SOFTWARE SYSTEMS FOR SUPPORTING SCIENTIFIC CAMPAIGNS ON HIGH PERFORMANCE & DISTRIBUTED COMPUTING RESOURCES

by

IOANNIS PARASKEVAKOS

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree of

Doctor of Philosophy

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Dr. Shantenu Jha and Dr. Matteo Turilli

and approved by

_____

_____

_____

_____

_____

New Brunswick, New Jersey

October, 2020

ABSTRACT OF THE DISSERTATION

Software Systems for Supporting Scientific Campaigns on High Performance & Distributed Computing Resources

by Ioannis Paraskevakos

Dissertation Director: Dr. Shantenu Jha and Dr. Matteo Turilli

Executing multiple workflows to achieve scientific insight is needed in computational sciences such as biomolecular sciences [1, 2], ecological sciences [3, 4] and particle physics [5]. The aggregated set of workflows that must be executed to achieve a computational objective is defined as a computational campaign.

This dissertation addresses the problem of effectively and efficiently executing a computational campaign on High Performance Computing (HPC) resources. Specifically, the dissertation focuses on computational campaigns with data- and compute-intensive workflows that utilize HPC resources at scale. Data-intensive workflows processes large amounts of data and their execution time heavily depends on I/O and data management. Compute-intensive workflows perform large amounts of computations with minimal I/O and therefore require little data management.

Currently, the execution of data-intensive workflows is not well supported on HPC resources. MapReduce is one of the most successfully abstraction used to execute data-intensive workflows at scale on cloud resources. However, implementing MapReduce on HPC resources is non-trivial and requires the use of additional abstractions. We utilize the Pilot abstraction as an integrating concept between HPC and MapReduce. We extend RADICAL-Pilot – a framework implementing the Pilot abstraction – to

support Hadoop – a framework implementing the MapReduce abstraction – on HPC resources. We experimentally characterize the execution time of data-intensive workflows and extension's overheads and show that Hadoop indeed reduces the execution time of such workflows on HPC resources.

MapReduce rests on task-parallelism to effectively and efficiently execute data-intensive workflows and several frameworks have emerged to support general-purpose task-parallelism. We experimentally investigate the suitability of three task-parallel frameworks for the execution of data-intensive workflows on HPC resources. Based on our experimental analysis, we provide a conceptual model to determine which framework is more suitable based on the characteristics of the selected data-intensive workflow. That conceptual model and the Pilot abstraction provide a methodology that application developers can use to maximize resource utilization while reducing the engineering effort needed to develop and execute data-intensive workflows on HPC resources.

In addition to data-intensive, workflows can also be compute-intensive, requiring different capabilities to effectively and efficiently utilize heterogeneous resources. Such capabilities can be implemented by multiple designs with different architectural and performance characteristics. Selecting a suitable design can significantly increase concurrency, resource utilization and reduce the overhead imposed by each design implementation. Nonetheless, choosing the right design can be challenging, especially in absence of an established methodology for characterizing and comparing alternative but functionally equivalent designs. We implement and experimentally characterize three functionally equivalent designs, showing which design approach is best suited for data- and compute-intensive workflows when executed on HPC resources at scale.

After establishing the methodology to effectively and efficiently execute data- and compute-intensive workflows on HPC, we investigate the support of computational campaigns. We elicit the requirements for a campaign manager from three scientific computational campaigns. Based on those requirements, we design and prototype a campaign manager. Our prototype is domain-agnostic and adheres to the building blocks design approach. The campaign manager prototype creates an execution plan and simulates the execution of a campaign on HPC resources.

As computational campaigns can utilize multiple HPC resources, they require an execution plan that efficiently and effectively map workflows on resources. In addition, computational campaigns may utilize HPC resources that may offer different computational capabilities, e.g., number of operations per second, and thus are heterogeneous. Further, HPC resources are dynamic as their performance changes over time for several reasons, and users offer a workflow runtime estimation which is uncertain at best.

Selecting a planning algorithm to derive an effective execution plan for a campaign on dynamic and heterogeneous HPC resources, while workflow runtime is uncertain, can be challenging. We investigate three algorithms to derive an execution plan for campaigns, characterizing their performance in terms of campaign makespan, and plan sensitivity to resource dynamism and workflow runtime estimation uncertainty. Based on our analysis, we provide a conceptual model for selecting a suitable planning algorithm based on characteristics of a computational campaign and HPC resources.

Our dissertation makes the following contributions: 1) Integrate MapReduce frameworks and HPC platforms to offer an unified environment for the effective and efficient execution of data-intensive workflows at scale. 2) Provide a conceptual model for selecting a task-parallel framework based on the workflow requirements, HPC resource capabilities and framework abstractions and performance. 3) Provide design guidelines to support the execution of data- and compute-intensive workflows on HPC resources at scale, alongside an experimental methodology to compare functionally equivalent designs. 4) Design and implementation of a campaign manager prototype to plan and simulate the execution of computational campaigns on multiple, heterogeneous and dynamic HPC resources. 5) Provide a conceptual model for selecting a planning algorithm to derive an execution plan based on campaign, resource heterogeneity and dynamism, workflow runtime uncertainty, and algorithm characteristics.

Acknowledgements

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Executing multiple workflows to achieve scientific insight is needed in computational sciences such as biomolecular sciences [1, 2], ecological sciences [3, 4] and particle physics [5]. The aggregated set of workflows that must be executed to achieve a computational objective is defined as a computational campaign.

For example, quantum chemistry campaigns [6] execute hundreds of workflows for months and on different platforms, such as workstations, campus and high performance computing (HPC) resources [6]. Ecological sciences analyze terabytes of very high resolution satellite imagery [3] to derive time series of ecological changes [4]. High Energy Physics experiments, such as the Large Hadron Collider ATLAS experiment [5], analyze petabytes of detector or simulation data [7] by executing campaigns with hundreds to thousands workflows, at different timescales and frequencies [7].

These use cases define and execute a computational campaign to achieve scientific insight. Although they serve different scientific purposes, computational campaigns share a number of characteristics and requirements: (1) they are comprised of O(100) workflows; (2) workflow requirements are heterogeneous across a campaign; (3) use multiple HPC platforms with heterogeneous and dynamic resources; (4) have finite allocations on said resources; and (5) have well defined computational objectives.

Based on the above discussion, it becomes important to effectively and efficiently support the execution of computational campaigns on HPC resources. This has the potential to increase overall resource utilization across all the resources that are available to a campaign. Further, increased resource utilization can allow computational scientists to execute computational campaigns to a larger scale, leading to new scientific discoveries and innovations. Thus, we focus our research on supporting the effective and efficient

execution of computational campaigns used to analyze data on HPC resources. It is important to note that our solutions and results are agnostic towards the scientific domain and size of the computational campaigns.

Computational campaigns can execute compute- and data-intensive workflows. While compute-intensive workflows are at the epicenter of high performance scientific computing, data-intensive workflows are not well supported on HPC resources. Compute-intensive workflows execute either a single, very large and long-running executable or an ensemble of smaller compute-intensive tasks [8]. Data-intensive workflows execute a large number of short-running and I/O, memory and compute bound tasks in multiple stages. As a result, a diverse set of abstractions (e.g., MapReduce and Resilient Distributed Datasets) and frameworks (e.g., Hadoop [9] and Spark [10]) have emerged to support the scalable execution of data-intensive workflows.

The MapReduce [11] abstraction has been successfully used to execute workflows to analyze data in different domains [12]. Utilizing the MapReduce abstraction on HPC platforms has the potential to increase resource utilization and reduce the time to completion of data analysis. However, data analysis frameworks that implement the MapReduce abstraction (e.g., Hadoop and Spark) mainly support data-intensive workflows on cloud platforms. At best, current solutions allow the use of such frameworks on HPC resources in isolation from the HPC environment and only for workflows with data-intensive requirements. There is therefore a need for abstractions that support frameworks like Hadoop and Spark on HPC resources as well as compute-intensive workflows. The goal is to provide a solution for executing computational campaigns, independent of the compute- or data-intensive workflows. To the best of our knowledge, there is no solution that provides the integrated capabilities of data analysis frameworks while preserving the ability to execute compute-intensive workflows.

Data analysis frameworks exploit task-parallelism to analyze data in a scalable and efficient manner. However, task-parallel frameworks offer different abstractions and capabilities. As workflows that analyze data can have different characteristics, alternative abstractions and capabilities may offer better performance and efficiency. In addition to performance gains, the implementation of abstractions and capabilities by a

framework has the potential to increase the development efforts of a workflow. As a result, there is a need to understand which framework is more suitable for performing a specific type of analysis while reducing the time developers invest to define the analysis workflow.

Some workflows, developed to analyze data, are both data- and compute-intensive and do not necessarily conform to data-parallel abstractions. The tasks of such workflows can be heterogeneous, implementing a diverse set of functionalities and compute-intensive. As such, frameworks (e.g., Spark) specifically designed for data parallelism may not support them well. Utilizing a task-parallel framework, though, may still be beneficial in terms of scalability, workflow execution time and resource utilization. However, the design of the framework can significantly affect performance and resource utilization. As a consequence, a specific design may not be well suited to execute workflows that are both data- and compute-intensive. In addition, the space of equivalent architectural designs to support such workflows is large and identifying the design that can efficiently execute data- and compute-intensive workflows becomes important. As a result, there is a need to understand which design is best suited to execute such workflows. Such an understanding will allow application developers to develop their workflows with a framework whose design is more suited for their application.

After establishing our understanding on how to effectively and efficiently support the execution of data- and compute-intensive workflows on HPC resources, we focus on supporting the execution of computational campaigns. Currently, the software systems that execute computational campaigns, called "campaign managers", make assumptions about resources and the middleware they are utilizing, have a monolithic design, and tend to be domain specific. As a consequence, these campaign managers cannot support computational campaigns from different scientific domains without significant changes in their implementation. In addition, due to the assumptions their designs make about the middleware, they cannot support workflows that do not have similar characteristics and are developed using different workflow execution frameworks. As a result, a campaign manager that supports the scalable execution of computational campaigns on HPC resources and is agnostic of the middleware used and scientific domain becomes necessary.

Finally, achieving a computational objective during the execution of a computational campaign requires an execution plan to map workflows on multiple HPC resources. However, actual HPC resources performance is affected by several factors and can change significantly during the execution of a campaign. In addition, users offer an estimation of each workflow runtime in the form of a range of possible values. As a result, the effectiveness of a plan is not guaranteed. It is therefore important for an execution plan to efficiently map the workflows of the campaign to resources, so as to increase resource utilization and achieve the objective of the campaign, despite any effect from HPC resource performance changes and runtime estimations. There is, thus, a need to understand how to derive such an execution plan and utilize a well suited planning algorithm. As there is a plethora of algorithms to derive an execution plan for a campaign [13]. To the best of our knowledge, there is no methodology to decide which algorithm is best suited given the characteristics of the campaign and resources.

In response to these challenges, we make contributions providing the necessary abstractions and software systems to efficiently execute data-driven workflows on HPC resources. Further, we design a campaign manager prototype which plans and executes computational campaigns with heterogeneous workflows on heterogeneous and dynamic HPC resources. Our work is designed and engineered to be domain and resource agnostic. The next section discusses in more detail the contributions of this dissertation.

1.1 Contributions and Organization of the Dissertation

The goal of this research is to advance the state of the art of the execution of data-intensive scientific computational campaigns on HPC platforms. To that extent, this dissertation makes the following contributions.

In chapter 2, we explore the integration between Hadoop, Spark and HPC resources, utilizing the Pilot abstraction [14, 15] to execute compute- and data-intensive workflows at scale. We evaluate that integration by measuring and characterizing the startup times of the Pilot and tasks. In addition, we measure the performance of a well-known algorithm to investigate the runtime trade-offs of a typical data-intensive workflow.

In chapter 3, we characterize and compare three task-parallel frameworks: Spark [10], Dask [16] and RADICAL-Pilot [17]. We asses their suitability for implementing molecular dynamics (MD) trajectory data analysis algorithms. MD trajectories are time series of atoms/particles positions and velocities, which are analyzed using different statistical methods to infer certain properties, e.g., the relationship among distinct trajectories, snapshots of a trajectory, etc. We characterize and explain the behavior of alternative MD analysis algorithms on the three frameworks and we provide a methodology to compare the abstractions, capabilities and performance of these frameworks.

In chapter 4, we focus on the design of computing frameworks that support the execution of data-driven, compute-intensive workflows on HPC resources. We provide specific design guidelines for supporting data- and compute-intense workflows on HPC resources with a task-based computing framework. We develop an experiment-based methodology to compare design performance of alternative designs that does not depend on the considered use case and computing framework.

In chapter 5, we design and implement a campaign manager prototype to plan and execute computational campaigns. We elicit three data analysis use cases to derive the functional requirements of the campaign manager. We provide a design and architecture, based on the buildings blocks [18] so that the manager is agnostic to the scientific domain of the workflows and the underlying middleware.

In chapter 6, we investigate three planning algorithms: HEFT [19], a genetic algorithm [20], and simple heuristic algorithm. Those algorithms enable planning the execution of a computational campaign on HPC resources. We develop an experiment-based methodology to compare the performance of planning algorithms, a methodology that does not depend on the considered use case, computing framework and resources used. In addition, we provide a conceptual framework for selecting planning algorithms based on the algorithm, campaign and resources characteristics.

