

[89] Liangxiu Han, Ally Hume, Chee Sun Liew, Jano I. van Hemert, and Malcolm P. Atkinson, "ADMIRE – Research Prototype: DMI Using a Pipeline," tech. rep., The ADMIRE Project, 2009. [pp. 159.]

[90] Liangxiu Han, Chee Sun Liew, Jano I. van Hemert, and Malcolm P. Atkinson, "A generic parallel processing model for facilitating data mining and integration," *Parallel Computing*, 37(3):157–171, March 2011. [pp. 7 and 58.]

[91] Liangxiu Han, Chee Sun Liew, Gagarine Yaikhom, Jano I. van Hemert, and Malcolm P. Atkinson, "ADMIRE – Research Prototype: Parallel Processing a DMI task," tech. rep., The ADMIRE Project, 2010. [pp. 159.]

[92] Liangxiu Han, Jano I. van Hemert, and Richard A. Baldock, "Automatically identifying and annotating mouse embryo gene expression patterns," *Bioinformatics*, 27(8):1101–1107, April 2011. [pp. 120.]

[93] Yanbo Han, A Sheth, and Christoph Bussler, "A taxonomy of adaptive workflow management," in *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, pp. 1–11, ACM, November 1998. [pp. 30.]

[94] Mihael Hategan, Justin Wozniak, and Ketan Maheshwari, "Coasters: uniform resource provisioning and access for clouds and grids," in *Proceedings of the 2011 Fourth IEEE International Conference on Utility and Cloud Computing*, UCC '11, pp. 114–121, IEEE Computer Society, December 2011. [pp. 39.]

[95] Brian Hayes, "Cloud computing," *Communications of the ACM*, 51(7):9–11, July 2008. [pp. 20.]

[96] George Heald, Michael Bell, Andreas Horneffer, André Offringa, Roberto Pizzo, Sebastiaan van der Tol, Reinout van Weeren, Joris van Zwieten, James Anderson, Rainer Beck, Ilse van Bemmel, Laura Bîrzan, Annalisa Bonafede, John Conway, Chiara Ferrari, Francesco De Gasperin, Marijke Haverkorn, Neal Jackson, Giulia Macario, John McKean, Halime Miraghaei, Emanuela Orrù, David Rafferty, Huub Röttgering, Anna Scaife, Aleksandar Shulevski, Carlos Sotomayor, Cyril Tasse, Monica Trasatti, and Olaf Wucknitz, "LOFAR: Recent Imaging Results and Future Prospects," *Journal of Astrophysics and Astronomy*, 32(4):1–10, December 2011. [pp. 3.]

[97] Bruce Hendrickson and Tamara G. Kolda, "Graph partitioning models for parallel computing," *Parallel Computing*, 26(12):1519–1534, 2000. [pp. 95.]

[98] Tony Hey, Dennis Gannon, and Jim Pinkelman, "The future of data-intensive science," *Computer*, 45(5):81–82, May 2012. [pp. 2.]

[99] Tony Hey, Stewart Tansley, and Kristin Tolle, eds., *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Redmond, WA: Microsoft Research, 2009. [pp. 2, 16, 176, and 183.]

[100] Duncan Hull, Katy Wolstencroft, Robert Stevens, Carole Goble, Mathew R. Pocock, Peter Li, and Tom Oinn, "Taverna: a tool for building and running workflows of services," *Nucleic Acids Research*, 34(suppl 2):W729–W732, July 2006. [pp. 25.]

[101] E. Ilavarasan, P. Thambidurai, and R. Mahilmannan, "Performance effective task scheduling algorithm for heterogeneous computing system," in *Proceedings of the 4th International Symposium on Parallel and Distributed Computing*, ISPDC '05, pp. 28–38, IEEE Computer Society, July 2005. [pp. 72.]

[102] Institute for Telecommunication Sciences, *Federal Standard 1037C, Telecommunications: Glossary of Telecommunication Terms*, August 1996. [pp. 61.]

