UNIVERSITY OF AMSTERDAM

Dynamic Enactment of Scientific Workflows using Pilot-Abstractions

A thesis submitted for the degree of

Master of Science (MSc)

at the

Informatics Institute
Faculty of Science
University of Amsterdam

Mark Alexander Santcroos

October 2016

Supervised by

dr. Adam Belloum

Abstract

On a high level, scientific workflow enactment has to deal with the consecutive execution of computational tasks. As these tasks often require input and produce output data, the enactment of these workflows also involves the transfer of input and output data of these tasks to and from the resource where these tasks are executed. On Distributed Computing Infrastructure (DCI) like the Open Science Grid (OSG), which is inherently heterogeneous, the complexity and dynamism of data and processing distribution have increased. The mapping of logical workflow tasks to physical resources of the DCI and the subsequent transfer of data to and from these sources exhibit a large degree of freedom. We argue that the management of dynamic data and compute should become part of the runtime system of workflow engines to enable workflows to scale as necessary to address big data challenges and fully exploit the capabilities of DCI. The P* model for pilot-abstractions defines a clear separation between the logical compute and data units and their realization as a job or a file at a physical resource. In this thesis we describe the implementation of Pilot-Data, an extension of RADICAL-Pilot, that satisfies the data aspects of the P* model for RADICAL-Pilot. To explore both functionally and experimentally whether this Pilot-Data implementation can provide the capabilities for such a workflow system runtime environment, we also implemented Marvin, a workflow engine for the GWENDIA workflow language that interfaces to both the compute and data capabilities of RADICAL-Pilot. For the empirical evaluation we use various synthetic transfers and workloads and a real life case study: a DNA sequencing analysis workflow. We conclude that the pilot abstraction offers a valid approach to explore the design of a new generation of workflow management systems and runtime environments that are capable of intelligently deciding on application-aware late binding of compute tasks and data to physical resources.

Acknowledgements

The start of my study at the University of Amsterdam turned out to be also the beginning of a new career. My initial course was with Adam Belloum who not only enthusiastically lectured about distributed computing, but also brought me in contact with Silvia Delgado Olabarriaga. Adam and I ran into each other multiple times over the last years, of course in the context of study, but as well as collaborators and peers. Adam, thanks for your role in bringing my study to a good end in the form of my graduation research, and I hope we will cross paths again in the future!
Silvia and I worked together for many years at the AMC where we did interesting projects and had much fun in the process. Silvia, thank you for your guidance and I would like to express my gratitude that you are now also part of the committee, I'm sure we will meet again!
Paola Grosso, thank you for our encounters during the course of my study and your willingness to take part in my defence committee.
Silvia on her turn was instrumental in the adventure that lead me to the USA to work with Shantenu Jha. Shantenu: one thesis down, one to go, thanks!
As you can read in Section 1.3, this work was done in close collaboration with many excellent researchers and dear colleagues, with a special mention of Shayan Shahand and Barbera van Schaik at the AMC, and Andre Merzky at Rutgers.
A 'Bedankt voor alles!' to my mother who although has given up trying to understand what I'm doing, didn't give up showing interest and motivating me.
Not many theses are written without sacrifices in the personal sphere, and mine was no exception. Kalinka, thank you for continued support and patience during this sometimes turbulent period for both of us, love you!
Finally, a High-Five to Alec!

Contents

List of Figures

List of Tables

Abbreviations

API Application Programming Interface.

BWA Burrows-Wheeler Aligner.

CE Compute Element.

DCI Distributed Computing Infrastructure.

HPC High Performance Computing.

HTC High Throughput Computing.

LHC Large Hydron Collider.

MPI Message Passing Interface.

OGF Open Grid Forum.

OSG Open Science Grid.

RP RADICAL-Pilot.

SAGA Simple API for Grid Applications.

SE Storage Element.

SRM Storage Resource Manager.

TTC Time to Completion.

VO Virtual Organization.

Chapter 1

Introduction

A wide range of Biomedical applications have been successfully ported and executed on Distributed Computing Infrastructure (DCI) using workflow technology. Workflow management systems can hide details of the underlying infrastructure, and serve as an excellent abstraction to carry out high throughput experiments [1]. By lowering the barriers to use such complex infrastructures, workflow management systems have been valuable allies in the realization of the e-Science vision [2].

At the Academic Medical Center of the University of Amsterdam (AMC), a workflow-based software platform has been adopted for many years to enable medical imaging [3] and DNA sequencing [4] research. Workflows are extensively used as the primary abstraction for programming and running applications on the Dutch production grid infrastructure, facilitating access to both advanced and novice users. With this approach, data processing "pipelines" can be easily described into grid workflows, and high throughput performance can be achieved by splitting the datasets and distributing their processing on the DCI. Running computations on the DCI has become a trivial exercise on this platform.

With the growth of the data volumes, the solution provided by the platform turned out to be insufficient to address the increasing complexity and dynamism of data and processing distribution [4]. The main challenges shifted from the processing to the data, and the alignment of the two became more important. Much more than before, it is now necessary to optimize the mapping of logical tasks to physical resources to maintain high throughput. This includes, for example, workload balancing to avoid bottlenecks, but also co-locating tasks together to minimize data transfers. Furthermore we know from experience that one size does not fit all, and that one approach can be an optimization for one application and a pessimization for the other.

Ideally the location of data and processing should take into account the dynamic availability of resources and the data flow requirements derived from a given workflow execution. In practice we have seen that such optimization is hard to achieve using workflow abstractions as we know today. Optimization attempts often found their way into the workflow descriptions, for example, by early binding a given computation or dataset to resources that were known in advance to have sufficient capacity. In this way the workflow descriptions became 'polluted' with all types of DCI-specific information, and their execution became limited to a subset of the resource available at runtime. Users (the workflow developer or the workflow executors) became responsible for the optimizations that the workflow management system was unable to do.

Although our hands-on experience is limited to a couple of workflow management systems, we argue that this is a fundamental characteristic in most workflow management systems today due to (a) the lack of an explicit approach to handle distributed data in a workflow and (b) the lack of a proper abstraction to separate logical tasks and data flow from their mapping into physical location on a DCI. While (b) has been partially addressed by using a pilot job framework as back-end for workflow systems [5], to our knowledge handling data distribution has not been properly addressed yet in the context of workflow systems. Based on our observations from backstage of various workflow systems, we realize that implementing our vision of the 'ideal case' is very complex and requires some out-of-the-box thinking and looking at fresh alternatives.

This thesis explores the P* model for pilot-abstractions [6], which proposed a clear separation between the logical compute and data units and their realization as jobs or files in some physical resources. This model is accompanied by an Application Programming Interface (API) – the Pilot-API, which provides an interface to Pilot-Job frameworks that adhere to the P* model, and which supports programming of distributed applications that can implement complex and dynamic scheduling of resources. We believe that this API exposes powerful features to address (a) and (b), forming an interesting basis to explore for the construction of a new generation of workflow management systems that are more capable of intelligently deciding on application-aware late binding to physical resources.

In this thesis we work with a real use case and the corresponding reported challenges [7]. Modern DNA sequencing machines produce data in the range of 1-100 GB per experiment and with ongoing technological developments this amount is rapidly increasing. The majority of experiments involve re-sequencing of human genomes and exomes to find genomic regions that are associated with disease. There are many sequence analysis tools freely available, e.g. for sequence alignment, quality control and variant detection, and frequently new tools are developed to address new biological questions. The group is using workflow technology (MOTEUR [8], GWENDIA [9]) to allow easy incorporation of such software in the data analysis pipelines, as well as to leverage grid infrastructures for the analysis of large datasets in parallel. The size of the datasets had grown from 1 GB to 70 GB in 3 years, therefore adjustments were needed to optimize these workflows. Procedures have been implemented for faster data transfer to and from grid resources, and for fault recovery at run time. A split-and-merge procedure for a frequently used sequence alignment tool, Burrows-Wheeler Aligner (BWA), resulted in a three-fold reduction of the total time needed to complete an experiment and increased efficiency by a reduction in number of failures. The success rate was increased from 10% to

70%.

1.1 Research Questions

Q1 *Can the semantics of a GWENDIA workflow be expressed using the Pilot-API?*

Q2 *What factors of data and compute placement exist that can be exploited to improve execution of data intensive workflows?*

Q3 *Can the decision making about compute and data placement be automated in such a way that it has an impact on data intensive workflow execution?*

1.2 Structure

The remainder of this thesis is organized as follows. In Chapter 2 related work for all the topics discussed so far is put into context. Chapter 3 presents the conceptual foundations on which this thesis is constructed. That is followed in Chapter 4 by a description of the implementation of the software used for the conducted experiments. The experiments and target infrastructure to verify the concepts and implementation are laid out in Chapter 5, followed by the results of the experiments in Chapter 6 and completed with a discussion of these results in Chapter 7. In Chapter 8 we close the loop between the research questions and the acquired results and end with a future outlook.

1.3 Contributions

This thesis is the aggregate of both published and (still) unpublished work. The following papers are used as material for this thesis.

The P* model

- A Luckow, Mark Santcroos, A Merzky, O Weidner, P Mantha, and S Jha. P∗: A model of pilot-abstractions. In *E-Science (e-Science), 2012 IEEE 8th International Conference on*, pages 1–10, 2012
- André Luckow, Mark Santcroos, Ole Weidner, Andre Merzky, Sharath Maddineni, and Shantenu Jha. Towards a common model for pilot-jobs. In *HPDC '12: Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing.* ACM, June 2012

RADICAL-Pilot

- Andre Merzky, Mark Santcroos, Matteo Turilli, and Shantenu Jha. Executing Dynamic and Heterogeneous Workloads on Super Computers, 2016. (under review) `http://arxiv.org/abs/1512.08194`

- Mark Santcroos, Ralph Castain, Andre Merzky, Iain Bethune, and Shantenu Jha. Executing dynamic heterogeneous workloads on blue waters with radical-pilot. In *Cray User Group 2016*, 2016

DNA Sequencing Analysis on DCIs

- BDC van Schaik, Mark Santcroos, and V Korkhov. Challenges in DNA sequence analysis on a production grid. In *EGI Community Forum 2012*, 2012

- BDC van Schaik, Mark Santcroos, S Madougou, A Jongejan, A H C van Kampen, and S.D Olabarriaga. e-Bioscience Solutions and Challenges for Next Generation Sequencing Experiments. In *2nd International Work-Conference on Bioinformatics and Biomedical Engineering*, pages 333–334, 2013

Pilot-Data

- A Luckow, Mark Santcroos, A Zebrowski, and S Jha. Pilot-data: an abstraction for distributed data. *Journal of Parallel and Distributed Computing*, 79-80:16–30, 2015

- Mark Santcroos, Barbera DC van Schaik, Shayan Shahand, Sílvia Delgado Olabarriaga, Andre Luckow, and Shantenu Jha. Exploring Dynamic Enactment of Scientific Workflows using Pilot-Abstractions. In *13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 1–9, Delft

- Mark Santcroos, S Delgado Olabarriaga, D S Katz, and S Jha. Pilot abstractions for compute, data, and network. In *E-Science (e-Science), 2012 IEEE 8th International Conference on*, pages 1–2, 2012

Unpublished

Materials in this thesis not yet published are:

- The software implementations described in Chapter 4.

- The experiments and results in Chapter 5 and Chapter 6 respectively.

- The discussion and conclusion in Chapter 7 and Chapter 8 that are based on the experiments.

Chapter 2

Related Work

History of Pilot-Jobs Around twenty systems with pilot capabilities have been implemented since 1995 [17]. AppLeS [18] is one of the first published implementations of both placeholders for resources and application-level scheduling; HTCondor [19] and Glidein [20] enabled pilot-based resource allocation and execution to OSG; DIANE [21], AliEn [22], DIRAC [23], PanDA [24], and GlideinWMS [25] brought pilot-based workload execution to the Large Hydron Collider (LHC) and other grid communities.

In contrast to RADICAL-Pilot, the aforementioned systems are often tailored for specific workloads, resources (particularly High Throughput Computing (HTC)), interfaces, or development models. They often encapsulate pilot capabilities within monolithic tools with greater functional scope. For example, HTCondor with Glidein on the Open Science Grid (OSG) [26] is one of the most widely used Pilot systems but serves mostly single core workloads. The Pilot systems developed for the LHC communities execute millions of jobs a week [24] but specialize on supporting LHC workloads and, in most cases, specific resources like those of Worldwide LHC Computing Grid (WLCG).