Chapter 7.2 concludes the dissertation with a summary of our contributions and results. In addition, it identifies topics for further research and development.

Chapter 2

Data Analytics and High Performance Computing

Computational campaigns on High Performance Computing (HPC) resources can execute compute- or data- intensive workflows and applications. On one hand, compute-intensive applications and workflows are associated with either a single long running executable or an ensemble of compute-intensive tasks [8]. On the other hand, data-intensive applications are associated with multiple stages of execution which can be I/O, memory and compute bound. While MPI is the most common programming model for compute-intensive applications, the MapReduce [11] abstraction is commonly used for data-intensive applications [12].

Hadoop [9] popularized the use of MapReduce [11] for data-intensive applications. Hadoop's distributed filesystem (HDFS) automatically partitions data and distributes them to different nodes of a cluster machine. In addition, Hadoop's engine takes advantage of data-locality and transfers computations to nodes, allowing to process data in parallel while reducing data transfers. While Hadoop simplified the processing of vast volumes of data, it has limitations in its expressiveness [21, 22]. The complexity of creating sophisticated applications such as, for example, iterative machine learning algorithms [23], required multiple MapReduce jobs and persistence to Hadoop's Filesystem (HDFS) after each iteration. This led to devise several higher-level abstractions and the development of processing frameworks to implement sophisticated data pipelines.

Spark [10] is the most well-known processing framework in the Hadoop ecosystem. In contrast to MapReduce, Spark provides a richer API, more language bindings and a memory-centric processing engine that can utilize distributed memory and retain resources across multiple task generations. Spark's *Reliable Distributed Dataset (RDD)* [24] abstraction provides a powerful way to manipulate distributed collections stored in

cluster nodes' memory. Spark is used to build complex data workflows and advanced analytic tools, such as MLLib [25]. Although the addition/development of new and higher-level execution frameworks addressed some of the problems of data processing, it introduced the problem of heterogeneity of access and resource management.

Some computational campaigns cannot be easily classified as only compute- or data-intensive. For example, the simulations of contemporary bio-molecular dynamics [26] campaigns require increasing time scales and problem sizes, generating amounts of data several order of magnitude larger than those so far generated. Further, data generated by those simulations often need to be analyzed so as to determine the next set of simulation configurations. Several tools evolved to meet the demand for molecular dynamics data analysis, such as MDAnalysis [27, 28], CPPTraj [29] and HiMach [30]. Although these tools offer domain-specific analytics, scaling them to high data volumes remains a challenge, especially when the scale of the computation requires the use of high performance computing (HPC) infrastructures. There is therefore a need for computing environments that support scalable data processing while preserving the ability to run simulations at large scale. Utilizing existing frameworks that support the MapReduce abstraction on HPC resources has the potential to provide scalable data processing. In addition, it will significantly reduce engineering time as there will not be a need to implement these capabilities from scratch. To the best of our knowledge, there does not exist a solution that provides the integrated capabilities of Hadoop and HPC.

In this chapter, we explore the integration between Hadoop [9], Spark [10] and HPC resources. We utilize the Pilot-Abstraction [14], allowing to manage HPC and data-intensive applications in a uniform way. We explore two extensions to RADICAL-Pilot [31], a Pilot-Job [14] runtime system designed for implementing task-parallel applications on HPC: 1) the ability to spawn and manage Hadoop/Spark clusters on HPC infrastructures on demand (Mode I); and 2) to connect and utilize Hadoop and Spark clusters for HPC applications (Mode II) . Both extensions facilitate the application and resource management requirements of data-intensive applications. By supporting these two usage modes, RADICAL-Pilot simplifies the barrier of deploying and executing HPC and Hadoop/Spark workflows side-by-side.

The chapter is organized as follows: In section §2.1 we discuss the current solutions and challenges of integrating data analytics and HPC. We continue with discussing the modes of integration and interoperability between data analytics and HPC in §2.2 and the experimental validations of the integration in §2.3. The chapter closes with our conclusions in section §2.4.

2.1 Related Work and Challenges

There are several frameworks which explored the usage of Hadoop on HPC resources, e.g., Hadoop on Demand [32], JUMMP [33], MagPie [34], MyHadoop [35] and MyCray [36]. These frameworks provide users with a set of scripts that can spawn and manage a Hadoop cluster on HPC resources. As soon as Hadoop is started, users can either submit their MapReduce jobs interactively or via a script.

Although these frameworks can spawn and manage Hadoop clusters, they isolate the Hadoop cluster from the HPC environment. The scripts and capabilities they offer interact directly with the HPC's submission system. As a result, users cannot execute on the same set of nodes HPC and Hadoop applications. Further, they do not necessarily optimize configurations and resource usage, including the use of available SSDs, parallel filesystems and high-end network features, e.g., RDMA [37].

Hadoop originally provided a rudimentary resource management system called the YARN scheduler. The YARN scheduler [38] provides an application-level scheduling framework, addressing the increased requirements with respect to applications and infrastructure, such as data localities (memory, disk, datacenter), long-lived services, periodic jobs, and interactive and batch jobs. In contrast to batch schedulers of HPC resources, YARN is optimized for supporting data-intensive environments and managing a large number of fine-grained tasks.

While YARN manages system-level resources, applications need to implement application-level scheduling that optimizes their specific requirements. This is implemented by an application-level scheduler, the *Application Master*, responsible for allocating resources—called containers—and executing tasks on these resources. In

addition, Application Master manages data locality, for example between HDFS blocks and container locations, by allocating containers on specific nodes.

Managing resources on top of YARN poses several challenges. While applications with fine-grained, short-running tasks are well supported, applications with coarse-grained, long-running task, such as MPI applications, are not. To achieve interoperability and integration between Hadoop and HPC, it is essential to consider a more diverse set of applications on top of YARN.

A particular challenge for Hadoop on HPC deployment is the choice of storage and filesystem backend. Typically, when using Hadoop local storage is preferred. Nevertheless, in some cases, like when processing many small files or when the number of parallel tasks is low to medium, random data access is required. In those cases, using a parallel filesystem can yield a better performance. For this purpose, many parallel filesystems provide a special client library which improves the interoperability with Hadoop, but limits data locality and any data placement optimization.

Another challenge is the the integration between HPC and Hadoop environments. Rather than preserving HPC and Hadoop "environments" as software silos, there is the need for an approach that integrates them. We utilize the Pilot-Abstraction as a unifying concept to efficiently support that integration, and not just the interoperability between HPC and Hadoop. By utilizing the multi-level scheduling capabilities of YARN, the Pilot-Abstraction can efficiently manage Hadoop resources, providing an application with the means to reason about data, compute resources and allocation. In addition, we show how the Pilot-Abstraction can be used to manage Hadoop applications on HPC environments.

The Pilot-Abstraction [14] is successfully used on HPC resources supporting a diverse set of task-based workflows. A Pilot-job is a placeholder job which is submitted to the resource management system and represents a container for a dynamic set of tasks. Pilot-jobs are, also, a well-known example of multi-level scheduling, which is often used to separate system-level resource and user-level workload management. The Pilot-Abstraction defines the following entities: 1) a Pilot which allocates a set of computational resources (e.g., cores); and 2) a Compute-Unit (CU) as a self-contained

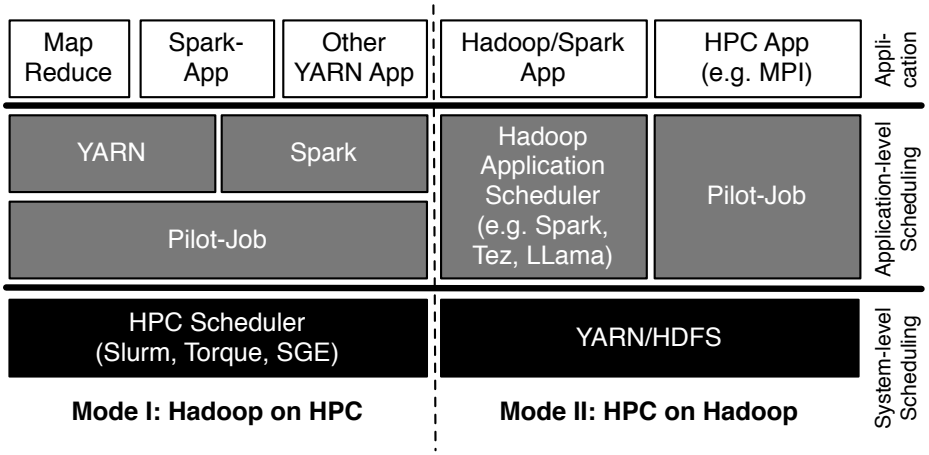


Figure 2.1: Hadoop and HPC Interoperability modes

piece of work represented as an executable. The Pilot-Abstraction has been implemented within RADICAL-Pilot [31].

2.2 Data analytics and HPC integration: RADICAL-Pilot and YARN/Spark

Having established the potential of the Pilot abstraction for a range of high-performance applications [39, 40, 41], we use that abstraction as the starting point for integrating HPC and data-intensive analytics. As depicted in Figure 2.1, there are at least two different usage modes to consider: 1) Mode I: Running Hadoop/Spark applications on HPC environments (Hadoop on HPC), 2) Mode II: Running HPC applications on YARN clusters (HPC on Hadoop).

Mode I is critical to support traditional HPC environments so as to execute applications with both compute and data requirements. Mode II is important for cloud environments (e.g., Amazon's Elastic MapReduce) and an emerging class of HPC machines with new architectures and usage modes, such as Wrangler [42], that support Hadoop natively. For example, Wrangler supports dedicated Hadoop environments (based on Cloudera Hadoop 5.3) via a reservation mechanism. Using these new capabilities, applications can seamlessly connect HPC stages (e.g., simulation stages) with analysis stages, using the Pilot-Abstraction to provide unified resource management.

With the introduction of YARN, Hadoop can execute a broader set of applications

than before. However, developing and deploying YARN applications, potentially side-by-side with HPC applications, remains a difficult task. Established abstractions which are easy to use and that enable users to reason about compute and data resources across infrastructure types (i.e., Hadoop, HPC and clouds) are missing.

Schedulers such as YARN facilitate application-level scheduling but the development effort required by YARN applications is still high. YARN provides a low-level abstraction for resource management, e.g., a Java API and protocol buffer specification. Typically, interactions between YARN and applications are much more complex than the interactions between an application and an HPC scheduler. Further, applications must be able to run on a dynamic set of resources, e.g., YARN may have to preempt containers in high-load situations. Data/compute locality requires to be manually managed by the application scheduler by requesting resources at the location of a file chunk.

To address these shortcomings, various frameworks that aid the development of YARN applications have been proposed. Llama [43] offers a long-running application master for YARN designed for the Impala SQL engine. Apache Slider [44] supports long-running distributed application on YARN with dynamic resource needs allowing applications to scale to additional containers on demand. While these frameworks simplify development, they do not address concerns such as interoperability and integration of HPC/Hadoop. Integrating YARN and RADICAL-Pilot (RP) allows applications to run HPC and YARN applications on HPC resources concurrently.

RADICAL-Pilot and YARN Overview

Figure 2.2 illustrates the architecture of RP and the components that were extended to integrate YARN. The figure on the left shows the macro architecture of RP, while the figure on the right shows the architecture of the RP-Agent component. RP consists of a client module with the Pilot-Manager and Unit-Manager components and a set of RP-Agent components running on one or more resources. Pilot-Manager is responsible for managing a set of pilots. Pilots are described via pilot descriptions, which contain the pilot's resource requirements. Pilot-Manager submits a placeholder job which will run the RP-Agent via the Resource Management System using the SAGA-API [45] (steps

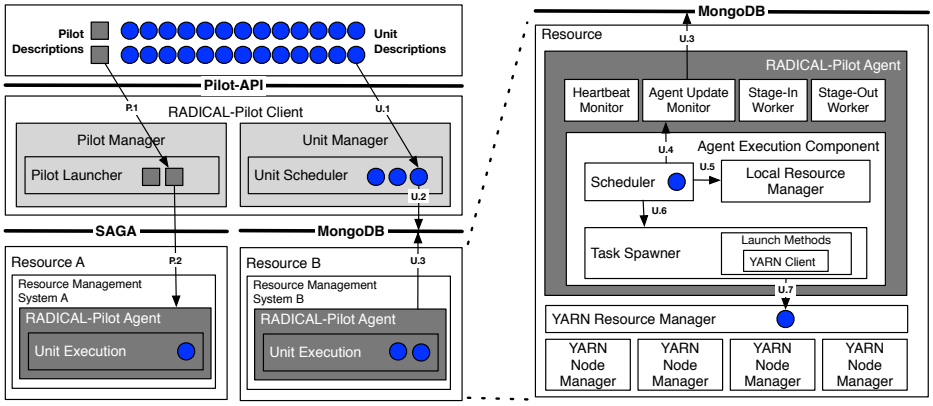


Figure 2.2: **RADICAL-Pilot and YARN Integration:** There are two main interactions between the application and RADICAL-Pilot: the management of Pilots (P.1–P.2) and the management of Compute Units (U.1–U.7). All YARN specifics are encapsulated in the RP-Agent.

P.1–P.2). Subsequently, the Unit-Manager and the RP-Agent manage the execution of the application workload (the Compute-Units, steps U.1–U.7) [17].