[103] Interagency Working Group on Digital Data, "Harnessing the power of digital data for science and society: report of the interagency working group on digital data to the national science and technology council," tech. rep., Executive office of the President, Office of Science and Technology, Washington D.C., USA, January 2009. [pp. 2, 4, and 16.]

[104] Adam Jacobs, "The pathologies of big data," *Communications of the ACM*, 52(8):36–44, August 2009. [pp. 2 and 16.]

[105] Andrew C. Jones, "Workflow and biodiversity e-science," in Taylor *et al.* [164], pp. 80–90. [pp. 27.]

[106] Douglas B. Kell and Stephen G. Oliver, "Here is the evidence, now what is the hypothesis? the complementary roles of inductive and hypothesis-driven science in the post-genomic era," *BioEssays*, 26(1):99–105, January 2004. [pp. 17.]

[107] Steve Kelling, Daniel Fink, Wesley Hochachka, Ken Rosenberg, Robert Cook, Theodoros Damoulas, Claudio Silva, and William Michener, "Estimating species distributions – across space, through time and with features of the environment," in Atkinson *et al.* [10]. [pp. 4.]

[108] Brucek Khailany, William J. Dally, Ujval J. Kapasi, Peter Mattson, Jinyung Namkoong, John D. Owens, Brian Towles, Andrew Chang, and Scott Rixner, "Imagine: media processing with streams," *IEEE Micro*, 21(2):35–46, March-April 2001. [pp. 61 and 62.]

[109] Jihie Kim, Ewa Deelman, Yolanda Gil, Gaurang Mehta, and Varun Ratnakar, "Provenance trails in the Wings/Pegasus system," *Concurrency and Computation: Practice and Experience*, 20(5):587–597, April 2008. [pp. 25 and 34.]

[110] Donald Kossmann, "The state of the art in distributed query processing," *ACM Computing Surveys*, 32(4):422–469, December 2000. [pp. 59.]

[111] Heiko Koziolek, "Introduction to performance metrics," in *Dependability Metrics* (Irene Eusgeld, Felix Freiling, and Ralf Reussner, eds.), vol. 4909 of *Lecture Notes in Computer Science*, pp. 199–203, Springer Berlin / Heidelberg, 2008. [pp. 110.]

[112] Vijay S. Kumar, Mary Hall, Jihie Kim, Yolanda Gil, Tahsin M. Kurç, Ewa Deelman, Varun Ratnakar, and Joel H. Saltz, "Designing and parameterizing a workflow for optimization: A case study in biomedical imaging," in *Proceedings of the 2008 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '08, pp. 1–5, April 2008. [pp. 2.]

[113] Marco Lackovic, Domenico Talia, Rafael Tolosana-Calasanz, José A. Bañares, and Omer F. Rana, "A taxonomy for the analysis of scientific workflow faults," in *Proceeding of the 13th IEEE International Conference on Computational Science and Engineering*, CSE '10, pp. 398–403, December 2010. [pp. 30.]

[114] Language and Architecture Team, ADMIRE project, "DISPEL: Data-Intensive Systems Process Engineering Language Users' Manual (Version 1.0)," tech. rep., School of Informatics, University of Edinburgh, 2011. [pp. 100, 101, and 109.]

[115] Edward A. Lee and Stephen Neuendorffer, "MoML – A Modeling Markup Language in XML – Version 0.4," tech. rep., University of California at Berkeley, March 2000. [pp. 100.]

[116] Chee Sun Liew, Malcolm P. Atkinson, Radosław Ostrowski, Murray Cole, Jano I. van Hemert, and Liangxiu Han, "Performance database: capturing data for optimizing distributed streaming workflows," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 369(1949):3268–3284, August 2011. [pp. 7, 114, and 159.]

[117] Chee Sun Liew, Malcolm P. Atkinson, Jano I. van Hemert, and Liangxiu Han, "Towards optimising distributed data streaming graphs using parallel streams," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing* (Salim Hariri and Kate Keahey, eds.), HPDC '10, pp. 725–736, ACM, June 2010. [pp. 7 and 159.]