Pilots on HPC Falkon is a Pilot system for High Performance Computing (HPC) systems. Just as RADICAL-Pilot, Falkon exposes an API that is used to develop distributed applications or to be integrated within an end-to-end system such as Swift and it has been designed to implement concurrency at multiple levels including dispatching, scheduling, and spawning of tasks across multiple compute nodes of possibly multiple resources. Falkon is designed for single core applications. Coasters is similar to RADICAL-Pilot in that it supports heterogeneity at resource level. RADICAL-Pilot supports a greater variety of resources though, mainly due to the use of Simple API for Grid Applications (SAGA) as its resource interoperability layer. The two systems differ in their architectures and workload heterogeneity (RADICAL-Pilot also supports multi-node Message Passing Interface (MPI) applications). RADICAL-Pilot's modular and extensible architecture was demonstrated by also supporting the Cray architecture on Blue Waters and Titan [12].

Recognizing the potential for HTC on HPC resources, IBM developed an HTC mode resembling a Pilot system [27] for the series of IBM BG/L products. Unsupported by later IBM Blue Gene series, RADICAL-Pilot brings back this HTC capability generalizing it to HPC architectures beyond IBM BG/L machines, like the BG/Q. Nitro [28], is a high-throughput scheduling solution for HPC systems that works in collaboration with the Moab scheduler for TORQUE. Instead of requiring individual job scheduling, Nitro enables high-speed throughput on short computing jobs by allowing the scheduler to incur the scheduling overhead only once for a large batch of jobs.

MySGE [29] allows users to create a private instance of a Sun GridEngine cluster on large parallel systems like Hopper and Edison. Once the cluster is started, users can submit serial jobs, array jobs, and other throughput oriented workloads into the personal SGE scheduler. The jobs are then run within the user's private cluster.

QDO [30] is a lightweight high-throughput queuing system for workflows that have many small tasks to perform. It is designed for situations where the number of tasks to perform is much larger than the practical limits of the underlying batch job system. Its interface emphasizes simplicity while maintaining flexibility.

Distributed Data Management Systems Managing distributed data and compute is an ongoing research theme. For grid environments for example, the Stork [31] data-aware batch scheduler provides advanced data and compute placement for HTCondor and DAGMan. Stork supports multiple transfer protocols like, Storage Resource Manager (SRM), (Grid)FTP, HTTP and SRB. Romosan et al. [32] present another data-compute co-scheduling approach based on HTCondor and SRM. Both approaches are built on top of existing job scheduling and data-transfer and storage solutions. Further frameworks for other distributed environments have been proposed. For example, FRIEDA [33] provides a data management framework for cloud-environments.

Various research on when to distribute and replicate data has been conducted: for example, Foster [34] and Bell [35] investigate algorithms for data replication management system and dynamic replication in the context of scientific data grids. A limitation of said approaches is that the systems and algorithms are usually constrained to system-level replication, making it difficult for the user to control replication on application-level and employ dynamic replication strategies. Glatard et al. [36] provide a classification of data placement and replications algorithms and systems for distributed computing environments.

Programming models Various abstractions for optimizing access and management of distributed data have been proposed: Filecule [37] is an abstraction that groups a set of files that are often used together, allowing an efficient management of data using bulk operations. This includes the scheduling of data transfers and/or replications. Similar file grouping mechanisms have been proposed by Amer et al. [38], Ganger et al. [39] and BitDew [40]. Further several higher-level, less resource-oriented abstractions for enabling data analysis on large volumes of data have been proposed. A well-known example is the MapReduce programming model [41] for which various implementations exist [42, 43]. Another example is DataCutter [44], a framework that enables exploration and querying of large datasets while minimizing the necessary data movements. While various abstractions for data-intensive applications exist, these are typically bound to a specific infrastructure. For example, Hadoop – the most-widely used MapReduce implementation – intermingles resource management, programming abstraction in a monolithic solution sacrificing flexibility and extensibility with respect to other kinds of data-intensive workloads.

Pilot-Jobs and Data Management Pilot-Jobs have been successful abstractions in distributed computing as evidenced by a plethora of Pilot-Job frameworks. With the increasing importance of data, Pilot-Jobs have been also used to process and analyze large data. However, in most Pilot-Job framework the support for data movement and placement is insufficient [45]. Only a few of them provide integrated compute/data capabilities, and where they exist, they are often non-extensible and bound to a particular infrastructure. In general, one can distinguish two kinds of data management: (i) the ability to stage-in/stage-out files from another compute node or a storage backend, such as SRM and (ii) the provisioning of integrated data/compute management mechanisms. An example for (i) is HTCondor-G/Glide-in, which provides a basic mechanism for file staging and also supports access to SRM. Another example is Swift [46], which provides a data management component called Collective Data Management (CDM). DIANE provides in-band data transfer functionality over its CORBA channel. In the context of the LHC Grid several type (ii) Pilot-Job frameworks that support access to the vast amounts of experimental data created by the Large Hadron Collider have been developed. DIRAC [47] is an example of such a system. It interfaces to SRM storage resources and enables the application to stage-in/out data to this system. AliEn [48] also provides the ability to tightly integrate storage and compute resources and is also able to manage file replicas. While all data can be accessed from anywhere, the scheduler is aware of data localities and attempts to schedule compute close to the data. Similarly, PanDA [49] provides support for the retrieval of data from the XRootD storage infrastructure. The PanDA Dynamic Data Placement component [50] provides a demand-based replication system, which can replicate popular datasets to underutilized resources for later computations. However, this capability is provided on system-level and constrained to official Atlas datasets, i. e. it cannot be applied to user-level datasets. The data/compute management capabilities of AliEn and PanDA are built on top of HTCondor-G/Glide-in. In addition to this strong coupling to the underlying infrastructure, these frameworks are tightly bound to their specific applications. Another example for a type (ii) system is Falkon [51], which provides a data-aware scheduler on top of a pool of dynamically acquired compute and data resources [52]. The so called data diffusion mechanism automatically caches data on Pilot-level enabling the efficient re-use of data. Falkon provides limited interoperability and is constrained to Globus-based HPC environments.

Workflow Systems Pilots and pilot-like capabilities are also implemented or used by various workflow management systems. Pegasus [53] uses Glidein via providers like Corral [54]; Makeflow [55] and FireWorks [56] enable users to manually start workers on HPC resources via master/worker tools called Work Queue [57] and LaunchPad [56]; and Swift [58] uses two Pilot systems called Falkon [59] and Coasters [60]. In these systems, the pilot is not always a stand-alone capability and in those cases any innovations and advances of the pilot capability are thus confined to the encasing system. Pegasus-MPI-Cluster (PMC) [61] is an MPI-based Master/Worker framework that can be used in combination with Pegasus. In the same spirit as RADICAL-Pilot, this enables Pegasus to run large-scale workflows of small tasks on HPC resources. In contrast with RADICAL-Pilot, tasks are limited to single node execution. In addition there is a dependency on $fork()/exec()$ on the compute node which rules out PMC on some HPC resources. WS-VLAM [62], a Service Oriented Architecture (SOA) re-implementation of VLAM-G, is a data stream based workflow system, with considering the 'Human in the loop' as one of the defining properties. Workflow management systems such as MOTEUR [8] threat data as first class citizen semantically but do not provide any performance optimization capabilities. The overheads involved in accessing distributed resources can lead to poor performance that a workflow system is not able to mitigate. Resource provisioning techniques such as advance reservations, multi-level scheduling, and infrastructure as a service (IaaS) may be used to reduce these overheads. Advantages and disadvantages of such technique are explained in [63]. For example, Juve et al. showed that a resource provisioning system based on multi-level scheduling called Corral could improve workflow runtime by reducing scheduling overheads [64]. Similarly, Singh and Deelman [65] showed that the completion time of scientific workflows could be reduced by 50% by means of task clustering and resource provisioning using advance reservations based on statistics or dynamic provisioning mechanisms. Both of these examples use Pegasus WfMS [66] in combination with HTCondor Glidein [67] for resource provisioning, which is a mechanism to add one or more remote grid resources to a local HTCondor resource pool temporarily. It uses the same method as typically used by Pilot-Job frameworks, which is to submit a setup task that creates daemons on a remote grid resource. Once the daemons are created and started, they contact the local pool to fetch and run jobs. HTCondor's matchmaking mechanism is used to map jobs to resources, however, no direct control over the placement of the pilots is exposed to the user.

Chapter 3

Foundations

3.1 P*: Pilot Abstraction for jobs and data

DCIs are by definition comprised of a set of resources that are fluctuating – growing, shrinking, changing in load and capability, in contrast to a static resource utilization model of traditional parallel and cluster computing systems. The ability to utilize a dynamic resource pool is thus an important attribute of any application that needs to utilize DCIs effectively and efficiently.

Pilot-Jobs offer a simple approach for decoupling workload management and resource assignment/scheduling, providing an effective abstraction for dynamic execution and resource utilization. In essence, a Pilot-Job is a placeholder job serving as a container for a set of compute tasks. Not surprisingly, Pilot-Jobs have been very successful abstractions in distributed computing because they liberate applications and or users from the challenging requirement of mapping specific tasks onto explicit heterogeneous and dynamic resource pools. Pilot-Jobs thus shield applications from having to load-balance tasks across such resources.

The Pilot-Job abstraction is also a promising route to address specific requirements of distributed scientific applications, such as coupled-execution and application-level scheduling.

The P* model introduced in [10] and further described in [6] provides a unified model for describing and analyzing common elements of Pilot-Job implementations. The P* approach to pilots has the following natural advantages: (i) it permits late binding of workloads to resources and (ii) the decoupling of tasks from resource management can be extended to data. The extension of the P* model with Pilot-Data is described explored in detail in [14].

In the extended model two fundamental abstractions are defined: Pilot-Compute and Pilot-Data. The abstraction of a Pilot-Compute (PC) generalizes the reoccurring concept of utilizing a placeholder job as a container for a set of compute tasks or Compute-Units (CU). Instances of that placeholder job are commonly referred to as Pilot-Jobs or pilots. Analogous to Pilot-Compute,

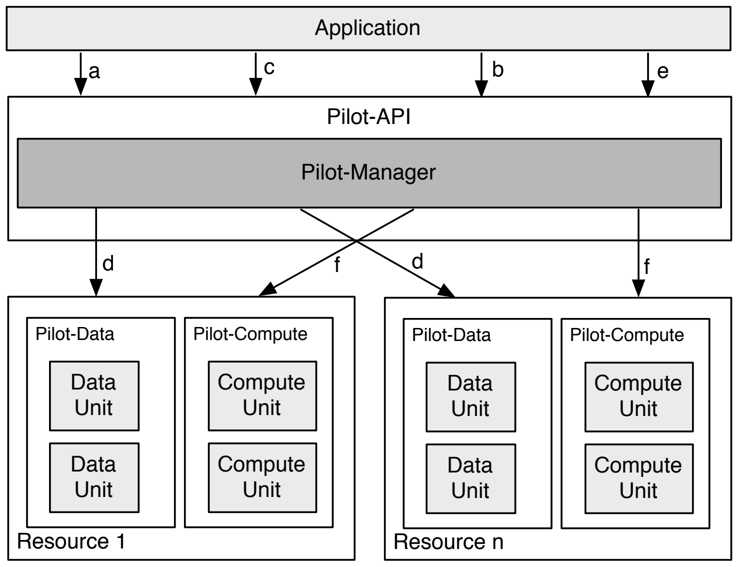

Figure 3.1: P* Architecture: The application allocates Pilot-Compute (a) and Pilot-Data (b) resources through the Pilot-API. The application also describes the Data-Units and passes these to the Pilot-Manager (c). The Pilot-Manager is responsible for transferring (d) the Data-Units to their physical locations (Pilot-Data). When the data is in place, the Application can submit Compute-Units (e) that will run in Pilot-Computes (f).

the Pilot-Data (PD) abstraction has been introduced to provide a placeholder for data as a container for a set of application-level logical Data-Units (DU), separately from their physical allocation. A Compute-Unit represents a self-containing piece of the processing to be carried out (e.g. a workflow task), while a Data-Unit represents user data (e.g., input or output files for a task). The Pilot-Compute and Pilot-Data abstractions enable application level or user level control and management of the set of allocated resources, with late binding of Compute-Unit and Data-Unit to pilots. Figure 3.1 shows the architecture of the model.