RP-Agent has a modular and extensible architecture, and consists of the following components: Agent Execution Component, Heartbeat Monitor, Agent Update Monitor, and Stage In and Stage Out Workers. The main integration of YARN and RP is done in Agent Execution Component. The Agent Execution component consist of four sub-components: a) the *scheduler* is responsible for monitoring the resource usage and assigning CPUs/GPUs to CUs; b) the *Local Resource Manager* (LRM) interacts with the batch system and communicates to the pilot the available number of computing resources and how they are distributed; c) the *Task Spawner/ Executor* configures the execution environment, executes and monitors each unit; and d) the *Launch Method* encapsulates the environment specifics for executing an application, e.g., the usage of `mpiexec` for MPI applications, machine-specific launch methods (e.g. `aprun` on Cray machines) or the usage of YARN. After a unit completes its execution, the Task Spawner collects the exit code, standard output, and informs the scheduler about the freed cores.

RADICAL-Pilot and YARN Integration

There are two main approaches to the integration of RP and YARN: (i) Integration at the Pilot-Manager level via a SAGA adaptor; and (ii) integration at the RP-Agent

level. The first approach poses several challenges, among which the most difficult to address is that, typically, firewalls prevent the communication between external machines and a YARN cluster. A YARN application is not only required to communicate with the resource manager, but also with the node managers and containers. Further, this approach would require significant extension of Pilot-Manager, which currently relies on the SAGA Job API for launching and managing pilots. Capabilities required by YARN, like on-demand provisioning of a YARN cluster and application resource management, are very difficult to implement via the SAGA API.

The second approach encapsulates YARN specifics at resource-level, enabling the decentralized provision of a YARN cluster. Units are scheduled and submitted to the YARN cluster via the Unit-Manager and RP-Agent scheduler. By integrating RP and YARN at the RP-Agent level, RP can support both Mode I and II, as outlined in Figure 2.1.

As illustrated in Figure 2.2, RP (steps P.1–P.2) starts on the remote resource using SAGA. In Mode I, during the launch of RP-Agent a YARN cluster is spawned on the allocated resources (Hadoop on HPC), while in Mode II, RP-Agent connects to a YARN cluster that runs on the resources managed by RP-Agent. Once started, RP-Agent accepts CUs submitted via Unit-Manager (step U.1). Unit-Manager queues new CUs using a shared MongoDB (step U.2) database. RP-Agent periodically checks for new CUs (step U.3) and queues them in the scheduler (step U.4). Finally, the Executor manages the execution of the CUs (step U.6 and U.7).

The *Local Resource Manager (LRM)* provides an abstraction to local resource details for other components of the RP-Agent. The LRM evaluates the environment variables provided by the resource management systems to obtain information, such as the number of cores per node, memory and the assigned nodes. This information can be accessed through the Resource Manager's API. In Mode I (Hadoop on HPC), during the initialization of the RP-Agent, the LRM setups the HDFS and YARN daemons. First, the LRM downloads Hadoop and creates the necessary configuration files, i.e., the `mapred-site.xml`, `core-site.xml`, `hdfs-site.xml`, `yarn-site.xml` and the slaves and master file containing the allocated nodes. The node that is running the Agent,

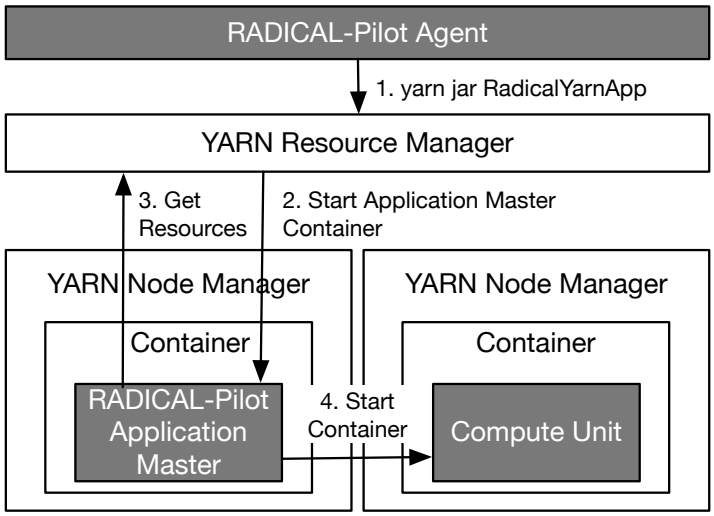


Figure 2.3: RADICAL-Pilot YARN Agent Application.

runs the master daemons of Hadoop: the HDFS Namenode and the YARN Resource Manager. As soon as HDFS and YARN area active, the scheduler receives meta-data about the cluster, i.e., the number of cores and memory. HDFS and YARN remain active until all tasks are executed. Before termination of the agent, the LRM stops the Hadoop and YARN daemons and removes the associated data files. In Mode II (Hadoop on HPC), the LRM solely collects the Hadoop cluster resource information.

The *scheduler* is another extensible component of the RP-Agent responsible for queuing CUs and assigning them to resources. To support YARN, we created a special scheduler that utilizes the cluster state information (e.g., the amount of available cores, memory, queue information, application quotas etc.) obtained via the Resource Manager's REST API of YARN. In contrast to other RP schedulers, our scheduler specifically utilizes memory in addition to cores for assigning resources to CUs.

The *Task Spawner* is responsible for managing and monitoring the execution of a CU. The *Launch Method* component encapsulates resource/launch-method specific operations, e.g., the usage of the `yarn` command line tool for submitting and monitoring applications. After the launch of a CU, Task Spawner periodically monitors its execution and updates its state in a shared MongoDB instance by utilizing the application log file.

RADICAL-Pilot Application Master: YARN's multi-step resource allocation process depicted in Figure 2.3 differs significantly from HPC schedulers and, as such, poses an

integration challenge. The central component of a YARN application is the Application Master, which negotiates resources assignment with the YARN Resource Manager and manages the execution of an application in the assigned resources. The unit of allocation in YARN is a container [46] (not to be confused with a Docker container). The YARN client is part of the YARN Launch Method and implements a YARN Application Master, which is the central instance for managing the resource requirements of an application. RP utilizes a managed application master that runs inside a YARN container. Once the Application Master container is started, the managed application master requests a YARN container that meets the resource requirements of the CU from the YARN's Resource Manager. Once YARN allocates a container, the CU starts inside that container. A wrapper script sets up a RP environment, staging the specified files and running the executable defined in the CU Description. Every Compute Unit is mapped to a YARN application consisting of an Application Master and a container of the size specified in the CU Description.

RADICAL-Pilot and Spark Integration

Spark offers multiple deployment modes: standalone, YARN and Mesos. While it is possible to support Spark on top of YARN, the YARN and Mesos approaches incur into significant complexities and overheads as they require for two frameworks to be configured and bootstrapped. Since RP operates in user-space and single-user mode, no advantages with respect to using a multi-tenant YARN cluster environment exist. Thus, we support Spark via the standalone deployment mode.

Similar to the YARN integration, the necessary changes for Spark are confined to the RP-Agent. Local Resource Manager (LRM) initializes and deploys the Apache Spark environment. In the first step, LRM detects the number of cores, memory and nodes provided by Resource Management System, and verifies and downloads necessary dependencies (e.g., Java, Scala, and Spark binaries). In the second step, LRM creates the necessary configuration files to run a multi-node, standalone Spark cluster (e.g., `spark-env.sh`, `slaves` and `master` files). Finally, LRM starts a Spark cluster, using the previously generated configuration.

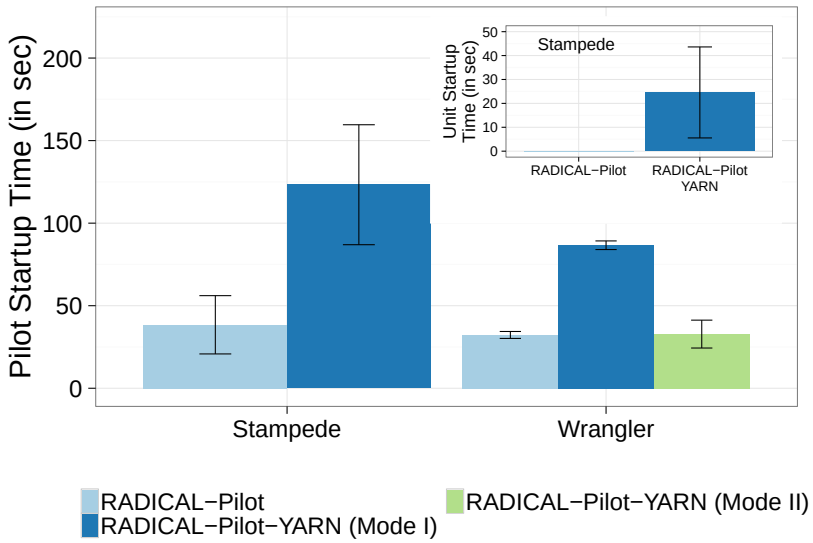


Figure 2.4: RADICAL-Pilot and RADICAL-Pilot-YARN startup overheads.

Similar to YARN, a Spark RP-Agent scheduler is used for managing Spark resource slots and assigning CUs. During the termination of the RP-Agent, LRM shuts down the Spark cluster using Spark's termination script, which stops both the master and the slave nodes. Similarly, the Spark specific methods for launching and managing CUs on Spark are encapsulated in a Task Spawner and Launch Method component.

2.3 Experiments and Evaluation

To evaluate the RADICAL-Pilot-YARN (RP-YARN) and Spark extension, we conduct two experiments. In §2.3.1, we analyze and compare RP and RP-YARN with respect to startup times of both the Pilot and the CUs. In §2.3.2, we use the well-known K-Means algorithm to investigate the performance and runtime trade-offs of a typical data-intensive application. We performed our experiments on two XSEDE machines: Wrangler [42] and Stampede [47]. On Stampede, every node has 16 cores and 32 GB of memory; on Wrangler 48 cores and 128 GB of memory. For our experiments, we used RP v0.45, Hadoop v2.6 and Spark v2.0.2.

2.3.1 Pilot Startup and Compute Unit Submission Characterization

In Figure 2.4 we analyze the measured overheads when starting RP and RP-YARN, and when submitting CUs. The agent startup time for RP-YARN is defined as the time between when RP-Agent starts and when the first CU starts executing. On Wrangler, we compare both Mode I (Hadoop on HPC) and Mode II (HPC on Hadoop). For Mode I, the startup time is higher compared to the startup time of RP and of Mode II. This can be explained by the necessary steps required to download, configure and start the YARN cluster. For a single node YARN environment, the overhead for Mode I (Hadoop on HPC) is between 50-85 sec, depending on the selected resource. The startup times for Mode II on Wrangler—using the dedicated Hadoop environment provided via the data portal—are comparable to the normal RP startup times as RP does not spawn a Hadoop cluster.

The inset of Figure 2.4 shows the time taken to start Compute Units via RP to a YARN cluster. For each CU, resources have to be requested in two stages: first the application master container is allocated and, second, the containers for the actual compute tasks becomes available. For short-running jobs, that represents a bottleneck. In summary, while there are overheads associated with executing CUs inside of YARN, we believe these are acceptable, in particular for long-running CUs.

2.3.2 K-Means Performance Characterization

We compare the time to completion of the K-Means algorithm using two configurations: RP on HPC and RP in HPC/YARN mode. We use three different scenarios: 10,000 points and 5,000 clusters, 100,000 points and 500 clusters, and 1,000,000 points and 50 clusters. Each point belongs to a three dimensional space. The compute requirements depend on the product of the number of points and clusters, thus the requirements are constant for all three scenarios. We use an optimized version of K-Means in which the sum and number of points are precomputed in the map phase, thus only these two attributes per cluster need to be shuffled. The shuffle traffic depends on the number of clusters and decreases with the number of clusters. For the purpose of this benchmark,

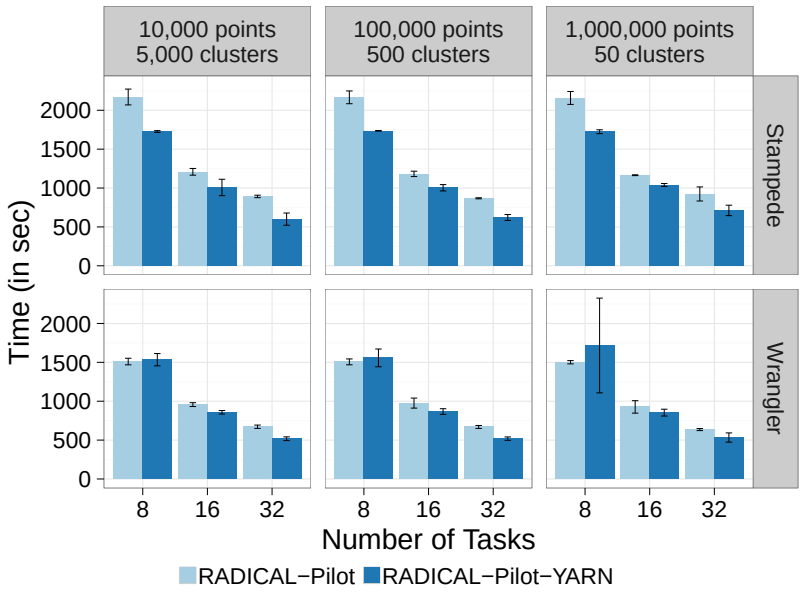


Figure 2.5: RADICAL-Pilot and YARN-based K-Means time to completion on Stampede and Wrangler.

we run two iterations of K-Means.