[118] Chunhyeok Lim, Shiyong Lu, Artem Chebotko, and Farshad Fotouhi, "Prospective and retrospective provenance collection in scientific workflow environments," in *proceedings of the Seventh IEEE International Conference on Services Computing*, SCC '10, pp. 449–456, July 2010. [pp. 25.]

[119] Abel W. Lin, Steven T. Peltier, Jeffrey S. Grethe, and Mark H. Ellisman, "Case studies on the use of workflow technologies for scientific analysis: The biomedical informatics research network and the telescience project," in Taylor *et al.* [164], pp. 109–125. [pp. 26.]

[120] Michael Litzkow, Miron Livny, and Matthew Mutka, "Condor - a hunter of idle workstations," in *Proceedings of the 8th International Conference of Distributed Computing Systems*, pp. 104 –111, IEEE Computer Society Press, June 1988. [pp. 19 and 24.]

[121] Ling Liu and M. Tamer Özsu, eds., *Encyclopedia of Database Systems*. Springer US, 2009. [pp. 176 and 182.]

[122] Xavier Llorà, "Data-intensive computing for competent genetic algorithms: a pilot study using Meandre," in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, GECCO '09, pp. 1387–1394, ACM, July 2009. [pp. 58.]

[123] Xavier Llorà, Bernie Ács, Loretta S. Auvil, Boris Capitanu, Michael E. Welge, and David E. Goldberg, "Meandre: Semantic-driven data-intensive flows in the clouds," in *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, e-Science '08, pp. 238–245, IEEE Computer Society, 2008. [pp. 1, 25, 47, and 100.]

[124] André Luckow, Lukasz Lacinski, and Shantenu Jha, "SAGA BigJob: An extensible and interoperable pilot-job abstraction for distributed applications and systems," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pp. 135–144, IEEE Computer Society, May 2010. [pp. 39 and 156.]

[125] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao, "Scientific workflow management and the Kepler system," *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, August 2006. [pp. 25, 27, and 58.]

[126] Bertram Ludäscher, Mathias Weske, Timothy McPhillips, and Shawn Bowers, "Scientific workflows: Business as usual?," in *Business Process Management* (Umeshwar Dayal, Johann Eder, Jana Koehler, and Hajo Reijers, eds.), vol. 5701 of *Lecture Notes in Computer Science*, pp. 31–47, Springer Berlin / Heidelberg, 2009. [pp. 5, 22, and 26.]

[127] Steven Lynden, Arijit Mukherjee, Alastair C. Hume, Alvaro A.A. Fernandes, Norman W. Paton, Rizos Sakellariou, and Paul Watson, "The design and implementation of OGSA-DQP: A service-based distributed query processor," *Future Generation Computer Systems*, 25(3):224–236, March 2009. [pp. 59 and 60.]

[128] Philip Maechling, Ewa Deelman, Li Zhao, Robert Graves, Gaurang Mehta, Nitin Gupta, John Mehringer, Carl Kesselman, Scott Callaghan, David Okaya, Hunter Francoeur, Vipin Gupta, Yifeng Cui, Karan Vahi, Thomas Jordan, and Edward Field, "SCEC CyberShake Workflows—Automating Probabilistic Seismic Hazard Analysis Calculations," in Taylor *et al.* [164], pp. 143–163. [pp. 26 and 34.]

[129] Nandita Mandal, Ewa Deelman, Gaurang Mehta, Mei-Hui Su, and Karan Vahi, "Integrating existing scientific workflow systems: the Kepler/Pegasus example," in *Proceedings of the 2nd workshop on Workflows in support of large-scale science*, WORKS '07, pp. 21–28, ACM, June 2007. [pp. 19.]

[130] Paul Martin, Malcolm P. Atkinson, Mark Parsons, Adam Carter, and Gareth Francis, "EDIM1 Progress Report," tech. rep., University of Edinburgh, December 2011. [pp. 124.]