3.2 RADICAL-Pilot

RADICAL-Pilot is a scalable and interoperable pilot system that implements the Pilot abstraction to support the execution of diverse workloads. We describe the design and architecture (see Figure 3.2) and characterize the performance of RADICAL-Pilot's task execution components, which are engineered for efficient resource utilization while maintaining the full generality of the Pilot abstraction. RADICAL-Pilot is supported on Crays such as Blue Waters (NCSA), Titan (ORNL), Hopper & Edison (NERSC) and ARCHER (EPSRC), but also on

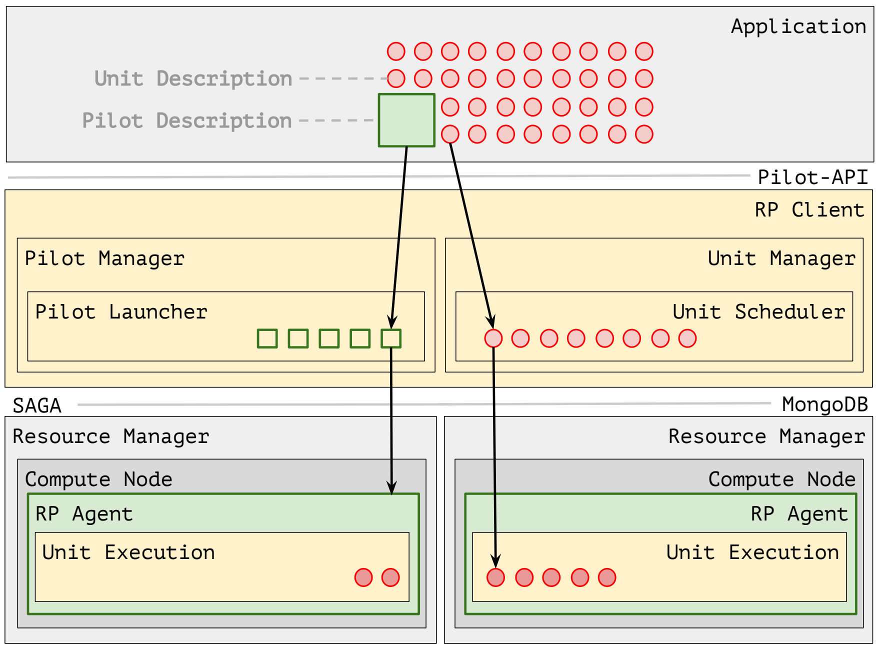

Figure 3.2: RADICAL-Pilot Architecture. Pilots (description and instance) in green are for resource allocation; Units (description and instance) in red are for task execution. Applications interact with RADICAL-Pilot through the Pilot-API. Resource interoperability comes through SAGA. Unit Manager to Agent communication is via MongoDB, all other communication is via ZeroMQ.

IBM's Blue Gene/Q, many of XSEDE's HPC resources, Amazon EC2, and on the Open Science Grid (OSG).

RADICAL-Pilot is a runtime system designed to execute heterogeneous and dynamic workloads on diverse resources. Workloads and pilots are described via the Pilot-API and passed to the RADICAL-Pilot runtime system, which launches the pilots and executes the tasks of the workload on them. Internally, RADICAL-Pilot represents pilots as aggregates of resources independent from the architecture and topology of the target machines, and workloads as a set of units to be executed on the resources of the pilot. Both pilots and units are stateful entities, each with a well-defined state model and life cycle. Their states and state transitions are managed via the three modules of the RADICAL-Pilot architecture: PilotManager, UnitManager, and Agent (Figure. 3.2). The PilotManager launches pilots on resources via the SAGA API [68]. The SAGA API implements an adapter for each type of supported resource, exposing uniform methods for job and data management. The UnitManager schedules units to pilots for execution. A MongoDB database is used to communicate the scheduled workload between the UnitManager and Agents. For this reason, the database instance needs to be accessible both from the user's workstation and the target resources. The Agent bootstraps on a remote resource, pulls units from the MongoDB instance, and manages their execution on the cores held by the pilot. RADICAL-Pilot has a well defined component and state model which is described in detail in [11].

The modules of RADICAL-Pilot are distributed between the client and the target resources. The PilotManager and UnitManager are executed on the user workstation (client) while the Agent runs on the target resources. RADICAL-Pilot requires Linux or OS X with Python 2.7 or newer on the workstation but the Agent has to execute different types of units on resources with very diverse architectures and software environments.

3.3 Pilot-API

RADICAL-Pilot (RP) is a Python library that enables the user to declaratively define the resource requirements and the workload. While the Pilot-API is a well-defined interface, the application specific relationships between resources and workload can be programmed in generic Python. In the following code snippets we walk the reader through a minimal but complete example of running a workload on OSG using RADICAL-Pilot.

```
# Create a session -- closing it will destroy all Managers
# and all things they manage.
session = rp.Session()

# Create a Pilot Manager.
pmgr = rp.PilotManager(session)

# Create a Unit Manager.
umgr = rp.UnitManager(session)
```

Listing 3.1: Code example showing the declaration of Pilot Manager and Unit Manager within a Session.

In Listing 3.1 we show the code used to declare the respective managers for pilots and units, whose lifetime is managed by a session object.

```
# Define a single core Compute Pilot that will run for 10 minutes.
cpdesc = rp.ComputePilotDescription({
    'resource': 'osg.xsede-virt-clust',
    'cores'   : 1,
    'runtime' : 10,
    'project' : 'TG-CCR140028',
    'queue'   : None
})

# Submit the Compute Pilot for launching.
compute_pilot = pmgr.submit_pilots(cpdesc)

# Make the Compute Pilot resources available to the Unit Manager.
umgr.add_pilots(pilot)
```

Listing 3.2: Code example showing the declaration of a Compute Pilot, its subsequent submission to the Pilot Manager and the attachment to the Unit Manager.

In Listing 3.2 we declare a Compute Pilot, by specifying where to start it, how many cores, the walltime, and optional queuing and project/accounting details. Once the pilot is submitted to the Pilot manager, it will get passed to the queuing system asynchronously. In the last step the pilot is associated to the unit manager, which means that this pilot can be used to execute units on.

```
# Define a Data Pilot on an \gls{srm} Storage Element.
dpdesc = rp.DataPilotDescription({
    'resource': 'osg.UCSDT2'
})

# Make the Data Pilot resources available to the Unit Manager.
data_pilot = pmgr.submit_data_pilots(dpdesc)
```

Listing 3.3: Declaration and submission of a Data Pilot.

In Listing 3.3 we declare a Data Pilot, by specifying its resource. In the last step the pilot is associated to the unit manager, which means that the storage on this Data Pilot can be used by Compute Units.

```
# Create a new Data Unit Description.
dud = rp.DataUnitDescription()
dud.file_urls = ["/etc/passwd"]

# Associate the Data Unit with all available Data Pilots.
data_unit = umgr.submit_data_units(dud, existing=True)
```

Listing 3.4: Declaration of a DataUnit.

In Listing 3.4 we declare a Data Unit. The Data Unit is now a logical handle to the specified files. At the final step, the Data Unit is brought under the management of the Unit Manager.

```
# Create a new CU description and fill it.
cud = rp.ComputeUnitDescription()

# Grep for the string 'John Doe' in a file named 'passwd'
# in the current directory.
cud.executable = '/bin/grep'
cud.arguments = ['-i', 'John Doe', 'passwd'],

# Associate the earlier created Data Unit as input to this Compute Unit.
cud.input_data = data_unit.uid

# Submit Compute Unit to Unit Manager.
umgr.submit_units(cud)

# Wait for the completion of the Compute Unit.
umgr.wait_units()

# Tear down Pilots and Managers.
session.close()
```

Listing 3.5: Code example showing the declaration of a Compute Units, the subsequent submission to the Unit Manager and the statement to wait for its completion.

In Listing 3.5 we finally declare the workload by creating a compute unit that specifies what to run with what input. The unit is then submitted to the unit manager which schedules the unit to a pilot. Once the pilot has become active, the unit may begin execution. The final wait call will block until all the units have reached a final state.

3.4 GWENDIA

GWENDIA is a data-driven workflow language for distributed computing based on array programming principles [9]. The orchestrations of tasks in workflows are well described through graphs where nodes represent data analysis processes and arcs represent their inter-dependencies.

In theory these inter-dependencies can either be data dependencies, where data exchange is needed between consecutive processes or pure control dependencies, where the dependency only enforces a synchronization of process execution in time. However, in practice, there is a data transfer involved in many cases encountered. Often scientific applications are described as data analysis pipelines: successive processes are inter-dependent through data elements, often exchanged by means of files, that are produced and consumed during the analysis. This is especially true when dealing with independent (legacy) applications without message passing interface. Indeed, among the many existing scientific workflow languages, focus is often put on the data although it does not always appear explicitly.

To illustrate this discussion, Figure 3.3 shows a simple application workflow pattern encoded using three different families of languages.

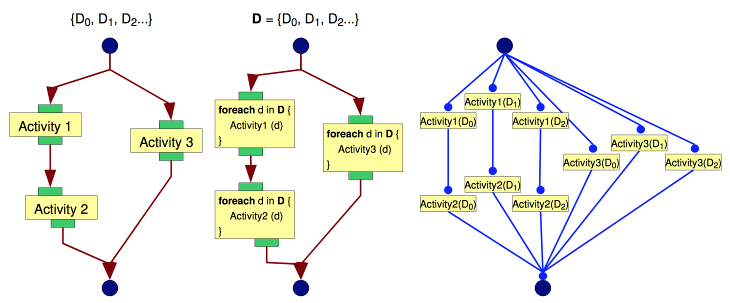

Figure 3.3: Application workflow pattern encoded using three different families of languages. From left to right: pure data-driven language, explicit variables assignments and parallel constructs, and pure control flow. The red arrows show data dependencies and the blue connectors represent control dependencies between activities. (Figure courtesy of [9])

Array programming was designed to improve the description of math processes for manipulating arrays [69]. Array programming principle is not limited to arithmetic operations and can be generalized to any case of function application.

In array programming, arrays are defined as indexed collections of data items with homogeneous type. An array of objects defines a new data type and therefore, arrays may be nested at any depth. Every data item is associated with a type, and a (multi-dimensional) integer index (one per nested level). For example, x = 'foo', 'bar', '42' is a 2 level array of strings and x(0,1) refers to the string 'bar'.

As an operator or function can be applied either to scalars or arrays in array programming languages, the data-driven language define computing activities independently from the data objects submitted to these activities. An activity will fire one or more times depending on the exact input data set it receives. Consider the example given on the left of Figure 3.3. Activity 1 will fire 3 times as it receives the array with 3 scalar values during the workflow execution. Depending on the activities port depth, the array is then either processed as a whole or unfolded. Iterations over the array element is handled (implicitly) by the execution engine. GWENDIA defines the following iteration strategies: dot product, cross product, flat cross product and match product. Iteration strategies (introduced in Scufl [70]) define how the activity processes data elements if multiple input and/or output ports are available.

In addition to implicit data flow constructs, GWENDIA also has explicit conditional and loop control structures to influence the execution of the workflow.

GWENDIA supports the integer, double, string and file data structures. In this thesis we only consider the file data structure.

Chapter 4

Implemented Software

In order to perform the experiments to validate the hypotheses posed a number of software systems had to be developed and extended.

4.1 RADICAL-SAGA

Simple API for Grid Applications (SAGA) is an Open Grid Forum (OGF) standard [71] that specifies a high-level interface to the most commonly used distributed computing functionality. SAGA defines an access-layer and mechanisms for distributed infrastructure components like job schedulers, file transfer and resource provisioning services. Given the heterogeneity of distributed infrastructure, SAGA provides am interoperability layer that decreases the complexity and lowers the threshold of using distributed infrastructure while at the same time enhancing the sustainability of distributed applications, services and tools.