We utilize up to 3 nodes on Stampede and Wrangler. We use the following configurations for our experiments: 8 tasks on 1 node, 16 tasks on 2 nodes and 32 tasks on 3 nodes. For RP-YARN, we use Mode II (Hadoop on HPC): the YARN Resource Manager is deployed on the node running the RP-Agent.

Figure 2.5 shows the results of executing K-Means over different scenarios and configurations. For RP-YARN, runtimes include the time required to download and start the YARN cluster on the allocated resources.

Independent of the scenario, runtimes decrease with the number of tasks. In particular, for the 8 task scenarios the overhead of YARN on Wrangler is enough to produce higher runtimes for RP-YARN than RP. For larger number of tasks, we observed on average 13 % shorter runtimes for RP-YARN. Also, RP-YARN achieves better speedups, e.g., 3.2 for 32 tasks for the 1 million points scenario, which is significantly higher than the RP speedup of 2.4 (both on Wrangler and compared to base case of 8 tasks). One of the reasons of that performance difference, is that RP-YARN uses the local file system, while RP uses the Lustre filesystem.

For similar scenarios and task/resource configuration, the runtimes show a significant performance improvement on Wrangler over Stampede. This is attributed to the better hardware (CPUs, memory) of Wrangler than Stampede. In particular, for RP-YARN we observed on average higher speedups on Wrangler, indicating that we saturated the 32 GB of memory available on each Stampede node.

In summary, despite the overheads of RP-YARN with respect to Pilot and CU startup time, we were able to observe performance improvements (on average 13 % better time to completion than RP) mainly due to the better performance of the local disks.

2.4 Conclusions

An increasing number of scientific applications use Hadoop and Spark mainly due to the accessible abstractions they provide. HPC and Hadoop environments are converging: For example, MLLib/Spark utilize BLAS, and Hadoop frameworks adopt parallel and in-memory computing concepts that originated in HPC. Currently, traditional HPC applications lack the ability to use Hadoop and other Hadoop tools, without sacrificing the advantages of HPC environments. One prominent reason for the limited uptake of Hadoop-based technologies on HPC infrastructures is the lack of effective, scalable and usable resource management techniques.

This chapter shows how the Pilot abstraction can be used as an integrating concept, discussing the design and implementation of RP extensions for Hadoop and Spark, and validating those extensions with scalability analysis on two HPC infrastructures. The Pilot abstraction enables users to combine the capabilities of best-of-breed tools which run on heterogeneous HPC infrastructures. Using these capabilities, data-intensive computational campaigns can utilize diverse Hadoop and HPC frameworks, enabling scalable data workflows and analytics pipelines.

In this chapter we demonstrated the importance of integrated capabilities for supporting data-intensive workflows on HPC infrastructures. In the next chapter, we move forward by characterizing the performance of those capabilities for paradigmatic

molecular dynamics workflows. When executing computational campaigns on HPC resources is paramount to understand the performance tradeoffs among application, middleware, executable and system capabilities.

Chapter 3

Task-Parallel Data Analytics on HPC Platforms

Molecular Dynamics campaigns are significant consumer of computing cycles and produce immense amounts of data, especially during simulation stages. Simulation workflows execute up to 10^5 μsec MD simulations of a $O(100k)$ atoms physical system, producing from $O(10)$ to $O(1000)$ GBs of data [1]. Increasingly, there is a need for analyses to be integrated with simulations to drive the next stages of the workflow execution at runtime [48]. Analyses can benefit from task-level parallelism as they can be partitioned into independent units of work.

Task-parallel applications involve partitioning a workflow into a set of self-contained units of work. Based on the application, these tasks can be independent, have no inter-task communication, or coupled with varying degrees of data dependencies. Data-intensive applications exploit task parallelism for data-parallel operations (e.g., `map`), but also require coupling, for computing aggregates (`reduce`). Typically, a reduce operation includes shuffling intermediate data from a set of nodes to node(s) where the reduce executes.

Data analytics workflow engines exploit task-parallelism by partitioning the data and generating stages of independent tasks. These stages then implement data dependencies by shuffling, aggregating or coupling intermediate results. The MapReduce [11] abstraction, along with its implementations, popularized this method of processing.

Spark [10] and Dask [16] are two MapReduce frameworks. Both provide abstractions which reason about data and are optimized for parallel processing of large data volumes, interactive analytics and machine learning. Their runtime engines can automatically partition data, generate parallel tasks, and execute them on a cluster. In addition, Spark offers in-memory capabilities allowing data caching data, making it suited for

interactive analytics and iterative machine learning algorithms. Dask also provides a MapReduce API (Dask Bags). Furthermore, Dask's API is more versatile, allowing custom workflows and parallel vector/matrix computations.

As task-parallel frameworks offer different abstractions and capabilities, executing data analysis workflows in an efficient and scalable manner remains a challenge. As data analysis applications can have different characteristics (e.g., embarrassingly parallel or MapReduce), alternative abstractions and capabilities may offer varying performance and efficiency. As a result, there is a need to understand which framework is more suitable for performing a specific type of analysis.

In this chapter, we investigate three task-parallel frameworks and their suitability for implementing MD trajectory analysis algorithms. In addition to Spark and Dask, we investigate RADICAL-Pilot (RP) [17]. We utilize MPI4py [49] to provide MPI equivalent implementations of the algorithms. The task-parallel implementations performance and scalability compared to MPI is the basis of our analysis. MD trajectories are time series of atoms/particles positions and velocities, which are analyzed using different statistical methods to infer certain properties, e.g., the relationship between distinct trajectories, snapshots of a trajectory etc.

The chapter is organized as follows: §3.1 discusses several approaches to support scalable MD trajectory data analytics. §3.2 describes the MD analysis algorithms investigated and reviews different MD analysis frameworks with respect to their ability to support scalable analytics of large volumes of MD trajectories. §3.3 describes RP, Spark and Dask, the three frameworks used for our performance comparison and evaluation. §3.4 provides a description of the implementation of the MD algorithms on top of those three frameworks, as well as a performance evaluation and a discussion of our findings. In §3.5, we provide a conceptual analysis that allows application developers to select a framework according to their requirements. Finally, §3.6 closes this chapter with our conclusions.

3.1 Related Work

Until recently, MD analysis algorithms were executed serially and parallelization was not straightforward. During the last years, several frameworks emerged providing parallel algorithms for analyzing MD trajectories. Some of those frameworks are CPPTraj [29, 50], HiMach [30], PMDA [51], Pteros 2.0 [52] and nMoldyn-3 [53].

Several techniques are used for parallelizing MD analysis algorithms. CPPTraj [50] utilizes MPI and OpenMP to execute large-scale analysis on HPC. OpenMP is also utilized by Pteros [52] to parallelize the compute intensive parts of the analysis. HiMach [30] extends Google's MapReduce and defines Map and Reduce methods. nMoldyn-3 [53] parallelizes the execution through a Master/Worker architecture. The master defines analysis tasks which are then executed by a set of worker processes.

A common denominator of most approaches is that they do not use general purpose frameworks for parallelizing the execution. Although they are optimized to get as much performance as possible from the environments they are developed for, their portability is limited. For example, CPPTraj [50] and Pteros [52] are highly dependent on the low level libraries of the resource they use, while HiMach [30] is build specifically for the Antons Supercomputer.

In contrast, our analysis focuses on frameworks that offer more general purpose approaches to the parallelizing MD analysis algorithms. These frameworks provide higher-level abstractions that facilitate the integration with other data analysis methods. In addition, resource acquisition and management is done transparently.

3.2 Molecular Dynamics (MD) Data Analysis

Root Mean Square Deviation (RMSD), Pairwise Distances (PD), and Sub-setting [54] are algorithms commonly used to analyze MD trajectories. Path Similarity Analysis (PSA) [55] and Leaflet Identification [27] are two more advanced algorithms. All these methods read and process a physical system generated via multiple simulations, reducing data to either a single number or a matrix. RMSD identifies the deviation of atom positions among frames, while PD and PSA calculate distances between atoms or

trajectories based on different metrics. Sub-setting methods are used instead to isolate parts of interest of a MD simulation, and Leaflet Identification provides information about groups of lipids by identifying the lipid leaflets in a lipid bilayer.

We discuss two of those methods—a PSA algorithm that uses the Hausdorff distance and a Leaflet identification algorithm—implementated in MDAnalysis [27, 28]. MDAnalysis is a Python library [27, 28] that provides a comprehensive environment for filtering, transforming and analyzing MD trajectories. MDAnalysis provides a common object-oriented API to trajectory data and leverages existing libraries of the scientific Python software stack, such as NumPy [56] and Scipy [57].

Path Similarity Analysis (PSA): Hausdorff Distance

Path Similarity Analysis (PSA) [55] quantifies the similarity between trajectories, considering their full atomic detail. The basic idea is to compute pair-wise distances (e.g., using the Hausdorff metric [58]) between members of an ensemble of trajectories, and cluster the trajectories based on their distance matrix. Each trajectory is represented as a two dimensional array: The first dimension corresponds to time frames of the trajectory; the second to the N atom positions in a 3-dimensional space. Algorithm 1 describes PSA with the Hausdorff metric over multiple trajectories. We apply a 2-dimensional data partitioning over the output matrix to parallelize, shown in algorithm 2.

The algorithm is embarrassingly parallel, has complexity $O(n^2)$ and its input data volume is medium to large while the output is small. Embarrassingly parallel algorithms are good candidates for task parallelization. Each task analyzes a partition of the dataset. Given enough resources, tasks can execute concurrently as a bag of tasks, using a task management API or a map-only application in a MapReduce-style API. Spark, Dask and RP support the execution of bag of tasks, with RP and Dask offering specific abstractions (as discussed in §3.3).

Leaflet Finder

Algorithm 3 describes the Leaflet Finder (LF) algorithm as presented in Ref. [27]. LF assigns particles to one of two curved but locally approximately parallel sheets,

Algorithm 1 Path Similarity Algorithm: Hausdorff Distance

procedure HAUSDORFFDISTANCE(T_1,T_2) ▷ T_1 and T_2 are a set of 3D points
  `List` D_1,D_2
  for $\forall frame_1$ in T_1 **do**
    for $\forall frame_2$ in T_2 **do**
      `Append in` D_1 d_{RMS}(`frame`$_1$, `frame`$_2$)
    end for
    D_{t_1} `append` $min(D_1)$
  end for
  for $\forall frame_2$ in T_2 **do**
    for $\forall frame_1$ in T_1 **do**
      `Append in` D_2 d_{RMS}(`frame`$_2$, `frame`$_1$)
    end for
    D_{t_2} `append` $min(D_2)$
  end for
  return $max\Big(max(D_{t_1}), max(D_{t_2})\Big)$
end procedure

procedure PSA($Traj$) ▷ $Traj$ is a set of trajectories
  for $\forall T_1$ in $Traj$ **do**
    for $\forall T_2$ in $Traj$ **do**
      $D_{(T_1,T_2)}$=`HausdorffDistance`$\Big(T_1, T_2\Big)$
    end for
  end for
  return D
end procedure

Algorithm 2 Two Dimensional Partitioning

1: Initially, there are N^2 distances, where N is the number of trajectories. Each distance defines a computation task.
2: Map the initial set to a smaller set with $k = N/n_1$ elements, where n_1 is a divisor of N, by grouping n_1 by n_1 elements together.
3: Execute over the new set with k^2 tasks. Each task is the comparisons between n_1 and n_1 elements of the initial set. They are executed serially.

provided that the inter-particle distance is smaller than the distance between sheets. In biomolecular simulation data analysis of lipid membranes, consisting of a double layer of lipid molecules, LF identifies the lipids of the outer and inner leaflets (sheets). The algorithm consists of two stages: a) construction of a graph connecting particles based on threshold distance (cutoff); and b) computing the connected components of the graph, determining the lipids located on the outer and inner leaflets.

The application stages have different complexities. The first stage identifies neighboring atoms. There are two alternative implementations: i) computing the distance between all atoms ($O(n^2)$); ii) utilizing a tree-based nearest neighbor (Construction: $O(n \log n)$, Query: $O(\log n)$). In both alternatives, the input data volume is medium size and the output is smaller than the input. The complexity of connected components is: $O(|V| + |E|)$ (V: Vertices, E: Edges), i.e., it greatly depends on the characteristics of the graph.

Algorithm 3 Leaflet Finder Algorithm

```
1: procedure LeafletFinder(Atoms, Cutoff)  ▷ Atoms is a set of 3D points that represent the position of atoms in space. Cutoff is an Integer Number
2:     Graph G =(V = Atoms, E = ∅)
3:     for ∀atom in Atoms do
4:         N = [a ∈ V : d(a, atom) ≤ Cutoff]
5:         Add edges [(atoms, a) : a ∈ N] in G
6:     end for
7:     C = ConnectedComponents(G)
8:     return C
9: end procedure
```

LF is more complex than PSA as it requires two stages. It is possible to implement LF with a simple task-management API, although the MapReduce programming model allows for a more efficient implementation with a `map` for computing and filtering distances and a `reduce` for finding the components. The shuffling required between map and reduce is medium as the number of edges is a fraction of the input data. Spark and Dask natively support MapReduce and implementing LF is relatively straightforward. RP, on the other hand, does not natively support MapReduce. As a result, the application developer has to define the data shuffling between the *map* and the *reduce* phases of the algorithm.