[131] Paul Martin and Gagarine Yaikhom, "Definition of the DISPEL language," in Atkinson *et al.* [10]. [pp. 101 and 104.]

[132] Timothy M. McPhillips and Shawn Bowers, "An approach for pipelining nested collections in scientific workflows," *SIGMOD Record*, 34(3):12–17, September 2005. [pp. 52 and 58.]

[133] William Michener, James Beach, Shawn Bowers, Laura Downey, Matthew Jones, Bertram Ludäscher, Deana Pennington, Arcot Rajasekar, Samantha Romanello, Mark Schildhauer, Dave Vieglais, and Jianting Zhang, "Data integration and workflow solutions for ecology," in *Data Integration in the Life Sciences* (Bertram Ludäscher and Louiqa Raschid, eds.), vol. 3615 of *Lecture Notes in Computer Science*, pp. 734–734, Springer Berlin / Heidelberg, 2005. [pp. 40.]

[134] Paolo Missier, Stian Soiland-Reyes, Stuart Owen, Wei Tan, Alexandra Nenadic, Ian Dunlop, Alan Williams, Tom Oinn, and Carole Goble, "Taverna, reloaded," in *Scientific and Statistical Database Management* (Michael Gertz and Bertram Ludäscher, eds.), vol. 6187 of *Lecture Notes in Computer Science*, pp. 471–481, Springer Berlin / Heidelberg, June 2010. [pp. 46, 52, 57, and 58.]

[135] J. Ian Munro and Mike Paterson, "Selection and sorting with limited storage," in *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, SFCS '78, pp. 253–258, October 1978. [pp. 62.]

[136] S. Muthukrishnan, "Data streams: Algorithms and applications," *Foundations and Trends in Theoretical Computer Science*, 1(2):117–236, 2005. [pp. 62.]

[137] Michael L. Norman and Allan Snavely, "Accelerating data-intensive science with Gordon and Dash," in *Proceedings of the 2010 TeraGrid Conference*, TG '10, pp. 14:1–14:7, ACM, August 2010. [pp. 19.]

[138] Thomas Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew Pocock, Anil Wipat, and Peter Li, "Taverna: a tool for the composition and enactment of bioinformatics workflows.," *Bioinformatics*, 20(17):3045–3054, November 2004. [pp. 1, 46, and 100.]

[139] Tom Oinn, Mark Greenwood, Matthew Addis, M. Nedim Alpdemir, Justin Ferris, Kevin Glover, Carole Goble, Antoon Goderis, Duncan Hull, Darren Marvin, Peter Li, Phillip Lord, Matthew R. Pocock, Martin Senger, Robert Stevens, Anil Wipat, and Chris Wroe, "Taverna: lessons in creating a workflow environment for the life sciences," *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, 2006. [pp. 44.]

[140] Tom Oinn, Peter Li, Douglas B. Kell, Carole Goble, Antoon Goderis, Mark Greenwood, Duncan Hull, Robert Stevens, Daniele Turi, and Jun Zhao, "Taverna/^{my}Grid: Aligning a Workflow System with the Life Sciences Community," in Taylor *et al.* [164], pp. 300–319. [pp. 45.]

[141] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, 26(1):80–113, 2007. [pp. 82.]

[142] Cesare Pautasso and Gustavo Alonso, "Parallel computing patterns for grid workflows," in *Proceedings of the workshop on Workflows in support of large-scale science*, WORKS '06, pp. 1–10, June 2006. [pp. 58, 62, 64, and 65.]

[143] Beth Plale, Dennis Gannon, Jerry Brotzge, Kelvin Droegemeier, Jim Kurose, David McLaughlin, Robert Wilhelmson, Sara Graves, Mohan Ramamurthy, Richard D. Clark, Sepi Yalda, Daniel A. Reed, Everette Joseph, and V. Chandrasekar, "CASA and LEAD: Adaptive cyberinfrastructure for real-time multiscale weather forecasting," *Computer*, 39(11):56–64, November 2006. [pp. 2 and 4.]