RADICAL-SAGA [68] provides a Python implementation that is compliant with the SAGA specification. Behind the API, RADICAL-SAGA implements a flexible adaptor architecture as depicted in Figure 4.1. Adaptors are (dynamically loadable) Python modules that interface applications through the API with different middleware systems and services. Most users and application developers use the adaptors that are already part of RADICAL-SAGA, but one can implement their own in case a backend system is not supported yet.

RADICAL-SAGA's main focus is ease of use and simple user-space deployment in heterogeneous distributed computing environments. It supports a wide range of application use-cases from simple, uncoupled tasks to complex workflows. RADICAL-SAGA is being used on many distributed cyberinfrastructures such as XSEDE and OSG, as well as on many leadership class super computers such as Titan and Blue Waters.

In the context of this thesis RADICAL-SAGA a Job adaptor[1] was developed

[1]`https://github.com/radical-cybertools/saga-python/blob/fix/mark_condor/src/saga/adaptors/condor/condorjob.py`

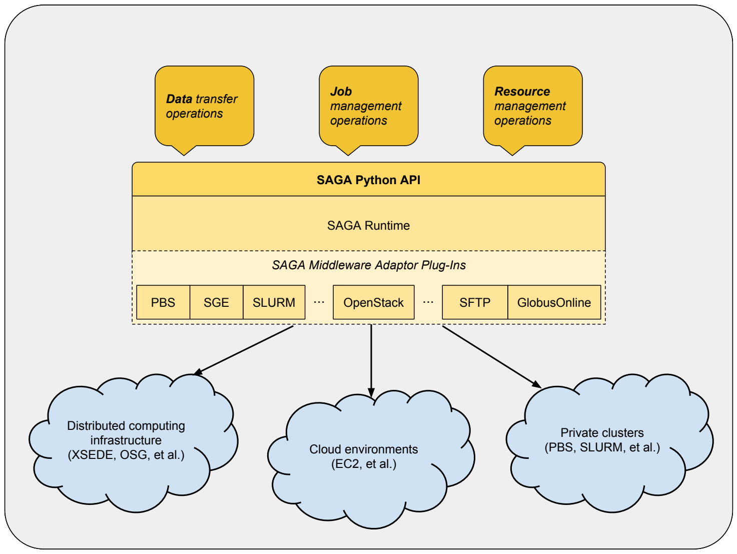
Data transfer operations
Job management operations
Resource management operations
SAGA Python API
SAGA Runtime
SAGA Middleware Adaptor Plug-Ins
PBS
SGE
SLURM
...
OpenStack
...
SFTP
GlobusOnline
Distributed computing infrastructure (XSEDE, OSG, et al.)
Cloud environments (EC2, et al.)
Private clusters (PBS, SLURM, et al.)

Figure 4.1: RADICAL-SAGA Architecture.

to run jobs on the OSG resources and a File adaptor was implemented[2] to support SRM [72] storage systems. The Condor adaptor is using the HTCondor client tools and the SRM adaptor is using the [73].

4.2 RADICAL-Pilot

In Section 3.2 we presented RADICAL-Pilot as a Pilot job system implemented in Python mainly for HPC systems. In the context of this thesis RADICAL-Pilot was extended to also support the OSG, as an instance of a HTC infrastructure. On HPC systems there is generally one pilot agent per job that orchestrates all the resources that belong to that job. In contrast, because of the distributed nature of resources, on the OSG there is a pilot agent for every compute resource. While this is not a fundamental difference, some practical obstacles had to be overcome in order for this to work. The mode of operation for RADICAL-Pilot on the OSG is that via a so called submission node that operates a GlideinWMS installation. The compute resource support of the OSG relies heavily on the HTCondor changes to SAGA as mentioned in Section 4.1. The other extension of RADICAL-Pilot required for support of the OSG is the capability of the agent to pull in input data into the agent environment from a tertiary source and push out output data back to a tertiary location.

[2]`https://github.com/radical-cybertools/saga-python/blob/feature/srm/src/saga/adaptors/srm/srmfile.py`

4.3 Pilot-Data

The main topic of this thesis is Pilot-Data [14], conceptually introduced in Section 3.1. In this section we describe the extension of RADICAL-Pilot that implements the Pilot-Data abstraction. In Listing 3.3 and Listing 3.4 (Section 3.3) we showed the code for declaring a data pilot and a data unit. When a data unit gets associated to a compute unit as input or output, the unit scheduler will take care of the data dependency resolution. Practically, this means that before a compute unit gets launched on a resource, the Pilot-Agent will stage in the data using the SAGA/SRM capabilities discussed in Section 4.2 into the compute unit's sandbox. Similarly the Pilot-Agent will stage out the output files of a compute unit's execution after its completion. When multiple data pilots (e.g. at multiple storage locations) have been associated to the runtime, and an input unit is available at more than one location, the unit scheduler has the freedom to pick one instance based on policy and/or heuristics. Currently the scheduler takes as input historic data transfer performance results and can either select the 'fastest', 'slowest' or a 'random' instance of an available data unit. A similar scheduling decision is applied for the output data.

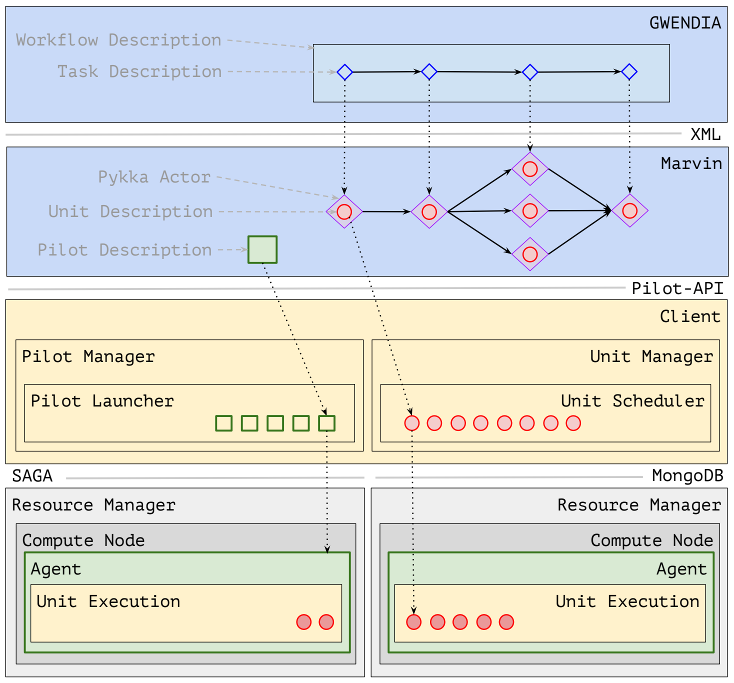

Figure 4.2: Marvin architecture and integration with other components in the stack. A GWENDIA workflow describes the activities and their relationships. During enactment, these activities get instantiated as actors representing tasks on the infrastructure. These tasks are then submitted using RADICAL-Pilot.

4.4 Marvin

Marvin [74] is a workflow system implemented using Pykka, supporting the execution of GWENDIA workflows. Marvin is fully aware of pilot jobs and pilot data. Pykka [75] is a Python implementation of the actor model [76]. The actor model introduces some simple rules to control the sharing of state and cooperation between execution units, which makes it easier to build concurrent applications. Figure 4.2 shows the high level architecture of the complete stack. Marvin takes a GWENDIA workflow and source data descriptions (both in XML) as input parameters. It then creates actors for all input and output ports and for the abstract activities. Based on the triggering of ports and activities it will create new actors for all instantiated tasks. Task actors live as long as they represent a running task on the infrastructure. Once all tasks are completed and output ports are satisfied, the execution terminates. In addition to the workflow and input descriptions, Marvin also takes a resource description as input. It will create pilots using RADICAL-Pilot based on the description provided. Marvin currently does not dynamically allocate resources based on the given workflow. Currently Marvin does not implement control structures either, however, these were not required for the given workflow, as the workflow is fully data flow oriented.

4.5 Discussion

The workflow presented here has first been manually translated into an application encoded using the Pilot-API in Section 3.3, and illustrates that compute-data orchestration, coordination and execution in a distributed environment can be expressed and captured using the Pilot-API[15]. In Section 4.4 we presented how Marvin, a workflow runtime system for the GWENDIA language, could be built on top of the Pilot-API.

Let us now revisit the qualitative research question Q1:

Q1 *Can the semantics of a GWENDIA workflow be expressed using the Pilot-API?*

A1-i *The combination of the Pilot-API expressiveness and the general purposeness of Python allows the user to specify dataflow oriented workflow patterns.*

Chapter 5

Experiments

To characterize the introduced concepts and to quantitatively answer the research questions, a set of experiments are designed. This chapter starts with a description of the infrastructure that is used for the experiments and then describes the different classes of experiments in detail.

5.1 Target infrastructure

The OSG [77] facilitates access to distributed HTC resources for research. The resources accessible through the OSG are contributed by the community members, but organized by the OSG. The OSG consists of computing and storage elements at over hundred individual sites, mainly spanning the US and some in South and Middle America. These sites are primarily at universities and national labs and range in size from a few hundred to tens of thousands of CPUs. The distributed nature of these resource providers allows users from a single Virtual Organization (VO) to submit their jobs at a single entry point and have them execute at whatever resource is available. Sharing is a core principle of the OSG. Over 100 million CPU hours delivered on the OSG are annually utilized opportunistically (resources that would otherwise have remained idle). This is the aspect of the OSG that allows individual researchers who might not otherwise have access to large computing resources to do so. A VO is a set of groups or users defined by some common infrastructure need. This can be anything from a scientific experiment, a university campus or a distributed research effort. A VO represents all its members and their common needs in a grid environment, and major projects such as CMS and ATLAS are represented in the OSG as VOs.

For the experiments in this thesis we access the OSG through the XSEDE glideinWMS installation at SDSC. This is a HTCondor pool that runs as the generic 'OSG' VO on all the OSG resources that support this VO. Table 5.1 shows the list of sites that have been used and whether the site has Compute Elements and/or Storage Elements. Figure 5.1 visualizes all used sites on the

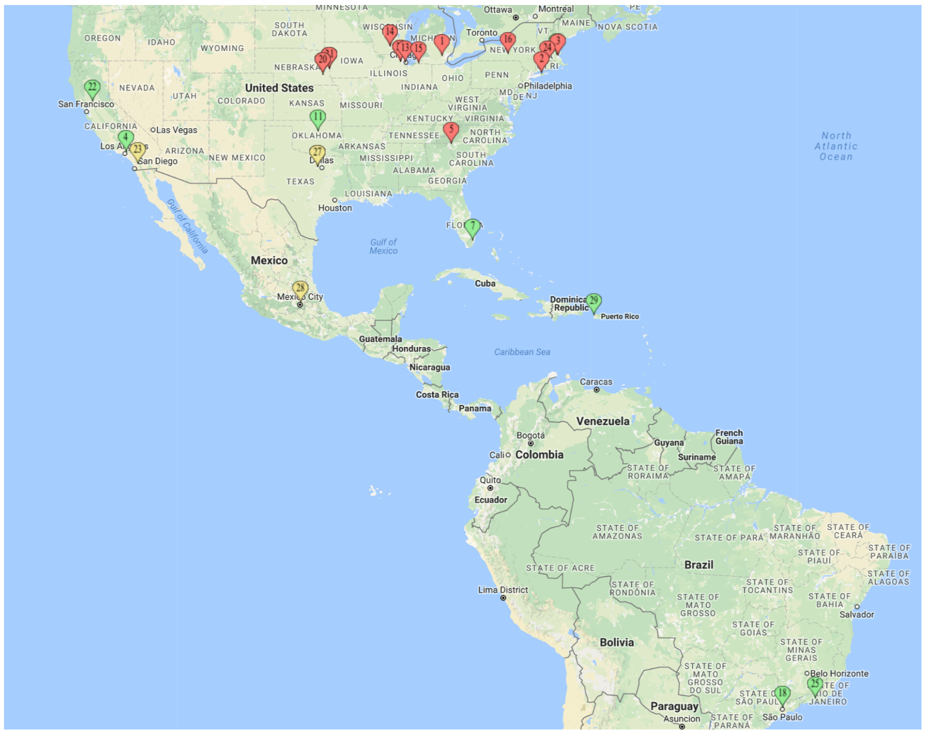
Figure 5.1: Sites used in experiments. Sites are numbered according to the order in Table 5.1. Green represents that a site only has Storage, Red only Compute, and Yellow both Compute and Storage.

map.