3.3 Task-Parallel Frameworks: Spark, Dask and RADICAL-Pilot

The landscape of frameworks for data-intensive applications is manifold [59, 60] and has been extensively studied in the context of scientific [61] applications. In this section, we discuss the suitability of task-parallel frameworks such as Spark [10], Dask [16] and RP [17] for MD data analytics.

3.3.1 Background

Spark and Dask provide APIs, caching and other capabilities that are critical to develop analytics applications. Spark is considered the standard solution for iterative data-parallel applications. Dask is quickly gaining support by the scientific community, since it offers a sought-after Python environment. RP also offers a Python-only programming API, supporting task-level parallelism on HPC resources. In this way, RP adds parallelization capabilities to MPI-based applications, enabling the concurrent and

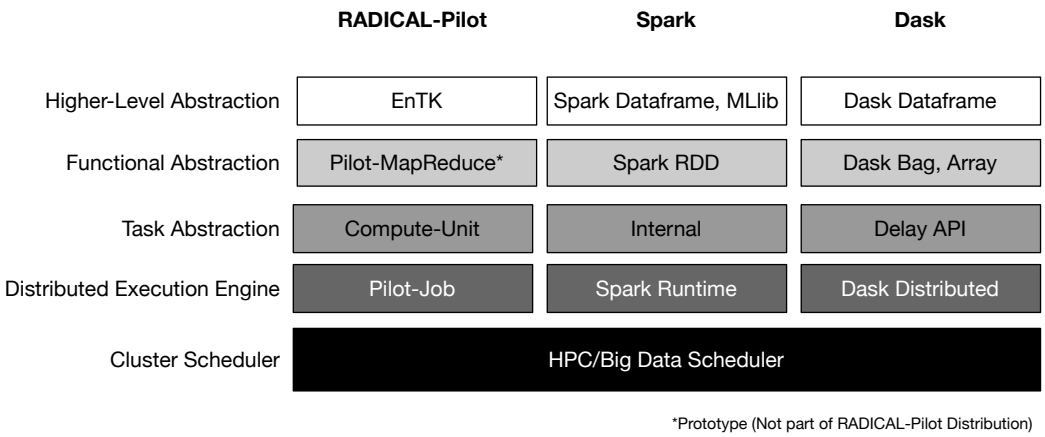


Figure 3.1: Architecture of RADICAL-Pilot, Spark and Dask.

sequential execution of bag of MPI tasks on HPC resources.

As described in [59], these frameworks typically comprise of distinct layers, e.g., cluster scheduler access, framework-level scheduling, and higher-level abstractions. Various higher-level abstractions can be provided on top of low-level resource management capabilities, e.g., MapReduce-inspired APIs. These capabilities provide the foundation for analytics abstractions, such as Dataframes, Datasets and Arrays. Figure 3.1 visualizes the components of RP, Spark and Dask.

RADICAL-Pilot

As discussed in §2, RP allows concurrent task execution on HPC resources. The user defines a set of Compute-Units (CU) which are submitted to RP. RP schedules these CUs for execution on the acquired resources and uses the existing environment of the resource to execute tasks. Any data communication between tasks is done via an underlying shared filesystem, e.g., Lustre. Task execution coordination and communication is done through a database (MongoDB).

Spark

Spark [10] extends MapReduce [11], providing a rich set of operations on top of the Resilient Distributed Dataset (RDD) abstraction [24]. RDDs are cached in-memory, making Spark well suited for iterative applications that need to cache a set of data

across multiple stages. PySpark provides a Python API to Spark.

A Spark workflow consists of multiple stages. Each stage is a set of independent parallel tasks (e.g., `map`) and an action (e.g., `reduce`). Spark's Scheduler translates the workflow specified via RDD transformations and actions to an execution plan. Its distributed execution engine handles the low-level details of task execution, which is triggered by actions. Spark can read data from different sources, such as HDFS, blob storage, parallel and local filesystems. Spark offloads data to disk when there is not enough free memory on a node. Persisted RDDs remain in memory, unless specified to use the disk either complementary or as a single target. In addition, Spark writes data that are used in a shuffle to disk. As a result, Spark allows quick access to those data when transmitted to an executor.

Dask

Dask [16] provides a Python-based parallel computing library, which is designed to parallelize native Python code written for NumPy and Pandas. In contrast to Spark, Dask also provides a lower-level task API (`delayed` API) that allows users to construct arbitrary workflow graphs. Being written in Python, it does not require to translate data types from one language to another like PySpark, which moves data between Python's interpreter and Java/Scala.

In addition to the low-level task API, Dask offers three higher-level abstractions: Bags, Arrays and Dataframes. Dask Arrays are a collection of NumPy arrays organized as a grid. Dask Bags are similar to Spark RDDs and are used to analyze semi-structured data, like JSON files. Dask Dataframes are distributed collections of Pandas dataframes that can be analyzed in parallel.

Dask also offers three schedulers: multithreading, multiprocessing and distributed. The multithreaded and multiprocessing schedulers support only single node execution and the parallel execution is done via threads and processes respectively. The distributed scheduler creates a cluster with a scheduling process and multiple worker processes. A client process creates and communicates a workflow as a direct acyclic graph to the scheduler. Finally, the scheduler assigns tasks to workers.

	RADICAL-Pilot	**Spark**	**Dask**
Languages	Python	Java, Scala, Python, R	Python
Task Abstraction	Task	Map-Task	Delayed
Functional Abstraction	-	RDD API	Bag
Higher-Level Abstractions	Pipelines [8], Replicas [2]	Dataframe, ML Pipeline, MLlib [62]	Dataframe, Arrays for block computations
Resource Management	Pilot-Job	Spark Execution Engines	Dask Distributed Scheduler
Scheduler	Individual Tasks	Stage-oriented DAG	DAG
Shuffle	-	hash/sort-based shuffle	hash/sort-based shuffle
Limitations	no shuffle, filesystem-based communication	high overheads for Python tasks (serialization)	Dask Array can not deal with dynamic output shapes

Table 3.1: Task-parallel Frameworks Comparison.

3.3.2 Comparison

Table 3.1 summarizes the properties of Spark, Dask and RP with respect to the abstractions and runtime systems provided to create and execute parallel data applications.

API and Abstractions

RP provides a low-level API for executing tasks onto resources. While this API can be used to implement high-level capabilities, e.g., MapReduce [63], they are not provided out-of-the box. Both Spark and Dask provide such capabilities. Dask's API is generally lower level than Spark's, i.e., it allows specifying arbitrary task graphs. Although Spark decides the partition size automatically, in many cases it is necessary to tune parallelism by specifying the number of partitions.

Another important aspect is the availability of high-level abstractions. High-level abstractions for RP, such as Pipelines [8] and Replicas [2], support compute-oriented tasks and workflows. Contrary, Spark and Dask already offer a set of high-level data-oriented abstractions, such as Dataframes.

Scheduling

Both Spark and Dask create a Direct Acyclic Graph (DAG) based on operations over data, which their respective execution engine executes. Spark jobs are separated into stages. When a stage finishes, the scheduler executes the next stage.

Dask's DAGs have a tree representation where each node is a task. Leaf tasks do not depend on other task for execution. Dask tasks execute when their dependencies are satisfied, starting from leaf tasks. When the scheduler reaches a task with unsatisfied dependencies, the scheduler executes the dependent task first. Dask's scheduler does not rely on synchronization points that Spark's stage-oriented scheduler introduces. RP does not provide a DAG-based API and requires the user to manage the execution order and synchronization among tasks at application level instead of runtime level (i.e., RP).

3.3.3 Frameworks Evaluation

As data-parallelism often involves a large number of short-running tasks, task throughput is a critical metric to assess the three frameworks we consider. To evaluate the throughput of those frameworks, we use zero workload tasks (`/bin/hostname`). We submit an increasing number of such tasks to RP, Spark and Dask and measure the execution time on a single node.

For RP, all tasks were submitted simultaneously. RP's backend database was running on the same node to avoid communication latencies. For Spark, we created an RDD with as many partitions as the number of tasks, as each task maps to a partition by Spark. For Dask, we created tasks using `delayed` functions that were executed by the Distributed scheduler. We used TACC Wrangler and SDSC Comet for this experiment. SDSC Comet is a 2.7 petaFLOSP cluster with 24 Haswell cores/node and 128 GB memory/node (6,400 nodes). TACC Wrangler has 24 Haswell hyper-threading enabled cores/node and 128 GB memory/node (120 nodes).

Figure 3.2 shows the total time to completion and task throughput. Dask needed the least time to schedule and execute the assigned tasks, followed by Spark and RP. Dask and Spark quickly reach their maximum throughput, which is sustained as the number of tasks increased. RP showed the worst throughput and scalability, mainly due to some architectural limitations. Specifically, RP relies on a MongoDB database to communicate between Client and Agent, as well as several components that allow RP to move data and introduce delays in the execution of the tasks. As a result, we were not able to scale RP to 32k or more tasks.

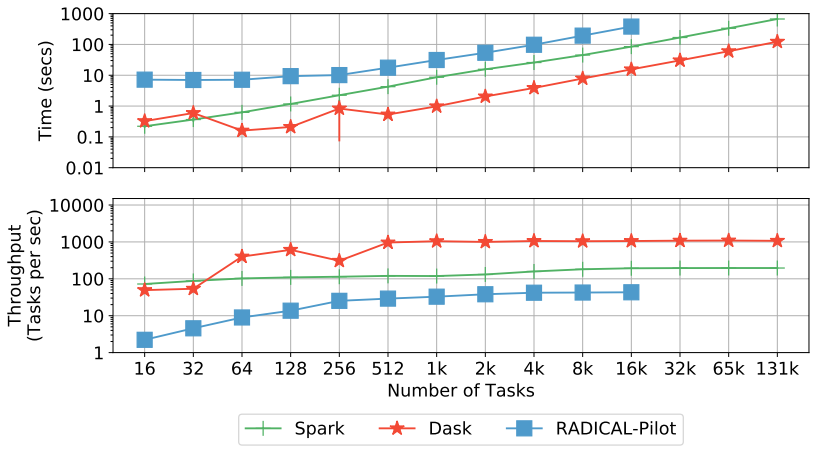


Figure 3.2: Total time to completion and task throughput by framework on a single node for an increasing number of tasks on Wrangler.

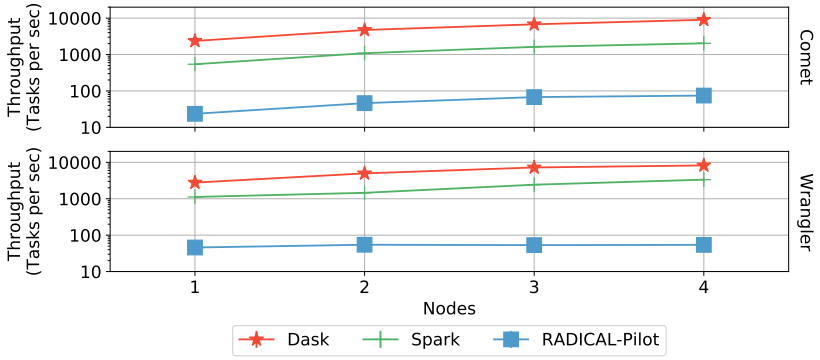


Figure 3.3: Task throughput by framewrork for $100k$ tasks on different number of nodes.

Figure 3.3 illustrates task throughput when scaling to multiple nodes, measured by submitting $100k$ tasks. Dask's throughput on both resources increases almost linearly to the number of nodes. Spark's throughput is an order of magnitude lower than Dask's. RP's throughput plateaus below $100 task/sec$. Wrangler and Comet show a comparable performance, with Comet slightly outperforming Wrangler.

3.4 Task-Parallel MD Trajectory Data Analysis: Implementation & Characterization

In this section, we characterize and evaluate the performance of two algorithms from MDAnalysis—PSA and LF—using different real-world datasets, when implemented with RP, Spark and Dask. We compare the performance of these three task-parallel

frameworks with an equivalent implementations of the algorithms using MPI4py. We investigate: 1) which capabilities and abstractions of the frameworks are needed to efficiently express these algorithms; 2) what architectural approaches can be used to implement these algorithms with these frameworks; and 3) the performance trade-offs of these frameworks.

The experiments were executed on SDSC Comet and TACC Wrangler. Experiments were carried using RP and Pilot-Spark (as discussed on §2). We utilize a set of custom scripts to start the Dask cluster. We used RP 0.46.3, Spark 2.2.0, Dask 0.14.1 and Distributed 1.16.3. We employed up to 10 nodes on Comet and Wrangler.

3.4.1 Path Similarity Analysis (PSA): Hausdorff Distance

The PSA algorithm is embarrassingly parallel and can be implemented using simple task-level parallelism or a map-only MapReduce application. We equally distribute the input data, i.e., a set of trajectory files, over the cores, generating one task per core. Each task reads its respective input files in parallel, executes and writes the result to a file.

For RP, we define a CU for each task and execute them using a Pilot-Job. For Spark, we create an RDD with one data partition per task which is a `map` function. In Dask, tasks are defined as `delayed` functions and in MPI, each task is executed by an MPI process. The dataset used for the experiments consists of three atom count trajectories: 1) small (3341 atoms/frame); 2) medium (6682 atoms/frame); and 3) large (13364 atoms/frame). We used 102 frames, and 128 and 256 trajectories of each size.