[144] Ioan Raicu, Yong Zhao, Catalin Dumitrescu, Ian Foster, and Mike Wilde, "Falkon: a Fast and Light-weight tasK executiON framework," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pp. 43:1–43:12, ACM, 2007. [pp. 20 and 39.]

[145] David De Roure, Carole Goble, Sergejs Aleksejevs, Sean Bechhofer, Jiten Bhagat, Don Cruickshank, Paul Fisher, Nandkumar Kollara, Danius Michaelides, Paolo Missier, David Newman, Marcus Ramsden, Marco Roos, Katy Wolstencroft, Ed Zaluska, and Jun Zhao, "The Evolution of ^{my}Experiment," in *Proceedings of the Sixth IEEE International Conference on e-Science*, e-Science '10, pp. 153–160, IEEE, December 2010. [pp. 50.]

[146] David De Roure, Carole Goble, and Robert Stevens, "The design and realisation of the ^{my}Experiment Virtual Research Environment for social sharing of workflows," *Future Generation Computer Systems*, 25(5):561–567, 2009. [pp. 24 and 44.]

[147] Nick Russell and Arthur ter Hofstede, "newYAWL: Towards Workflow 2.0," in *Transactions on Petri Nets and Other Models of Concurrency II* (Kurt Jensen and Wil van der Aalst, eds.), vol. 5460 of *Lecture Notes in Computer Science*, pp. 79–97, Springer Berlin / Heidelberg, 2009. [pp. 1.]

[148] Nicole Schweikardt, "One-pass algorithm," in Liu and Özsu [121], pp. 1948–1949. [pp. 62.]

[149] Igor Sfiligoi, "Making science in the grid world: using glideins to maximize scientific output," in *Proceedings of the IEEE Nuclear Science Symposium Conference Record*, vol. 2 of *NSS '07*, pp. 1107–1109, October-November 2007. [pp. 39, 56, and 156.]

[150] Matthew Shields, "Control- versus data-driven workflows," in Taylor *et al.* [164], pp. 167–173. [pp. 21.]

[151] Yogesh L. Simmhan, Roger Barga, Catharine van Ingen, Ed Lazowska, and Alex Szalay, "Building the Trident scientific workflow workbench for data management in the cloud," in *Proceedings of the Third International Conference on Advanced Engineering Computing and Applications in Sciences*, ADVCOMP '09, pp. 41–50, IEEE Computer Society, October 2009. [pp. 3 and 28.]

[152] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon, "A survey of data provenance in e-Science," *SIGMOD Record*, 34(3):31–36, September 2005. [pp. 25.]

[153] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon, "Karma2: Provenance management for data-driven workflows," *International Journal Of Web Services Research*, 5(2):1–22, April-June 2008. [pp. 25.]

[154] Gurmeet Singh, Carl Kesselman, and Ewa Deelman, "Optimizing grid-based workflow execution," *Journal of Grid Computing*, 3(3-4):201–219, September 2005. [pp. 5 and 24.]

[155] Aleksander Slominski, "Adapting BPEL to Scientific Workflows," in Taylor *et al.* [164], pp. 208–226. [pp. 1 and 31.]

[156] Robin J. Smith, "The Oracle platform for real time streaming event driven architecture based solutions," in *Proceedings of the ACM SIGSPATIAL International Workshop on GeoStreaming*, IWGS '10, pp. 3–3, ACM, 2010. [pp. 61.]

[157] Tiberiu Stef-Praun, Benjamin Clifford, Ian Foster, Uri Hasson, Mihael Hategan, Steven L. Small, Michael Wilde, and Yong Zhao, "Accelerating Medical Research Using the Swift Workflow System," *Studies in Health Technology and Informatics*, 126:207–216, 2007. [pp. 37.]