5.2 Storage Element Transfer Baseline

To characterize transfer capabilities between all sites we perform file transfer measurements between all storage elements for file sizes as specified in Table 5.2. These experiments do not involve Compute Elements, and therefore also does not involve RADICAL-Pilot. The transfers are orchestrated using third-party-transfer GridFTP commands via RADICAL-SAGA.

5.3 Pilot Transfer Baseline

To characterize transfer capabilities between Compute Elements and Storage Elements we perform file transfer measurements back and forth between all Compute Elements and all Storage Elements for file sizes as specified in Table 5.2.

Id	Site Name	Compute	Storage
1	AGLT2	Yes	No
2	BNL-ATLAS	Yes	No
3	BU_ATLAS_Tier2	Yes	No
4	CIT_CMS_T2	No	Yes
5	Clemson-Palmetto	Yes	No
6	Crane	Yes	No
7	FIUPG	No	Yes
8	GLOW	Yes	Yes
9	GPGrid	Yes	No
10	Hyak	Yes	No
11	LUCILLE	No	Yes
12	MIT_CMS	No	Yes
13	MWT2	Yes	No
14	NPX	Yes	No
15	NWICG_NDCMS	Yes	No
16	NYSGRID_CORNELL_NYS1	Yes	No
17	Nebraska	No	Yes
18	SPRACE	No	Yes
19	SWT2_CPB	Yes	Yes
20	Sandhills	Yes	No
21	Tusker	Yes	No
22	UCD	No	Yes
23	UCSDT2	Yes	Yes
24	UConn-OSG	Yes	No
25	UERJ	No	Yes
26	USCMS-FNAL-WC1	Yes	No
27	UTA_SWT2	Yes	Yes
28	cinvestav	Yes	Yes
29	uprm-cms	No	Yes

Table 5.1: List of sites, numbered by Id's in Figure 5.1 and specifying whether the site hosts Compute Elements and/or Storage Elements.

Label	Size
Micro	1 MB
Small	10 MB
Medium	100 MB
Large	1000 MB

Table 5.2: Data sizes for transfer experiments.

These experiments involve Compute Elements, and are therefore executed using RADICAL-Pilot. We create a RADICAL-Pilot application that consists of multiple Compute Units that have input and output configured in such a way that all combinations of Compute Element and Storage Element are measured. The RADICAL-Pilot Agent uses GridFTP via RADICAL-SAGA to effectuate the transfers. The inputs and outputs are 'hardcoded' by the experiment driver script and do not use RADICAL-Pilot's Pilot-Data capabilities.

5.4 Pilot-Data Characterizing

The experiments described in this section have similarities to the experiments described in the Section 5.3. The goal is again to create baseline insight into the performance of Compute Element to Storage Element transfers, but now using RADICAL-Pilot's Pilot-Data capabilities. This will allow us to compare and contrast the various Pilot-Data source and destination selection criteria. File sizes for the experiments are specified in Table 5.2.

5.5 Use case: Next-Generation Sequence Alignment

The final set of experiments build upon the experiments in Section 5.4. But instead of independent Compute Units with input and outputs, we now execute a fully integrated DNA sequencing workflow with real data [7].

The structure of the workflow is depicted in Figure 5.2. Besides the BWA alignment step, it contains data conversion steps to transform from the DNA sequencing machine format (*.csFasta) to the BWA format (*.fastq), as well as to split/merge the sequences to allow for parallel processing. The alignment of each data chunk is performed against the human genome reference database. First a data conversion step takes place for the paired-end files with the solid-to-fastq component, where the sequence and quality information are combined into two fastq files (solid-to-fastq component). Since the datasets are relatively large, these files are split into smaller chunks (split-fastq component). The user can define how large the chunks should be, and the files are split accordingly and transferred back to SRM storage. These chunks are then used as input to the sequence alignment step (bwa component), which is executed in parallel on each

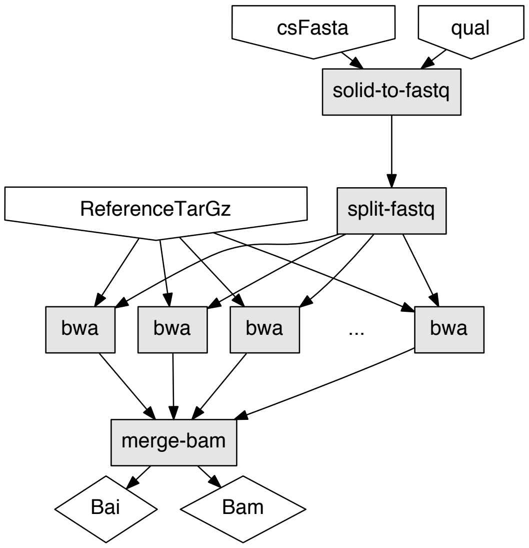

Figure 5.2: The GWENDIA/Marvin DNA Sequencing Workflow with legend.

chunk of data. The results of the parallel jobs are stored onto a single directory on grid storage. After all alignments have been performed the intermediate files are passed on to the merge component (merge-bam). This last component retrieves the files from grid storage and combines all alignment results into one file, which is the final output of the workflow. For comparative reasons we round-off the input and output sizes of the workflow to the sizes used in the baseline experiments specified as specified in Table 5.2.

Exp	Count	Input (MB)	Chunks	Reference (MB)	Conversion (s)	Split (s)	BWA (s)	Merge (s)
A	10	10	10	1	1	1	10	1
B	10	100	10	10	10	10	100	10
C	10	1000	10	100	100	100	1000	100

Table 5.3: Workflow data sizes and parameters configuration for experiment A, B and C. Size entries are in MBs and duration entries are in seconds.

The experiments are performed in three different configurations, named A, B and C. Table 5.3 shows the configurations. As described earlier, GWENDIA is a data parallel language, meaning that for every given (set of) input(s), the workflow is executed. Count refers to the number of input data sets. Chunks is the number of outputs that the Split component creates out of a single input. Reference refers to the size of the reference database used by the BWA component. The remaining four parameters are the respective (artificial) runtimes of the components that are relative to their input size.

Exp	Conversion		Split		BWA		Merge	
	In	Out	In	Out	In	Out	In	Out
A	10	10	10	10×1	1+1	1	10×1	10
B	100	100	100	10×10	10+10	10	10×10	100
C	1000	1000	1000	10×100	100+100	100	10×100	1000

Table 5.4: Input and output data volumes per component instance for experiments A, B and C. All entries are MBs.

Exp	Input	Output	Total
A	500	400	900
B	5000	4000	9000
C	50000	40000	90000

Table 5.5: Total data volumes for experiments A, B and C. All entries are in MBs.

Based on the number of chunks and input sizes per Table 5.3 we can derive the input and output volumes of every component which is shown in Table 5.4.

If we combine the number of inputs from Table 5.3 with the resulting data volumes in Table 5.4 we can derive the total input and output volumes of the workflow for the three experimental configurations as shown in Table 5.5.

Chapter 6

Results

In this chapter we present the results obtained from the experiments described in the previous chapter.

6.1 Baseline SE-SE Transfer Times

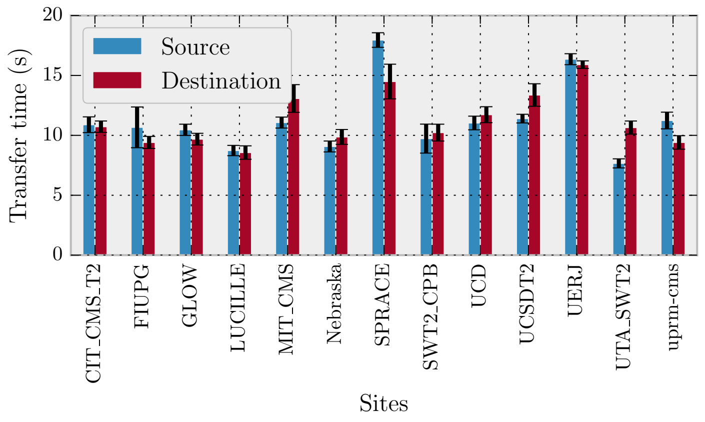

Figure 6.1: Average transfer time of a 1M file for each Storage Elements (SE) from and to all other SEs. The plot shows the results for both directions, in blue the SE is the source and in red the SE is the destination. Error bars show standard error.

We transferred files with the respective sizes many times over a longer period in both directions between all combinations of Storage Elements. In Figures 6.1 and 6.2 we display the results of the transfers of 1M and 1000M respectively. The error bars denote the standard error. The plots show the mean value of

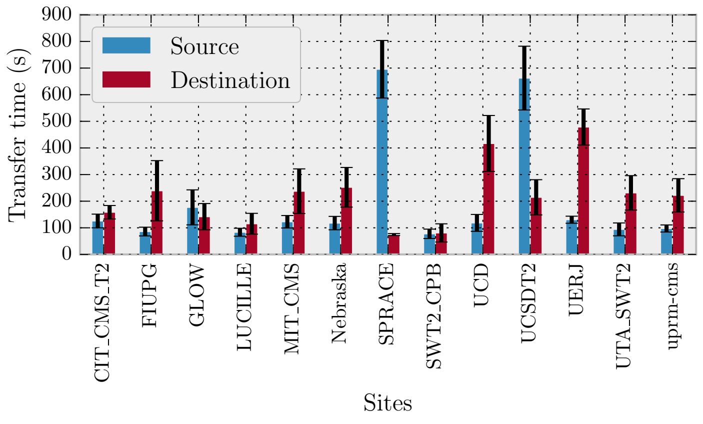

Figure 6.2: Average transfer time of a 1000M file for each Storage Elements (SE) from and to all other SEs. The plot shows the results for both directions, in blue the SE is the source and in red the SE is the destination. Error bars show standard error.

the transfers from one site to all other sites, and the other direction, from all sites to one site. We take both sides of the spectrum as the 1M files give an intuition for the connection overhead and the 1000M gives an intuition of the transfer speed. Results for 1M are in the same order of magnitude and mostly symmetric. In contrast, the results for 1000M show large variations between sites, and also large differences in the direction of the transfer.

Figures 6.3 and 6.4 show the same data, but in a full matrix.

6.2 Baseline SE-SE Reliability

While in Section 6.1 we presented the transfer times, in this section we look at the reliability of the same transfers.

Figures 6.5 and 6.6 show the reliability of the 1M and 1000M transfers respectively. Some of the sites have clearly better reliability than others. The results for 1M and 1000M show similar patterns which leads to the assumption that the file size has little impact on the success rate of transfers.

6.3 Baseline CE-SE Transfer Times

In this section we explore the baseline performance of transfers between Storage Elements and Compute Elements as described in Section 5.3. Note that as discussed in Section 5.1, some sites have both Compute Elements and Storage Elements, while others have only one of the two. This means that part of the Compute Element - Storage Element interactions remain on-site.

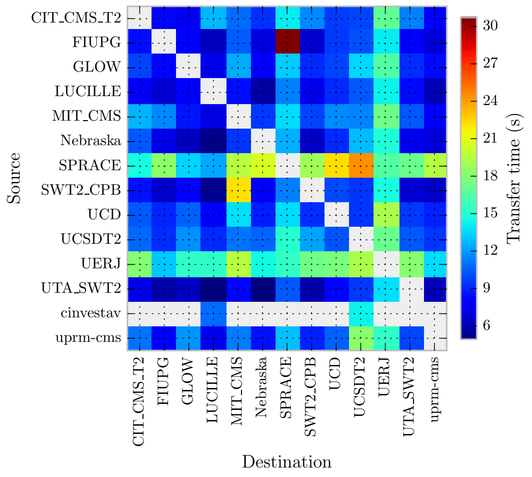

Figure 6.3: Average transfer time of a 1M file for each Storage Element (SE) from and to each other SE. The plot shows the results for both directions.