Figure 3.4 shows the runtime for 128 and 256 trajectories on Wrangler, while Figure 3.5 compares the execution times on Comet and Wrangler for 128 large trajectories. The three frameworks show similar performance in terms of runtime on both systems, as well as with MPI4py. Wrangler gives smaller speedup than Comet despite using the same number of cores. Wrangler uses hyperthreading and, as a result, multiple tasks share the same CPU, resulting in lower speedup than Comet.

MPI4py, RP, Spark and Dask have similar performance when used to execute embarrassingly parallel algorithms. All frameworks achieved similar speedups as the

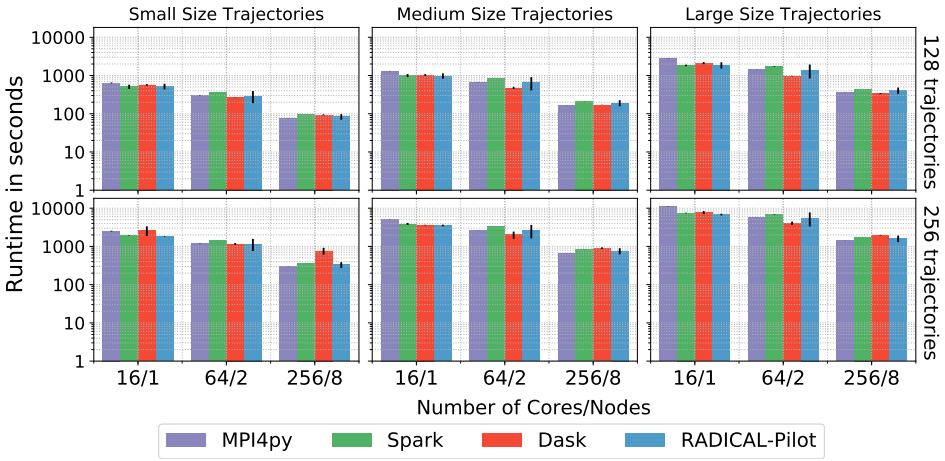


Figure 3.4: Time to completion of PSA on Wrangler using RADICAL-Pilot, Spark and Dask over different number of cores, trajectory sizes, and number of trajectories.

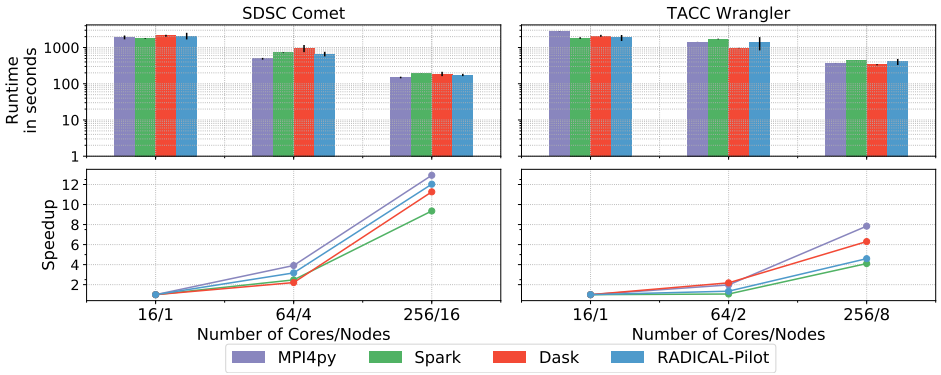


Figure 3.5: Time to completion and speedup of PSA on Comet and Wrangler for 128 large trajectories.

number of cores increased, scaling by a factor of 6 from 16 to 256 cores, which are lower than MPI4py. Although the frameworks' overheads are comparably low in relation to the overall runtime, the overheads were significant enough to impact the frameworks' speedup. Note that RP's large deviation is due to sensitivity to communication delays with the database.

In summary, for embarrassingly parallel algorithms all three frameworks provide appropriate abstractions and runtime performance compared to MPI. Beyond performance considerations, aspects such as programmability and integrate-ability are more important considerations when comparing the three frameworks. Both RP and Dask are native Python frameworks, making the integration with other Python tools easier and more efficient than with frameworks which are based on other languages like Spark.

	Broadcast and 1-D (Approach 1)	**Task API and 2-D** (Approach 2)	**Parallel Connected Components** (Approach 3)	**Tree-Search** (Approach 4)
Data Partitioning	1D	2D	2D	2D
Map	Edge Discovery via Pairwise Distance	Edge Discovery via Pairwise Distance	Edge Discovery via Pairwise Distance and Partial Connected Components	Edge Discovery via Tree-based Algorithm and Partial Connected Components
Shuffle	Edge List ($O(E)$)	Edge List ($O(E)$)	Partial Connected components ($O(n)$)	Partial Connected components ($O(n)$)
Reduce	Connect Components	Connected Components	Joined Connected Components	Joined Connected Components

Table 3.2: MapReduce Operations used by Leaflet Finder

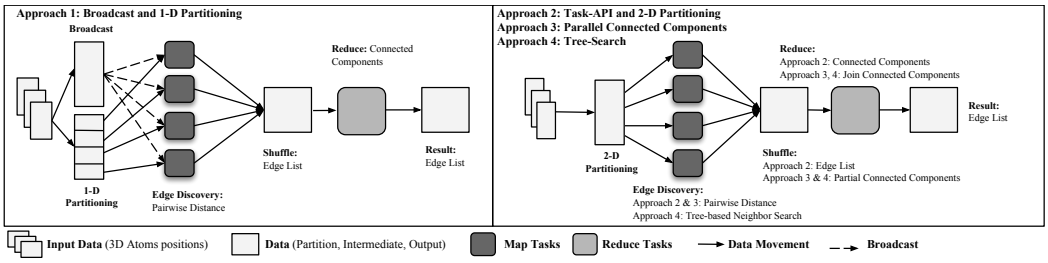


Figure 3.6: Architectural approaches for implementing the Leaflet finder algorithm

3.4.2 Leaflet Finder (LF)

We developed five approaches to implement the LF algorithm using RP, Spark, Dask, and MPI4py (see Fig 3.6 and Table 3.2):

1) **Broadcast and 1-D Partitioning:** Broadcast and partition through a data abstraction the physical system. Use of RDD API (broadcast), Dask Bag API (scatter), and MPI Bcast to distribute data to all nodes. A `map` function calculates the edge list using `cdist` from SciPy [57], realized as a loop for MPI. The master process (rank 0) collects (gathers) the list and calculates the connected components.

2) **Task API and 2-D Partitioning:** Data management is done without using the data-parallel API. The framework is used for task scheduling. Data are pre-partitioned in 2-D partitions and passed to a `map` function that calculates the edge list using `cdist`, realized as a loop for MPI. The master process (rank 0) collects (gathers) the list and calculates the connected components.

3) **Parallel Connected Components:** Data are managed as in approach 2. Each `map` task performs edge list and connected components computations. The reduce phase joins the calculated components into one, when there is at least one common node.

4) **Tree-based Nearest Neighbor and Parallel-Connected Components (Tree-Search):** This approach is different to approach 3 only for the way in which edge discovery is implemented in the `map` phase. A tree containing all atoms is created which is then used to query for adjacent atoms.

We use four physical systems with 131k, 262k, 524k, and 4M atoms with 896k, 1.75M, 3.52M, and 44.6M edges in their graphs. We utilized up to 256 cores on Wrangler for our experiments. Data partitioning results into 1024 partitions for each approach, thus 1024 `map` tasks. Due to memory limitations from using `cdist` – uses double precision floating point – Approach 3 data partitioning of the 4M atom dataset resulted to 42k tasks for both Spark and MPI4py.

Figure 3.7 shows the runtimes for all datasets for Spark, Dask and MPI4py. Figure 3.9 shows RP's performance. We continue by analyzing the performance of each architectural approach and used framework in detail.

Broadcast and 1-D Partitioning

Approach 1 broadcasts data to nodes via the capabilities supported by Spark, Dask and MPI. All nodes maintain a complete copy of the dataset and each `map` task computes the pairwise distance on a partition of the dataset. Figure 3.8 shows the detailed results.

The usage of broadcast capabilities has severe limitations for Spark and Dask. MPI broadcast is a fraction of the overall execution time and significantly smaller than Spark and Dask. MPI's broadcast times increase linearly as the number of processes increases, while Spark's and Dask's remain relatively constant for each dataset, due to more elaborate broadcast algorithms compared to MPI. Broadcast times are between 3% and 15% of the edge discovery time for Spark, from 40% to 65% for Dask, and from ¡ 1% to 10% for MPI4py. Thus, Spark offers a more efficient communication subsystem

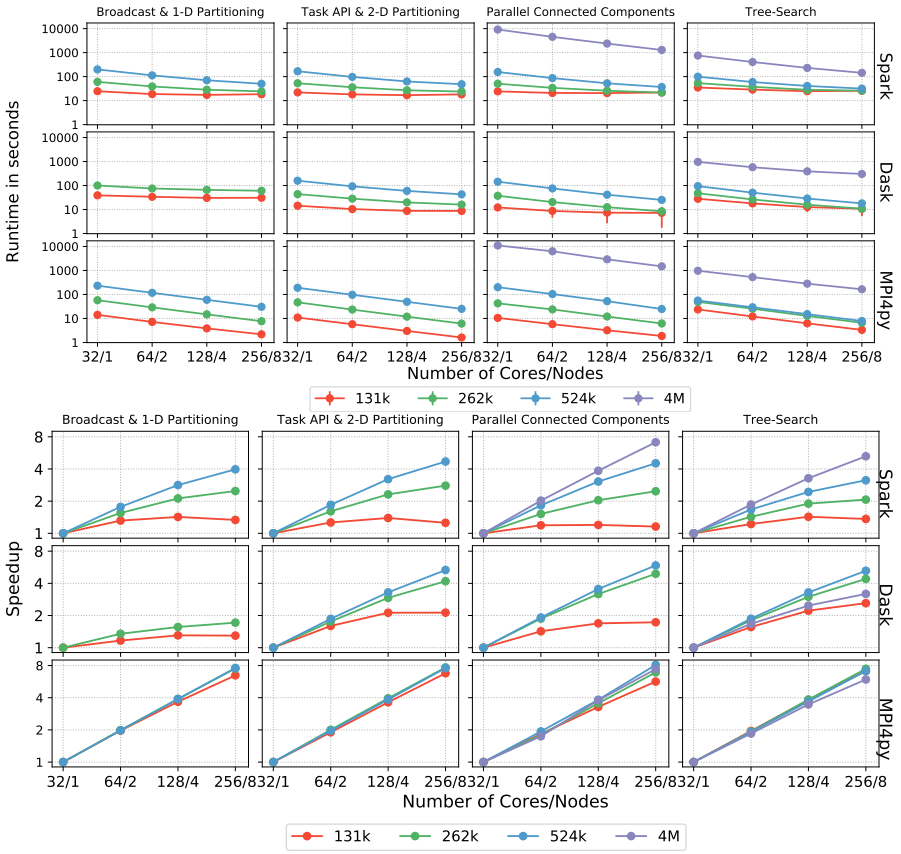


Figure 3.7: Leaflet Finder Performance of Different Architectural Approaches for Spark & Dask. Runtimes and Speedups for different system sizes over different number of cores for all approaches and frameworks.

compared to Dask. In addition, Dask broadcast partitions the dataset to a list where each element represents a value from the initial dataset. This did not allow broadcasting the 524k atom dataset. Nevertheless, the limited scalability of this approach due to transmitting the entire dataset makes it usable only for small datasets.

Using broadcast shows the worst performance and scaling of all approaches for Spark, Dask and MPI4py. This approach scales only up to 262k atoms for Dask, and 524k atoms for Spark and MPI4py on Wrangler. Spark's performance is comparable to MPI4py for the 262k, and 524k datasets. Spark also shows better performance for the smallest core count in the 524k case. Dask is at least two times slower than our MPI implementation.

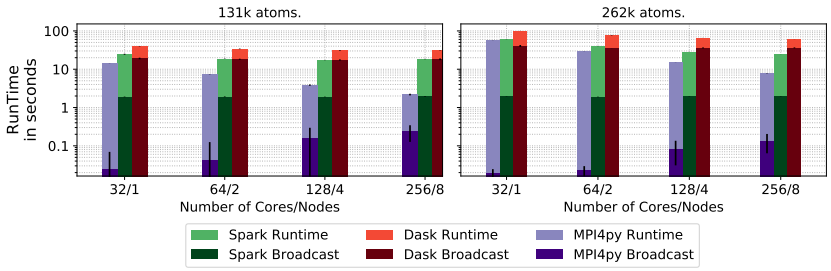


Figure 3.8: Broadcast and 1-D Partitioned Leaflet Finder (Approach 1). Runtime for multiple system sizes on different number of cores for Spark, Dask and MPI4py.

Task-API and 2-D Partitioning

Approach 2 tries to overcome the limitations of approach 1, especially broadcasting and 1-D partitioning. A 2-D block partitioning evenly distributes the compute and utilizes more efficiently the available memory. Spark and Dask do not support well 2-D partitioning. Spark's RDDs are optimized for data-parallel applications with 1-D partitioning. While Dask's array supports 2-D block partitioning, it was not used for this implementation. We return the adjacency list of the graph instead of an array to fully use the capabilities of the abstraction. Thus, each task works on a 2-D pre-partitioned part of the input data.