[158] Robert Stephens, "A survey of stream processing," *Acta Informatica*, 34(7):491–541, July 1997. [pp. 60 and 61.]

[159] Robert Stevens, Jun Zhao, and Carole Goble, "Using provenance to manage knowledge of in silico experiments," *Briefings in Bioinformatics*, 8(3):183–194, 2007. [pp. 25.]

[160] Michael Stonebraker, Jacek Becla, David J. DeWitt, Kian-Tat Lim, David Maier, Oliver Ratzesberger, and Stanley B. Zdonik, "Requirements for Science Data Bases and SciDB," in *CIDR*, 2009. [pp. 2 and 17.]

[161] Alexander S. Szalay, Gordon Bell, Jan vandenBerg, Alainna Wonders, Randal C. Burns, Dan Fay, Jim Heasley, Tony Hey, María A. Nieto-Santisteban, Ani Thakar, Catharine van Ingen, and Richard Wilton, "GrayWulf: Scalable Clustered Architecture for Data Intensive Computing," in *Proceedings of the 42nd Hawaii International Conference on System Sciences*, HICSS-42, pp. 1–10, January 2009. [pp. 19.]

[162] Alexander S. Szalay and José A. Blakeley, "Gray's Laws: Database-centric Computing in Science," in Hey *et al.* [99], pp. 5–11. [pp. 114 and 124.]

[163] Ian Taylor, Matthew Shields, Ian Wang, and Andrew Harrison, "The Triana workflow environment: Architecture and applications," in Taylor *et al.* [164], pp. 320–339. [pp. 24 and 25.]

[164] Ian J. Taylor, Ewa Deelman, Dennis B. Gannon, and Matthew Shields, *Workflows for e-Science: Scientific Workflows for Grids*. Springer London, 2007. [pp. 25, 170, 171, 172, 173, 174, 175, 176, 178, 179, 180, 181, 182, and 183.]

[165] Douglas Thain, Todd Tannenbaum, and Miron Livny, "Distributed computing in practice: the Condor experience," *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005. [pp. 19 and 24.]

[166] William Thies, Michal Karczmarek, and Saman Amarasinghe, "StreamIt: A Language for Streaming Applications," in *Compiler Construction* (R. Horspool, ed.), vol. 2304 of *Lecture Notes in Computer Science*, pp. 49–84, Springer Berlin / Heidelberg, 2002. [pp. 61 and 62.]

[167] Haluk Topcuoglu, Salim Hariri, and Min-You Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, March 2002. [pp. 72.]

[168] Hong-Linh Truong, Schahram Dustdar, and Thomas Fahringer, "Performance metrics and ontologies for grid workflows," *Future Generation Computer Systems*, 23(6):760–772, 2007. [pp. 109.]

[169] Thomas D. Uram, Michael E. Papka, Mark Hereld, and Michael Wilde, "A solution looking for lots of problems: generic portals for science infrastructure," in *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*, TG '11, pp. 44:1–44:7, ACM, 2011. [pp. 40.]

[170] W.M.P. van der Aalst and A.H.M. ter Hofstede, "YAWL: yet another workflow language," *Information Systems*, 30(4):245–275, 2005. [pp. 1.]

[171] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros, "Workflow patterns," *Distributed and Parallel Databases*, 14(1):5–51, July 2003. [pp. 22 and 29.]

[172] Jens Vöckler, Gaurang Mehta, Yong Zhao, Ewa Deelman, and Michael Wilde, "Kickstarting remote applications," in *Second International Workshop on Grid Computing Environments*, 2006. [pp. 35.]

[173] Gregor von Laszewski and Mike Hategan, "Workflow Concepts of the Java CoG Kit," *Journal of Grid Computing*, 3(3-4):239–258, September 2005. [pp. 37.]

[174] David W. Walker, Lican Huang, Omer F. Rana, and Yan Huang, "Dynamic service selection in workflows using performance data," *Scientific Programming*, 15(4):235–247, 2007. [pp. 58.]