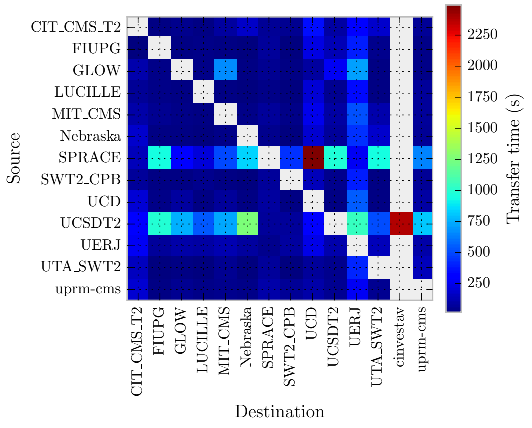

Figure 6.4: Average transfer time of a 1000M file for each Storage Element (SE) from and to each other SE. The plot shows the results for both directions.

In Figure 6.7 we show the mean transfer time of a 1000M file for each Compute Element from all Storage Elements.

Conversely, in Figure 6.8 we show the mean transfer time of a 1000M file

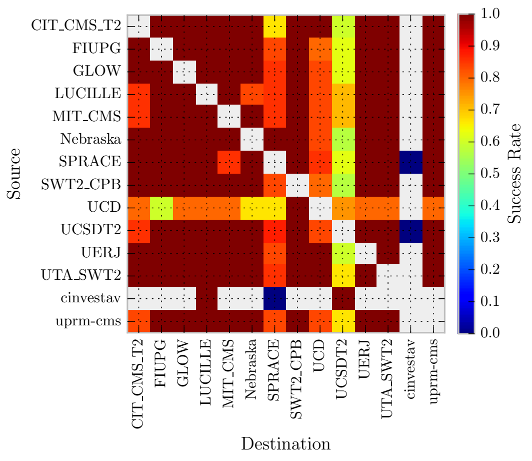

Figure 6.5: Success rate of the transfer of a 1M file for each Storage Element (SE) from and to each other SE. The plot shows the results for both directions.

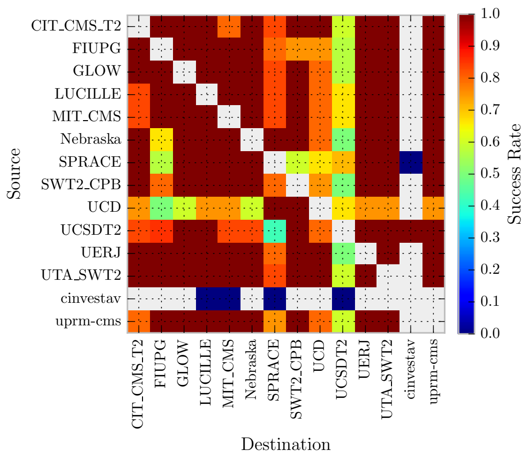

Figure 6.6: Success rate of the transfer of a 1000M file for each Storage Element (SE) from and to each other SE. The plot shows the results for both directions.

from each Storage Element to all Compute Elements.

Putting the two earlier plots together, in Figure 6.9 we display the transfer time of a 1000M file from each Storage Element to each Compute Element.

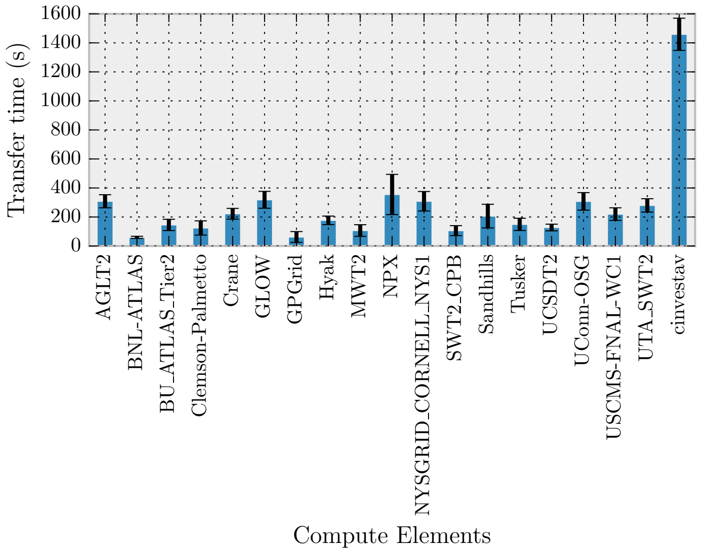

Figure 6.7: Average transfer time of a 1000M file for each Compute Element from all Storage Elements. Error bars show standard error.

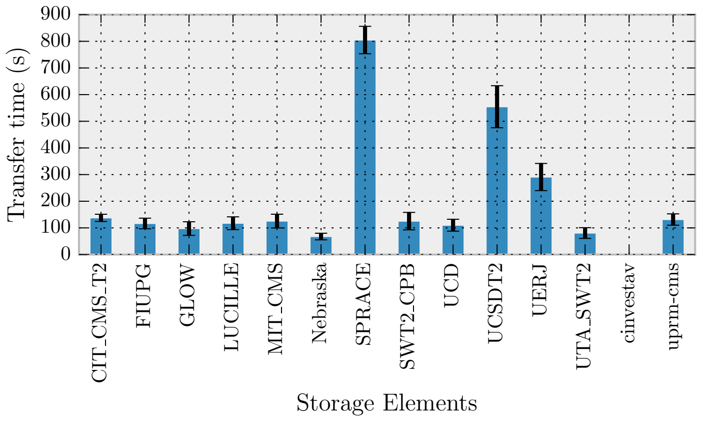

Figure 6.8: Average transfer time of a 1000M file for each Storage Element to all Compute Elements. Error bars show standard error.

6.4 Baseline CE-SE Reliability

For completeness we also show the reliability of all Storage Element to Compute Element transfers in Figure 6.10.

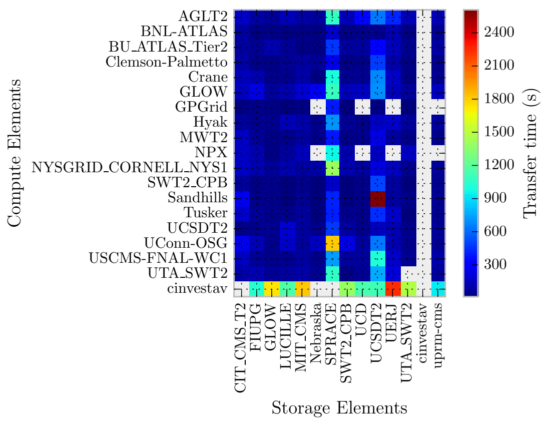

Figure 6.9: Average transfer time of a 1000M file for each Compute Element (CE) from each SE.

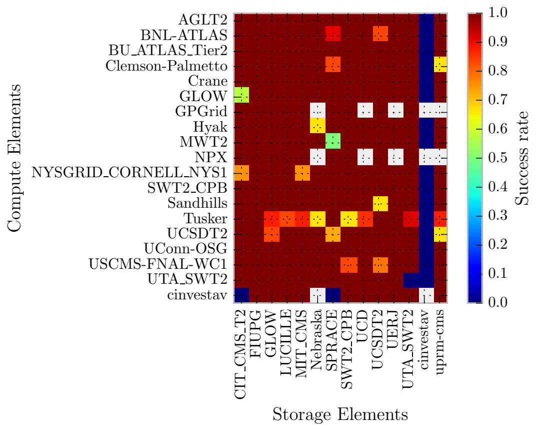

Figure 6.10: Success rate of the transfer of a 1000M file for each Compute Element (CE) from each SE.

6.5 Pilot-Data input location selection

In this section we show the first results based on the involvement of Pilot-Data. In Figure 6.11 we show mean transfer times of a 1000M file for each Compute Element (CE) with different source Storage Element (SE) selection methods. In blue are the results when the SE is randomly chosen. In purple we show the results of Pilot-Data selecting the fastest source site based on historical data. In purple we show the results of Pilot-Data selecting the slowest SE based on historical data. We can observe that selecting the 'slow' resource is in almost all situations indeed the 'worst' decision. Selecting the 'fast' resource is often the

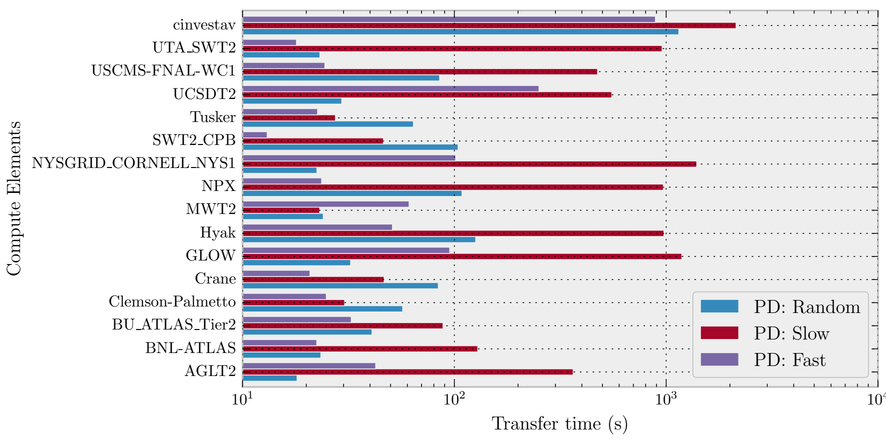

Figure 6.11: Average transfer time of a 1000M file for each CE with different selection methods. In blue are the results when the SE is randomly chosen. In purple we show the results of Pilot-Data selecting the fastest source site based on historical data. In purple we show the results of Pilot-Data selecting the slowest SE based on historical data.

best choice, but not always, as the 'random' pick seems to outperform the 'fast' method in a number of situations. We elaborate on this further in Section 7.

6.6 Marvin

Up till now we looked at results of transfers and components in an isolated way. This section will present the results of the integrated experiments with a DNA sequencing workflow executed by the Marvin workflow engine, as described in Section 5.5.

In Figure 6.12 we show the Time to Completion (TTC) for executing the BWA workflow with Marvin on the OSG. Sizes 10, 100, and 1000 correspond to the experiments A, B, and C from Table 5.3. For every input size configuration, we also the results for the 'fast', 'random', and 'slow' Pilot-Data selection mechanism. For input size 10MB the effect is negligible, for 100MB and 1000MB the improvement from selecting 'fast' is distinct. 'Slow' and 'random' perform similarly, with a non-symmetric standard error between 100MB and 1000MB.

Figure 6.13 shows the same data as the 1000M experiment in Figure 6.12, but now split out per component. The 'CU Duration' includes transferring the input data from an SE, running the task, and transferring the output data to an SE.

Figure 6.14 is a further refinement, now only showing the input and output transfers. Given that the runtime does not vary between the selection methods, this is a more insightful view of the difference. The difference in the performance

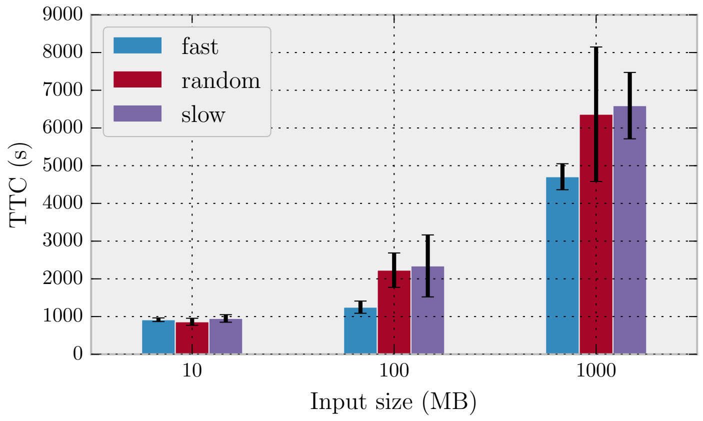

Figure 6.12: TTC for integrated Marvin experiment for different sizes and different selection methods.

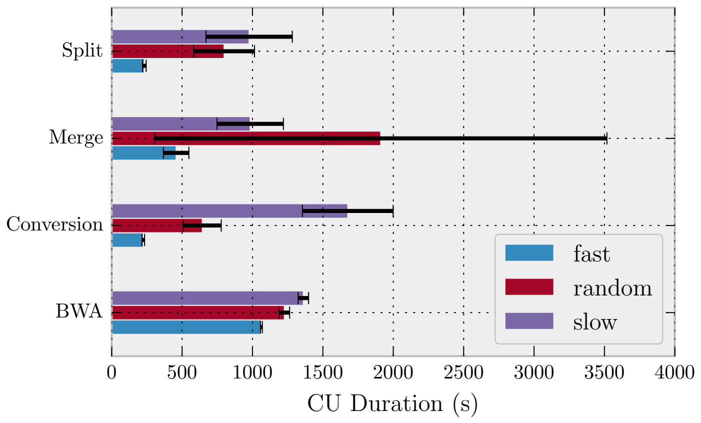

Figure 6.13: Duration per component for experiment size 1000.

between the selection methods is clearly not the same for all components. The standard error is generally lowest for the 'fast' method, except for the Merge component. In absolute terms for all methods the BWA component spends least time on transfers, which is consistent with the fact that each component deals with 1/10th of the data of the other components.