Figure 3.7 shows the runtimes of approach 2 for Spark, Dask, and MPI4py, and Figure 3.9 shows the runtime of approach 2 for RP. As expected, this approach overcomes the limitations of approach 1 and can easily scale to larger datasets (e.g., 524k atoms) while improving the overall runtime. Dask's execution time was smaller by at least a factor of two compared to approach 1. However, we were not able to scale this implementation to the 4M dataset due to the memory requirements of `cdist`. For RP, we observed significant task management overheads (see also section 3.3.3). This is a limitation of RP with respect to managing large numbers of tasks, particularly visible when running on a single node with 32 cores. As more than 64 cores become available, RP's performance improves dramatically.

Spark and Dask did not scale as well as MPI, which achieved linear speedups of ~ 8 when using 256 cores. Spark and Dask achieved maximum speedups of 4.5 and ~ 5 respectively. Despite this fact, both frameworks had similar performance on 32 cores for

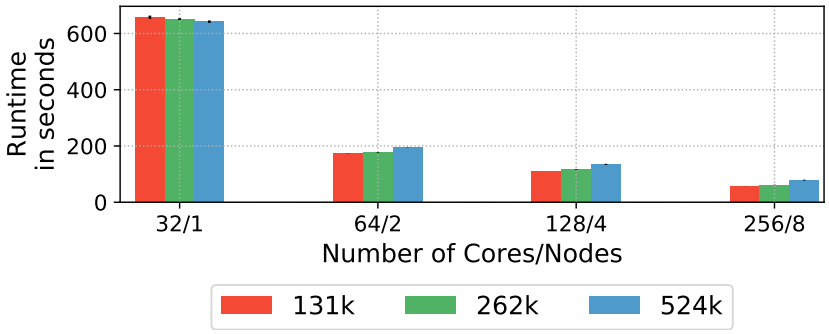


Figure 3.9: RADICAL-Pilot Task API and 2-D Partitioned Leaflet Finder (Approach 2). Runtime for multiple system sizes over different number of cores.

the 262k and 524k datasets.

Parallel Connected Components

Communication between the edge discovery and connected components phases is another important aspect. For the 524k atoms dataset, the output of the edge discovery phase is $\approx 100\,\text{MB}$. To reduce the amount of data that need to be shuffled, we refined the algorithm to compute the graph components on the partial dataset in the `map` phase. The `reduce` phase then merges the components. This reduces the amount of shuffle data by more than 50% (e.g., to 12MB for Spark and 48MB for Dask). Figure 3.7 shows the improvements in runtime, by $\sim 20\%$ for Spark and Dask, but not MPI4py. Further, we were able to run very large datasets, such as the 4M dataset, using this architectural approach only with Spark and MPI4py. Dask was restarting its worker processes because their memory utilization was reaching 95%.

Spark and Dask have comparable performance with MPI on 32 cores, which utilizes a single node on Wrangler. However, while the MPI4py implementation scales almost linearly for all datasets, Spark and Dask cannot, reaching a maximum of ~ 5 speedup for the three smaller datasets. In addition, Spark is able to scale almost linearly for the 4M atoms dataset, providing comparable performance to MPI4py.

Tree-Search

A bottleneck of approaches 1, 2 and 3 is the edge discovery via the naive calculation of the distances between all pairs of atoms. In approach 4, we replace the pairwise distance function with a tree-based, nearest neighbor search algorithm, in particular BallTree [64]. The algorithm: 1. constructs of a tree; and 2. queries for neighboring atoms. Using tree-search, the computational complexity can be reduced from n^2 to log. We use a BallTree as offered by Scikit-Learn [65] for our implementation.

Figure 3.7 illustrates the performance of the implementation. For small datasets, i.e., 131k and 262k atoms, approach 3 is faster than the tree-based approach, since the number of points is too small. For the large datasets, the tree approach is faster. In addition, the tree has a smaller memory footprint than `cdist`. As a result, the tree approach scaled to larger problems, e.g., a 4M atoms and 44.6M edges dataset, without changing the total number of tasks.

Dask shows better scaling performance than Spark for 131k, 262k and 524k atoms. This is not true for 4M atoms, indicating that Dask's communication layer is not able to scale as well as Spark's one. Spark shows similar performance to MPI4py for the largest dataset due to minimal shuffle traffic. Thus, MPI's efficient communication does not become relevant.

3.5 Conceptual Framework for Selecting Task-Parallel Frameworks

In this section, we provide a conceptual framework that allows application developers to select a framework according to their compute and I/O requirements. It is important to understand the properties of both the application and task-parallel frameworks. Table 3.3 illustrates the criteria of this conceptual model and ranks the three frameworks.

Application Perspective

We showed that we can implement applications for MD trajectory data analysis using Spark, Dask and RP, as well as MPI4py. Implementation aspects, such as computational complexity, and shuffled data size greatly influence performance. For embarrassingly

	RADICAL-Pilot	Spark	Dask
Task Management			
Low Latency	-	o	+
Throughput	-	+	++
MPI/HPC Tasks	+	o	o
Task API	+	o	++
Large Number of Tasks	–	++	++
Application Characteristics			
Python/native Code	++	o	+
Java	o	++	o
Higher-Level Abstraction	-	++	+
Shuffle	-	++	+
Broadcast	-	++	+
Caching	-	++	o

Table 3.3: Task-parallel framework selection decision methodology: Criteria and Ranking for Framework Selection. - : Unsupported or low performance + : Supported, ++ : Major Support, and o :Minor support.

parallel applications with coarse grained tasks, such as PSA, the choice of the framework does not significantly influence performance. In addition, the performance difference against MPI4py was not significant (Figures 3.4, 3.5). Thus, aspects such as programmability and integrate-ability, become more important than performance alone.

For fine-grained data parallelism, a data-parallel framework, such as Spark and Dask, clearly outperforms RP (Figures 3.7, 3.9). If there is coupling between tasks, i.e., task communication is required, using Spark becomes advantageous (Approaches 3 & 4). MPI4py outperformed Dask and Spark, despite both frameworks scaling for the larger datasets. Especially Spark was able to provide linear speedup for approach 3 of Leaflet Finder (Figure 3.7).

Integrating with frameworks that provide higher level abstractions provides scalable solutions for more complex algorithms. However, integrating Spark with other tools needs to be carefully considered. The integration of Python tools, e.g., MDAnalysis, often causes overheads due to the frequent need for serialization and copying data between the Python and Java space.

Dask has the smallest learning curve of all three frameworks. As a result, it allows for faster prototyping compared to RP and Spark. RP's learning curve is more steep, but is more versatile than Dask and Spark, by offering the lowest level abstraction. Spark had the slowest learning curve but it required tuning to get the number of tasks correctly, as well as argument passing to map and reduce functions.

Framework Perspective

RP is well suited for HPC applications, e.g., ensembles (up to $50k$ tasks) of parallel MPI applications, as shown in Ref. [31, 17]. It has limited scalability when supporting large numbers of short-running tasks, as often found in data-intensive workloads. The file staging implementation of RP is not suitable for supporting the data exchange patterns, i.e., shuffling, required for these applications. However, concurrently executing MPI and Spark applications on the same resource makes RP particularly suitable when different programming models need to be combined.

Dask provides a highly flexible, low-latency task management and excellent support for parallelizing Python libraries. We established that Dask has higher throughput than the other frameworks (Figures 3.2, 3.3). However, Spark provides better speedups for the largest datasets compared to Dask (Figure 3.7). Dask's broadcast (Leaflet Finder with Approach 1) and shuffle (Leaflet Finder with Approaches 2- 4) performance is worse for larger problems compared to Spark. Thus, Dask's communication layer shows some weaknesses that are particularly visible during broadcast and shuffle. Spark needs to be taken into consideration for shuffle-intensive applications. Its in-memory caching mechanism is particularly suited for iterative algorithms that maintain a static set of data in-memory and conduct multiple passes on that set.

3.6 Conclusions

In this chapter, we investigated the use of different programming abstractions and frameworks for implementing a range of algorithms for MD trajectory analysis. We conducted an in-depth analysis of applications' characteristics and assessed the architectures of RP, Spark and Dask. We provided a conceptual framework that enables application developers to qualitatively evaluate task parallel frameworks with respect to application requirements. Our benchmarks enable developers to quantitatively assess framework performance as well as the performance of different implementations. Our method can be used for any application in which data are represented as time series of simulated systems, e.g., weather forecast and earthquake simulation.

While the task abstractions provided by all frameworks are well-suited for implementing the considered use cases, the high-level MapReduce programming model provided by Spark and Dask has several advantages. It is easier to use and efficiently supports common data exchange patterns, e.g., shuffling between `map` and `reduce` stages. In our benchmarks, Spark outperforms Dask in communication-intensive tasks, such as broadcasts and shuffles. Dask provides more versatile low and high level APIs and integrates better with python frameworks. RP does not provide a MapReduce API, but is well suited for coarse-grained task-level parallelism [31, 17], and when HPC and analytics frameworks need to be integrated. We also identified a limitation in Dask and Spark: while both frameworks provide some support for linear algebra through a distributed array abstraction, it proved inflexible for an efficient all-pairs pattern implementation. Dask and Spark required workarounds and utilization of out-of-framework functions to read and partition data (Table 3.2). Although none of these frameworks outperformed MPI, their scaling capabilities along with their high-level APIs create a strong case on utilizing them for data analytics with HPC applications.

Our results allow users to utilize a well-suited task-parallel framework based on the characteristics of the data-intensive workflows of a computational campaign. This in turn reduces the time to define the data-intensive workflows of a computational campaign. Further, it allows the scalable execution of such workflows without significant, if any, effort from users, reducing the time to optimize a workflow. However, there is a set of data analysis workflows that do not necessarily conform with the MapReduce abstraction, such as the PSA analysis or earth science workflows that analyze imagery, as they have both data and compute intensive characteristics. As a result, the selected framework should efficiently and effectively perform data transfers and execution while offering the necessary abstractions. In the next chapter we discuss architectural equivalent designs of task-parallel frameworks to execute such data- and compute-intensive workflows.

Chapter 4

Evaluating Middleware for Task-Parallel Data Analytics on HPC Platforms

This far, we have discussed how to efficiently and scalably support the execution of MapReduce workflows, as part of a computational campaign, on High Performance Computing (HPC) resources. There are, though, computational campaigns whose workflows do not necessarily conform to the MapReduce abstraction. The tasks of such workflows can be heterogeneous, implementing a diverse set of functionalities that are compute-intensive. As a consequence, data-parallel frameworks, like Spark and Dask may not be the most suitable choice for their execution. From a middleware perspective, the design and architectural space is large and the lack of performance analysis makes it difficult to select among equivalent implementations.

From a design perspective, a promising approach is isolating tasks from execution management. Tasks are assumed to be self-contained programs which are executed in the operating system (OS) environment of HPC compute nodes. Compared to approaches in which tasks are functions or methods, a program-based approach offers several benefits as, for example, simplified implementation of execution management, support of general purpose programming models, and separate programming of management and domain-specific functionalities. Nonetheless, program-based designs also impose performance limitations, including OS-mediated intertask communication and task spawning overheads, as programs execute as OS processes and do not share a memory space.

Due to their performance limitations, program-based designs of computing frameworks are best suited to execute compute-intense workflows in which each task requires

a certain amount of parallelism and runs from several minutes to hours. The emergence of workflows that require heterogeneous, compute-intense tasks to process large amount of data is pushing the boundaries of program-based designs, especially when scale requirements suggest the use of modern HPC infrastructures with large number of CPUs/GPUs and dedicated data facilities.

We use a paradigmatic use case workflow from the polar science domain to evaluate three alternative task-based designs, and experimentally characterize and compare their performance. Our use case requires us to analyze satellite images of Antarctica to detect pack-ice seals taken across a whole calendar year. The resulting dataset consists of $3,097$ images for a total of ≈ 4 TB. The use case requires us to repeatedly process these images, running both CPUs and GPUs code that exchange several GB of data. The first design uses a pipeline to independently process each image, while the second and third designs use the same pipeline to process a series of images with differences in how images are bound to available compute nodes.

The chapter is organized as follows. § 4.1 discusses the current state-of-the-art in parallel image analysis. § 4.2 presents the use case and discusses its computational requirements as well as its task-based workflow. In §4.3, we discuss the three architecturally equivalent design to support task-based workflows. §4.4 presents a performance evaluation of the designs and discusses the lessons learned.

4.1 Related Work

Several tools and frameworks are available for image analysis based on diverse designs and programming paradigms, and implemented for specific resources. Numerous image analytics frameworks for medical, astronomical, and other domain specific imagery provide MapReduce implementations. MaReIA [66], built for medical image analysis, is based on Hadoop and Spark [10]. Kira [67], built for astronomical image analysis, is also built on top of Spark and pySpark, allowing users to define custom analysis applications. Further, Ref. [68] proposes a Hadoop-based cloud Platform as a Service, utilizing Hadoop's streaming capabilities to reduce filesystem reads and writes. These

frameworks support clouds and/or commodity clusters for execution.

BIGS [69] is a framework for image processing and analysis. BIGS is based on the master-worker model and supports heterogeneous resources, such as clouds, grids and clusters. The user is responsible to define the input, processing pipeline, and launch BIGS workers. BIGS utilizes a database to perform data transfers between workers. In addition, BIGS offers a diverse set of APIs for developers. BIGS approach is very close to Design 1 we described in §4.3.1.