[175] Hongbing Wang, Joshua Zhexue Huang, Yuzhong Qu, and Junyuan Xie, "Web services: problems and future directions," *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(3):309–320, April 2004. [pp. 31.]

[176] Marek Wieczorek, Andreas Hoheisel, and Radu Prodan, "Towards a general model of the multi-criteria workflow scheduling on the grid," *Future Generation Computer Systems*, 25(3):237–256, March 2009. [pp. 29.]

[177] Michael Wilde, Ian Foster, Kamil Iskra, Pete Beckman, Zhao Zhang, Allan Espinosa, Mihael Hategan, Ben Clifford, and Ioan Raicu, "Parallel scripting for applications at the petascale and beyond," *Computer*, 42(11):50 –60, November 2009. [pp. 20.]

[178] Michael Wilde, Mihael Hategan, Justin M. Wozniak, Ben Clifford, Daniel S. Katz, and Ian Foster, "Swift: A language for distributed parallel scripting," *Parallel Computing*, 37(9):633–652, September 2011. [pp. 1, 38, 39, and 100.]

[179] Matthew Woitaszek, John M. Dennis, and Taleena R. Sine, "Parallel High-resolution Climate Data Analysis using Swift," in *Proceedings of the 2011 ACM international workshop on Many task computing on grids and supercomputers*, MTAGS '11, pp. 5–14, ACM, November 2011. [pp. 37.]

[180] Alex Wright, "Data streaming 2.0," *Communications of the ACM*, 53(4):13–14, April 2010. [pp. 62.]

[181] Wenjun Wu, Thomas Uram, Michael Wilde, Mark Hereld, and Michael E. Papka, "Accelerating science gateway development with Web 2.0 and Swift," in *Proceedings of the 2010 TeraGrid Conference*, TG '10, pp. 23:1–23:7, ACM, August 2010. [pp. 40.]

[182] Gagarine Yaikhom, Malcolm P. Atkinson, Jano I. van Hemert, Oscar Corcho, and Amy Krause, "Validation and mismatch repair of workflows through typed data streams," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 369(1949):3285–3299, August 2011. [pp. 107 and 109.]

[183] Gagarine Yaikhom, Chee Sun Liew, Liangxiu Han, Jano I. van Hemert, Malcolm P. Atkinson, and Amrey Krause, "Federated enactment of workflow patterns," in *Euro-Par 2010 - Parallel Processing* (Pasqua D'Ambra, Mario Guarracino, and Domenico Talia, eds.), vol. 6271 of *Lecture Notes in Computer Science*, pp. 317–328, Springer Berlin / Heidelberg, 2010. [pp. 7.]

[184] Jia Yu and Rajkumar Buyya, "A taxonomy of workflow management systems for grid computing," *Journal of Grid Computing*, 3(3-4):171–200, September 2005. [pp. 1 and 29.]

[185] Yong Zhao, Mihael Hategan, Ben Clifford, Ian Foster, Gregor von Laszewski, Veronika Nefedova, Ioan Raicu, Tiberiu Stef-Praun, and Michael Wilde, "Swift: Fast, reliable, loosely coupled parallel computation," in *Proceedings of the 2007 IEEE Congress on Services*, SERVICES '07, pp. 199–206, IEEE Computer Society, July 2007. [pp. 25, 37, and 52.]

[186] Yong Zhao, Ioan Raicu, and Ian Foster, "Scientific workflow systems for 21st century, new bottle or new wine?," in *Proceedings of the 2008 IEEE Congress on Services - Part I*, SERVICES '08, pp. 467–471, IEEE Computer Society, July 2008. [pp. 2 and 25.]

[187] Daniel Zinn, Quinn Hart, Timothy McPhillips, Bertram Ludäscher, Yogesh Simmhan, Michail Giakkoupis, and Viktor K. Prasanna, "Towards reliable, performant workflows for streaming-applications on cloud platforms," in *Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID '11, pp. 235–244, May 2011. [pp. 52.]