Figures 6.15 and 6.16 are the breakup of Figure 6.14 for each direction. Now we can see that the BWA and Conversion components have similar characteristics for both input and output, which is explained by its symmetric input and output patterns. The opposite is true for Split and Merge, which have different input and output patterns in terms of number of transfers.

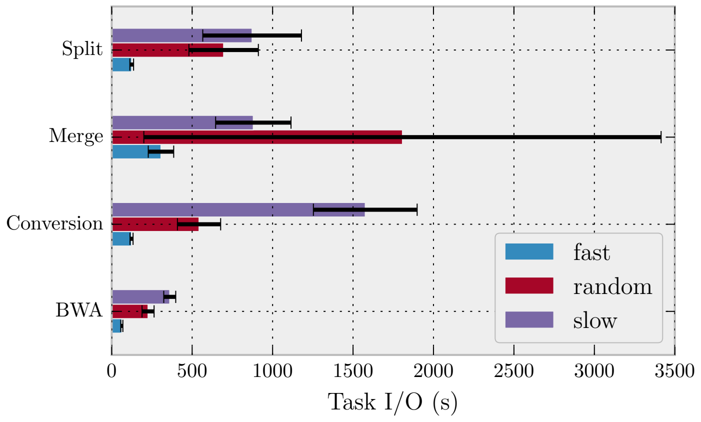

Figure 6.14: I/O transfer overhead per component for experiment size 1000.

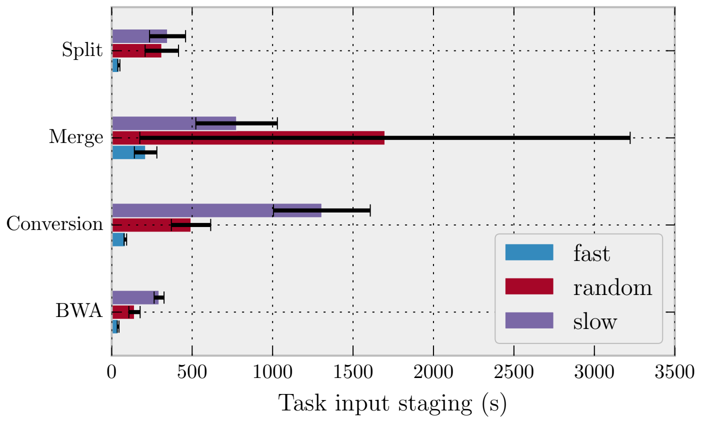

Figure 6.15: Input transfer overhead per component for experiment size 1000.

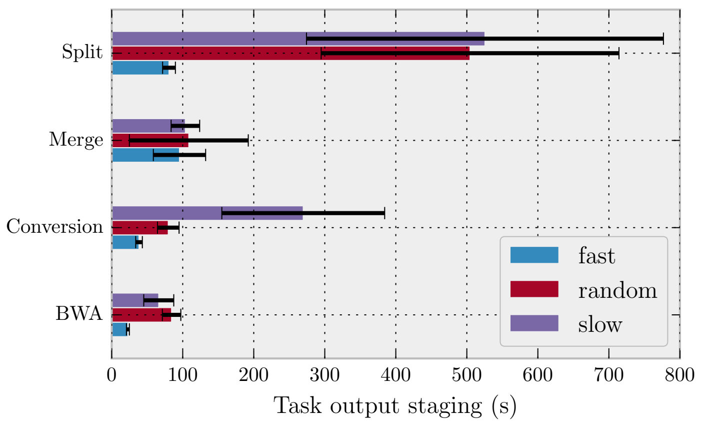

Figure 6.16: Output transfer overhead per component for experiment size 1000.

Chapter 7

Discussion

To structure the discussion of the results we revisit the research questions that relate to the quantitative aspects:

Q2 *What factors of data and compute coupling exist that can be exploited to improve execution of data intensive workflows?*

In Section 6.1 we report on the baseline transfer measurements between Storage Elements, which is expanded in Section 6.3 by also performing CE-SE measurements. The main observations are that there is wide spread in overall transfer performance of sites towards other sites, and that for some of the sites, the results are very asymmetric. Measurements with small files show relatively low performance, indicating that data volume is just one parameter, and that transfer setup overhead is significant. The outliers in Figure 6.8 correspond to the outliers in Figure 6.2, which satisfies the expectation that the differences that exist between SEs also translate to CEs. Although some sites stand out more than others, those sites are not exclusively 'slow', which means that the performance is ultimately dependent on a source and destination tuple.

A2-i *Knowledge about 'fast' and 'slow' source-destination tuples allows to give priority to faster transfer alternatives.*

Section 6.2 and report on the reliability of SE-SE and CE-SE transfers respectively. The data suggest that some sites are less reliable than the bulk of them, and that the reliability is not a function of the file size. Although some sites stand out more than others, they are not exclusively 'bad', which means that the reliability is ultimately dependent on a source and destination tuple.

A2-ii *Knowledge about 'good' and 'bad' source-destination tuples allows to give priority to reliable transfer alternatives.*

Q3 *Can the decision making about compute and data placement be automated in such a way that it has an impact on data intensive workflow execution?*

The Pilot-Data baseline measurements in Section 6.5 use historical data to determine which endpoint to choose. Without exception the ‘fast’ metric outperforms the ‘slow’ metric, and almost always the ‘random’ metric too.

The DNA sequencing workflow results in Section 6.6 show that, although in the integrated experiments there are more parameters that influence the performance, the Pilot-Data optimizations also perform better than random.

A3-i *RADICAL-Pilot’s PilotData implementation provided with historical data can make informed decisions that are better than random behavior.*

For the Pilot-Data experiments we gathered historical performance data to steer the decision process for future transfers. In Section 6.1 and 6.3 there is some correlation with transfer speed to distance, which can be both geographical and/or network distance, which could also be taken into account.

A3-ii *Any decision capability requires information, either actively gathered, or passively offered.*

Optimization is currently per transfer, and does not take global optimization into account. Specifically this means that many tasks can start to transfer to and/or from the same site at the same moment, and therefore undoing the effect of optimized selection. This implies that depending on the degree of concurrency, the ‘random‘ selection mechanism can be worse than ‘slow’ or alternatively, better than ’fast’.

A3-iii *Optimization decisions need to be global and not local.*

Chapter 8

Conclusion

We have shown how RADICAL-Pilot enhanced with Pilot-Data capabilities enabled the construction of an array based workflow system (Marvin) that makes transparently use of these capabilities. In both isolated and integrated experiments this lead to performance improvements.

In this work we only made the decision on location of input and output data dynamic. In similar fashion we could extend the implementation to also make the decision of where to place the Compute Unit more intelligently.

The considerations for locating data were centered around reliability and performance in this thesis. However, more dimensions of decision making exist, for example, privacy policies that dictate where (and where not) certain data can be processed, or a monetary cost factor of transferring data.

The results in the integrated DNA sequencing experiments were not completely isolated from other considerations. For example, the TTC of an application is not only dependent on the transfer performance, but also on the performance of the resources themselves, queuing effects, and policies.

In this thesis RADICAL-Pilot was extended with Pilot-Data. In [16] we have also introduced the concept of Pilot-Network, which would also make the network part more dynamic and part of the decision and control process.

The decision points in the Pilot-Data implementation are still rather static. Especially once the (pilot-)network would also become a dynamic entity, there is the need for a modelling effort to capture the relation between computing workload, data, and resources.

RADICAL-Pilot provides rich profiling and tracing capabilities and it would be interesting to explore how this functionality could be used to enable provenance in Marvin, in a similar way as [78].

The goal of this research was not to write (yet) another workflow system that would replace all existing systems. It is still a topic of debate whether the ecosystem of workflow systems would benefit from more shared efforts on the runtime level.

Bibliography

[1] 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-The International Journal Of Grid Computing-Theory Methods And Applications*, 25(5):528–540, 2009.

[2] Adam Belloum, Marcia A Inda, Dmitry Vasunin, Vladimir Korkhov, Zhiming Zhao, Han Rauwerda, Timo M Breit, Marian Bubak, and Luis O Hertzberger. Collaborative e-Science Experiments and Scientific Workflows. *Internet Computing, IEEE*, 15(4):39–47, 2011.

[3] M W A Caan, S. Shahand, F M Vos, A H C van Kampen, and S.D Olabarriaga. Evolution of grid-based services for Diffusion Tensor Image analysis. *Future Generation Computer Systems*, 28(8):1194–1204, October 2012.

[4] Angela CM Luyf, Barbera DC van Schaik, Michel de Vries, Frank Baas, Antoine HC van Kampen, and Silvia D Olabarriaga. Initial steps towards a production platform for DNA sequence analysis on the grid. *BMC Bioinformatics*, 11(1):598, December 2010.

[5] Shayan Shahand, Mark Santcroos, Antoine H C Kampen, and Sílvia Delgado Olabarriaga. A Grid-Enabled Gateway for Biomedical Data Analysis. *Journal of Grid Computing*, October 2012.

[6] A Luckow, Mark Santcroos, A Merzky, O Weidner, P Mantha, and S Jha. P∗: A model of pilot-abstractions. In *E-Science (e-Science), 2012 IEEE 8th International Conference on*, pages 1–10, 2012.

[7] BDC van Schaik, Mark Santcroos, and V Korkhov. Challenges in DNA sequence analysis on a production grid. In *EGI Community Forum 2012*, 2012.

[8] Tristan Glatard, J Montagnat, D Lingrand, and X 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, 2008.

[9] Johan Montagnat, Benjamin Isnard, Tristan Glatard, Ketan Maheshwari, and Mireille Fornarino. A data-driven workflow language for grids based

on array programming principles. *WORKS '09: Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, November 2009.

[10] André Luckow, Mark Santcroos, Ole Weidner, Andre Merzky, Sharath Maddineni, and Shantenu Jha. Towards a common model for pilot-jobs. In *HPDC '12: Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*. ACM, June 2012.

[11] Andre Merzky, Mark Santcroos, Matteo Turilli, and Shantenu Jha. Executing Dynamic and Heterogeneous Workloads on Super Computers, 2016. (under review) `http://arxiv.org/abs/1512.08194`.

[12] Mark Santcroos, Ralph Castain, Andre Merzky, Iain Bethune, and Shantenu Jha. Executing dynamic heterogeneous workloads on blue waters with radical-pilot. In *Cray User Group 2016*, 2016.

[13] BDC van Schaik, Mark Santcroos, S Madougou, A Jongejan, A H C van Kampen, and S.D Olabarriaga. e-Bioscience Solutions and Challenges for Next Generation Sequencing Experiments. In *2nd International Work-Conference on Bioinformatics and Biomedical Engineering*, pages 333–334, 2013.

[14] A Luckow, Mark Santcroos, A Zebrowski, and S Jha. Pilot-data: an abstraction for distributed data. *Journal of Parallel and Distributed Computing*, 79-80:16–30, 2015.

[15] Mark Santcroos, Barbera DC van Schaik, Shayan Shahand, Sílvia Delgado Olabarriaga, Andre Luckow, and Shantenu Jha. Exploring Dynamic Enactment of Scientific Workflows using Pilot-Abstractions. In *13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 1–9, Delft.

[16] Mark Santcroos, S Delgado Olabarriaga, D S Katz, and S Jha. Pilot abstractions for compute, data, and network. In *E-Science (e-Science), 2012 IEEE 8th International Conference on*, pages 1–2, 2012.

[17] Matteo Turilli, Mark Santcroos, and Shantenu Jha. A comprehensive perspective on pilot-jobs, 2016. (under review) `http://arxiv.org/abs/1508.04180`.

[18] Fran Berman, Rich Wolski, Silvia Figueira, Jennifer Schopf, and Gary Shao. Application-level scheduling on distributed heterogeneous networks. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 39–39. IEEE, 1996.

[19] 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.

[20] James Frey, Todd Tannenbaum, Miron Livny, Ian Foster, and Steven Tuecke. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[21] Jakub T Mościcki. DIANE - distributed analysis environment for GRID-enabled simulation and analysis of physics data. In *Proceedings of the IEEE Nuclear Science Symposium Conference Record*, volume 3, pages 1617–1620. IEEE, 2003.

[22] Pablo Saiz, L Aphecetche, Predrag Bunčić, Ružica Piskač, J-E Revsbech, Vedran Šego, Alice Collaboration, et al. AliEn: ALICE environment on the GRID. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 502(2):437–440, 2003.

[23] Adrian Casajus, Ricardo Graciani, Stuart Paterson, Andrei Tsaregorodtsev, et al. DIRAC pilot framework and the DIRAC Workload Management System. In *Proceedings of the 17th International Conference on Computing in High Energy and Nuclear Physics (CHEP09), Journal of Physics: Conference Series*, volume 219(6), page 062049. IOP Publishing, 2010.

[24] T Maeno, K De, A Klimentov, P Nilsson, D Oleynik, S Panitkin, A Petrosyan, J Schovancova, A Vaniachine, T Wenaus, et al. Evolution of the ATLAS PanDA workload management system for exascale computational science. In *Proceedings of the 20th International Conference on Computing in High Energy and Nuclear Physics (CHEP2013), Journal of Physics: Conference Series*, volume 513(3), page 032062. IOP Publishing, 2014.

[25] Igor Sfiligoi, Daniel C Bradley, Burt Holzman, Parag Mhashilkar, Sanjay Padhi, and Frank Würthwein. The pilot way to grid resources using glideinWMS. In *Proceedings of the World Congress on Computer Science and Information Engineering*, volume 2, pages 428–432. IEEE, 2009.

[26] Ruth Pordes et al. The Open Science Grid. *J. Phys.: Conf. Ser.*, 78(1):012057, 2007.

[27] Jason Cope, Michael Oberg, Henry M Tufo, Theron Voran, and Matthew Woitaszek. High throughput grid computing with an IBM Blue Gene/L. *2007 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 357–364, 2007.

[28] Nitro web site. `http://www.adaptivecomputing.com/products/hpc-products/high-throughput-nitro/`.

[29] Mysge. `http://www.nersc.gov/users/analytics-and-visualization/data-analysis-and-mining/mysge/`.

[30] QDO web site. `https://www.nersc.gov/users/data-analytics/workflow-tools/other-workflow-tools/qdo/`.

[31] T. Kosar and M. Livny. Stork: making data placement a first class citizen in the grid. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pages 342 – 349, 2004.

[32] Alexandru Romosan, Doron Rotem, Arie Shoshani, and Derek Wright. Co-scheduling of computation and data on computer clusters. In *Proceedings of 17th International Conference on Scientific and Statistical Databases Management (SSDBM)*, 2005.

[33] Devarshi Ghoshal and Lavanya Ramakrishnan. Frieda: Flexible robust intelligent elastic data management in cloud environments. *High Performance Computing, Networking Storage and Analysis, SC Companion:*, 0:1096–1105, 2012.

[34] K. Ranganathan and I. Foster. Decoupling computation and data scheduling in distributed data-intensive applications. In *High Performance Distributed Computing, 2002. HPDC-11 2002. Proceedings. 11th IEEE International Symposium on*, pages 352 – 358, 2002.

[35] William H. Bell, David G. Cameron, A. Paul Millar, Luigi Capozza, Kurt Stockinger, and Floriano Zini. Optorsim: A grid simulator for studying dynamic data replication strategies. *International Journal of High Performance Computing Applications*, 17(4):403–416, 2003.

[36] Jianwei Ma, Wanyu Liu, and Tristan Glatard. A classification of file placement and replication methods on grids. *Future Generation Computer Systems*, 29(6):1395 – 1406, 2013. Including Special sections: High Performance Computing in the Cloud & Resource Discovery Mechanisms for P2P Systems.

[37] A. Aamnitchi, S. Doraimani, and G. Garzoglio. Filecules in high-energy physics: Characteristics and impact on resource management. In *High Performance Distributed Computing, 2006 15th IEEE International Symposium on*, pages 69 –80, 0-0 2006.

[38] A. Amer, D.D.E. Long, and R.C. Burns. Group-based management of distributed file caches. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pages 525 – 534, 2002.

[39] Gregory Ganger and M. Frans Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *In Proceedings of the 1997 USENIX Technical Conference*, pages 1–17, 1997.

[40] Gilles Fedak, Haiwu He, and Franck Cappello. Bitdew: a programmable environment for large-scale data management and distribution. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 45:1–45:12, Piscataway, NJ, USA, 2008. IEEE Press.

[41] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 137–150, Berkeley, CA, USA, 2004. USENIX Association.

[42] Apache Hadoop. `http://hadoop.apache.org/`, 2014.

[43] Pradeep Kumar Mantha, Andre Luckow, and Shantenu Jha. Pilot-MapReduce: An Extensible and Flexible MapReduce Implementation for Distributed Data. In *Proceedings of third international workshop on MapReduce and its Applications*, MapReduce '12, pages 17–24, New York, NY, USA, 2012. ACM.

[44] Michael D. Beynon, Tahsin Kurc, Umit Catalyurek, Chialin Chang, Alan Sussman, and Joel Saltz. Distributed processing of very large datasets with datacutter. *Parallel Comput.*, 27(11):1457–1478, October 2001.

[45] Andre Luckow, Mark Santcroos, Andre Merzky, Ole Weidner, Pradeep Mantha, and Shantenu Jha. P*: A model of pilot-abstractions. *IEEE 8th International Conference on e-Science*, pages 1–10, 2012. `http://dx.doi.org/10.1109/eScience.2012.6404423`.

[46] 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, 2011.

[47] A. Tsaregorodtsev, N. Brook, A. Casajus Ramo, P. Charpentier, J. Closier, et al. DIRAC3: The new generation of the LHCb grid software. *J.Phys.Conf.Ser.*, 219:062029, 2010.

[48] S Bagnasco, L Betev, P Buncic, F Carminati, C Cirstoiu, C Grigoras, A Hayrapetyan, A Harutyunyan, A J Peters, and P Saiz. Alien: Alice environment on the grid. *Journal of Physics: Conference Series*, 119(6):062012, 2008.

[49] T Maeno, K De, T Wenaus, P Nilsson, G A Stewart, R Walker, A Stradling, J Caballero, M Potekhin, D Smith, and The Atlas Collaboration. Overview of atlas panda workload management. *Journal of Physics: Conference Series*, 331(7):072024, 2011.

[50] Tadashi Maeno, K. De, and S. Panitkin. PD2P: PanDA dynamic data placement for ATLAS. In *Journal of Physics: Conference Series*, volume 396, page 032070, 2012.

[51] Ioan Raicu, Yong Zhao, Catalin Dumitrescu, Ian Foster, and Mike Wilde. Falkon: A Fast and Light-Weight TasK ExecutiON Framework. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM.

[52] Ioan Raicu, Yong Zhao, Ian T. Foster, and Alex Szalay. Accelerating large-scale data exploration through data diffusion. In *Proceedings of the 2008 international workshop on Data-aware distributed computing*, DADC '08, pages 9–18, New York, NY, USA, 2008. ACM.

[53] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, et al. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, 46:17–35, 2015.

[54] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahl, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.

[55] Michael Albrecht, Patrick Donnelly, Peter Bui, and Douglas Thain. Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, page 1. ACM, 2012.

[56] Anubhav Jain, Shyue Ping Ong, Wei Chen, Bharat Medasani, Xiaohui Qu, Michael Kocher, Miriam Brafman, Guido Petretto, Gian-Marco Rignanese, Geoffroy Hautier, et al. FireWorks: a dynamic workflow system designed for high-throughput applications. *Concurrency and Computation: Practice and Experience*, 2015.

[57] Peter Bui, Dinesh Rajan, Badi Abdul-Wahid, Jesus Izaguirre, and Douglas Thain. Work Queue + Python: A framework for scalable scientific ensemble applications. In *Workshop on Python for High Performance and Scientific Computing at SC11*, 2011.

[58] 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, 2011.

[59] Ioan Raicu, Yong Zhao, Catalin Dumitrescu, Ian Foster, and Mike Wilde. Falkon: a Fast and Light-weight tasK executiON framework. In *Proceedings of the 8th ACM/IEEE conference on Supercomputing*, page 43. ACM, 2007.

[60] Mihael Hategan, Justin Wozniak, and Ketan Maheshwari. Coasters: uniform resource provisioning and access for clouds and grids. In *Proceedings of the 4th IEEE International Conference on Utility and Cloud Computing (UCC)*, pages 114–121. IEEE, 2011.

[61] Mats Rynge, Scott Callaghan, Ewa Deelman, Gideon Juve, Gaurang Mehta, Karan Vahi, and Philip J Maechling. Enabling large-scale scientific workflows on petascale resources using MPI master/worker. In *XSEDE*

'12: Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the campus and beyond, July 2012.

[62] Vladimir Korkhov, Dmitry Vasyunin, Adianto Wibisono, Victor Guevara-Masis, Adam Belloum, Cees de Laat, Pieter Adriaans, and L O Hertzberger. WS-VLAM: towards a scalable workflow system on the grid. In *WORKS '07: Proceedings of the 2nd workshop on Workflows in support of large-scale science.* ACM, June 2007.

[63] G. Juve and E Deelman. Resource Provisioning Options for Large-Scale Scientific Workflows. In *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, pages 608–613. IEEE, 2008.

[64] G. Juve, E Deelman, K Vahi, and G. Mehta. Experiences with resource provisioning for scientific workflows using Corral. *Scientific Programming*, 18(2):77–92, 2010.

[65] Gurmeet Singh and Ewa Deelman. The interplay of resource provisioning and workflow optimization in scientific applications. *Concurrency and Computation: Practice and Experience*, 23(16), November 2011.

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

[67] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed Computing in Practice:The Condor Experience. *Concurrency and Computation: Practice & Experience - Grid Performance*, 17(2-4):323–356, February 2005.

[68] Andre Merzky, Ole Weidner, and Shantenu Jha. SAGA: A standardized access layer to heterogeneous distributed computing infrastructure. *SoftwareX*, 2015. DOI: 10.1016/j.softx.2015.03.001.

[69] H Hellerman. Experimental personalized array translator system. *Communications of the ACM*, 7(7):433–438, July 1964.

[70] Daniele Turi, Paolo Missier, Carole Goble, David De Roure, and Tom Oinn. Taverna Workflows: Syntax and Semantics. In *Third IEEE International Conference on e-Science and Grid Computing (e-Science 2007)*, pages 441–448. IEEE, 2007.

[71] Tom Goodale, Shantenu Jha, Hartmut Kaiser, Thilo Kielmann, Pascal Kleijer, Andre Merzky, John Shalf, and Christopher Smith. A Simple API for Grid Applications (SAGA). OGF Recommendation, GFD.90, Open Grid Forum, 2007.

[72] A Sim et al. GFD.154: The Storage Resource Manager Interface Specification V2.2. Technical report, 2008. Global Grid Forum.

[73] GFAL2 utility tools web site. `https://dmc.web.cern.ch/projects/gfal2-utils`.

[74] Marvin github website. `https://github.com/marksantcroos/marvin`.

[75] Pykka github website. `https://github.com/jodal/pykka`.

[76] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. pages 235–245, August 1973.

[77] R Pordes, D Petravick, B Kramer, D Olson, M Livny, A Roy, P Avery, K Blackburn, T Wenaus, F Würthwein, I Foster, R Gardner, M Wilde, A Blatecky, J McGee, and R Quick. The open science grid status and architecture. *Journal of Physics: Conference Series*, 119, 2008.

[78] Ammar Benabdelkader, Mark Santcroos, Souley Madougou, Antoine H C van Kampen, and Silvia D Olabarriaga. A Provenance Approach to Trace Scientific Experiments on a Grid Infrastructure. In *ESCIENCE '11: Proceedings of the 2011 IEEE Seventh International Conference on eScience*, pages 134–141. IEEE Computer Society, December 2011.