Image analysis libraries, frameworks and applications have been proposed for HPC resources. PIMA(GE)^2 Library [70] provides a low-level API for parallel image processing using MPI and CUDA. SIBIA [71] is a framework for coupling biophysical models with medical image analysis, and provides parallel computational kernels through MPI and vectorization. Tomosaic [72] is a Python framework, used for medical imaging, employing MPI4py to parallelize different parts of the workflow.

Petruzza et al. [73] describe a scalable image analysis library. Their approach defines pipelines as data-flow graphs, with user defined functions as tasks. Charm++ is used as the workflow management layer, by abstracting the execution level details, allowing execution on local workstations and HPC resources. Teodoro et al. [74] define a master-worker framework supporting image analysis pipelines on heterogeneous resources. The user defines an abstract dataflow and the framework is responsible to schedule tasks on CPU or GPUs.Data communication and coordination is done via MPI.

Our approach proposes designs for image analysis pipelines that are domain independent. In addition, the workflow and runtime systems we use allow execution on multiple HPC resources with no change in our approach. Furthermore, parallelization is inferred in one of the proposed designs, allowing correct execution regardless of the multi-core or multi-GPU capabilities of the used resource.

All the above, except Ref. [67], focus on characterizing the performance of the proposed solution. Ref. [67] compares different implementations, one with Spark, one with pySpark, and an MPI C-based implementation. This comparison is based on the weak and strong scaling properties of the approaches. Our approach offers a well-defined methodology to compare different designs for task-based and data-driven pipelines with

heterogeneous tasks.

4.2 Satellite Imagery Analysis Application

Imagery employed by ecologists as a tool to survey populations and ecosystems come from a wide range of sensors, e.g., camera-trap surveys [75] and aerial imagery transects [76]. Very High Resolution (VHR) satellite imagery provides an effective alternative to perform large scale surveys at locations with poor accessibility such as surveying Antarctic fauna [77]. To take full advantage from increasingly large VHR imagery, and reach the spatial and temporal breadths required to answer ecological questions, it is paramount to automate image processing and labeling.

Convolutional Neural Networks (CNN) represent the state-of-the-art for nearly every computer vision routine. For instance, ecologists have successfully employed CNNs to detect large mammals in airborne imagery [78, 79], and camera-trap survey imagery [80]. We use a Convolutional Neural Network (CNN) to survey Antarctic pack-ice seals in VHR imagery. Pack-ice seals are a main component of the Antarctic food web [81]: estimating the size and trends of their populations is key to understanding how the Southern Ocean ecosystem copes with climate change [82] and fisheries [83].

This use case workflow processes WorldView 3 (WV03) panchromatic imagery, which has the highest available resolution for commercial satellite imagery. Due to limitations on GPU memory, it is necessary to tile WV03 images into smaller patches before sending input imagery through the seal detection CNN. Taking tiled imagery as input, the CNN outputs the latitude and longitude of each detected seal. We order the tiling and seal detection stages into a pipeline that can be run with different imagery datasets. This allows domain scientists to create seal abundance time series that can aid in Southern Ocean monitoring. To create this time series, domain scientists define a computational campaign where each workflow analyzes imagery from different calendar years or seasons.

4.3 Workflow Design and Implementation

Computationally, the use case described in §4.2 presents three main challenges: heterogeneity, scale and repeatability. The images of the use case dataset vary in size with a wide distribution. Each image requires tiling, per tile counting of seal populations and result aggregation across tiles.Tiling is memory intensive while counting is computationally intensive, requiring CPU and GPU implementations, respectively.

We address these challenges by codifying image analysis into a workflow. We then execute this workflow on HPC resources, leveraging the concurrency, storage systems and compute speed they offer to reduce time to completion. Typically, this type of workflow consists of a sequence (i.e., pipeline) of tasks, each performing part of the end-to-end analysis on one or more images. The implementation of this workflow varies, depending on the API of the chosen workflow system and its supported runtime system. Here, we compare two common designs: one in which each image is processed independently by a dedicated pipeline; the other in which a pipeline processes multiple images.

Note that both designs separate the functionalities required to process each image from the functionalities used to coordinate the processing of multiple images. This is consistent with moving away from vertical, end-to-end single-point solutions, favoring designs and implementations that satisfy multiple use cases, possibly across diverse domains. Accordingly, the two designs we consider, implement, and characterize use two tasks (i.e., programs) to implement the tiling and counting functionalities required by the use case.

Both designs are functionally equivalent, in that they both enable the analysis of the given dataset. Nonetheless, each design leads to different amounts of concurrency, resource utilization and overheads, depending on data/ compute affinity, scheduling algorithm, and coordination between CPU and GPU computations. Based on a performance analysis, it will be possible to know which design entails the best tradeoffs for common metrics as total execution time or resource utilization.

Consistent with HPC resources currently available for research and our use case, we make three architectural assumptions: (1) each compute node has c CPUs; (2) each

compute node has g GPUs where $g \leq c$; and (3) each compute node has enough memory to enable concurrent execute of a certain number of tasks. As a result, at any given point in time there are $C = n \times c$ CPUs and $G = n \times g$ GPUs available, where n is the number of compute nodes.

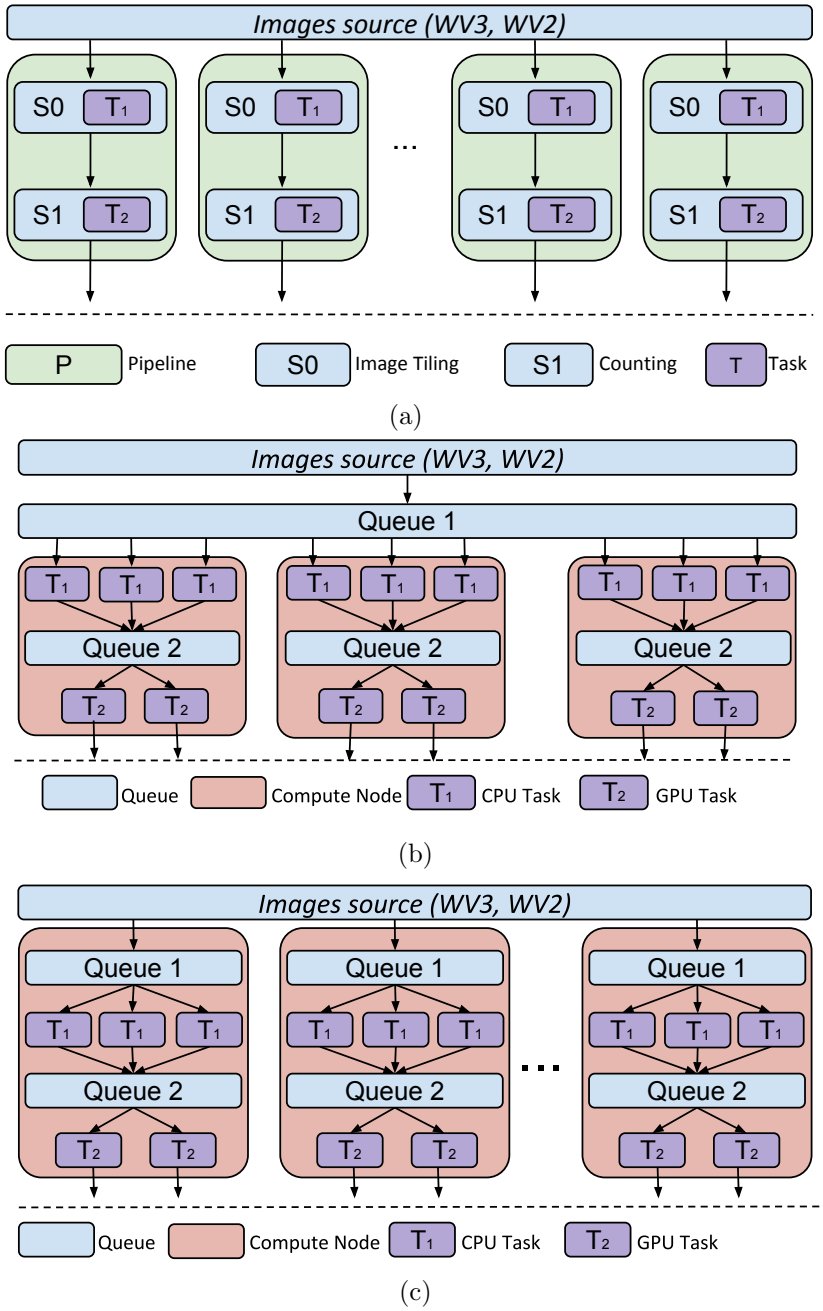


Figure 4.1: Design approaches to implement the workflow required for the Seals use case. 4.1a –**Design 1**: Pipeline, stage and task based design. 4.1b –**Design 2**: Queue based design with a single queue for all the tiling tasks. 4.1c –**Design 2.A**: Queue based design with multiple queues for the tiling tasks.

4.3.1 Design 1: One Image per Pipeline

We specify the workflow for counting the number of seals in a set of images as a set of pipelines. Each pipeline is composed of two stages, each with one type of task. The task of the first stage gets an image as input and generates tiles of that image as output. The task of the second stage gets the generated tiles as input, counts the number of seals found in each tile and outputs the aggregated result for the whole image.

Formally, we define two types of tasks:

- $T_1 =< I, f_I, t >$, where I is an image, f_I is a tiling function and t is a set of tiles that correspond to I.
- $T_2 =< t, f_A, S >$, where f_A is a function that counts seals from a set of tiles and S is the number of seals.

Tiling in T_1 is implemented with OpenCV [84] and Rasterio [85] in Python. Rasterio allows us to open and convert a GeoTIFF WV3 image to an array. The array is then partitioned to subarrays based on a user-specified scaling factor. Each subarray is converted to an compressed image via OpenCV routines and saved to the filesystem.

Seal counting in T_2 is performed via a Convolutional Neural Network (CNN) implemented with PyTorch [86]. The CNN counts the number of seals for each tile of an input image. When all tiles are processed, the coordinates of the tiles are converted to the geographical coordinates of the image and saved in a file, along with the number of counted seals. Note that the number of seals in an tile does not affect the execution of the network, i.e., the same number of operations will be executed.

Both tiling and seal counting implementations are invariant between the designs we consider. This is consistent with the designs being task-based, i.e., each task exclusively encapsulates the capabilities required to perform a specific operation over an image or tile. Thus, tasks are independent from the capabilities required to coordinate their execution, whether each task processes a single image or sequence of images.

We implemented Design 1 via EnTK, a workflow engine which exposes an API based on pipelines, stages, and tasks [8]. The user can define a set of pipelines, where

each pipeline has a sequence of stages, and each stage has a set of tasks. Stages are executed respecting their order in the pipeline while the tasks in each stage can execute concurrently, depending on resource availability.

For our use case, EnTK has three main advantages compared to other workflow engines: (1) it exposes pipelines and tasks as first-order abstractions implemented in Python; (2) it is specifically designed for concurrent management of up to 10^5 pipelines; and (3) it supports RADICAL-Pilot. Together, these features address the challenges of heterogeneity, scale and repeatability: users can encode multiple pipelines, each with different types of tasks, executing them at scale on HPC machines without explicitly coding parallelism and resource management.

When implemented in EnTK, the use case workflow maps to a set of pipelines, each with two stages St_1, St_2. Each stage has one task T_1 and T_2 respectively. The pipeline is defined as $P = (St_1, St_2)$. For our use case the workflow consists of N pipelines, where N is the number of images.

Figure 4.1a shows the workflow. For each pipeline, EnTK submits the task of stage St_1 to the runtime system (RTS). As soon as the task finishes, the task of stage St_2 is submitted for execution. This design allows concurrent execution of pipelines and, as a result, concurrent analysis of images, one by each pipeline. Since pipelines execute independently and concurrently, there are instances where St_1 of a pipeline executes at the same time as St_2 of another pipeline.

Design 1 has the potential to increase utilization of available resources as each compute node of the target HPC machine has multiple CPUs and GPUs. Importantly, computing concurrency comes with the price of multiple reading and writing to the filesystem on which the dataset is stored. This can cause I/O bottlenecks, especially if each task of each pipeline reads from and writes to the same filesystem, possibly over a network connection.

We used a tagged scheduler for EnTK's RTS to avoid I /O bottlenecks. This scheduler schedules T_1 of each pipeline on the first available compute node, and guarantees that the respective T_2 is scheduled on the same compute node. As a result, compute/data affinity is guaranteed among co-located T_1 and T_2. While this design avoids I /O bottlenecks, it

may reduce concurrency when the performance of the compute nodes and/or the tasks is heterogeneous: T_2 may have to wait to execute on a specific compute node while another node is free.

4.3.2 Design 2: Multiple images per pipeline

Design 2 implements a queue-based design. We introduce two tasks T_1 and T_2 as defined in §4.3.1. Contrary to Design 1, these tasks are started and then executed for as long as data and resources are available, processing input images at the rate taken to process each image. The number of concurrent T_1 and T_2 depends on available resources, including CPUs, GPUs, and RAM.

For implementing Design 2, we do not need EnTK, as we submit a bag of T_1 and T_2 tasks via the RADICAL -Pilot RTS, and manage the data movement between tasks via queues. As shown in Fig. 4.1b, Design 2 uses one queue (Queue 1) for the dataset, and another queue (Queue 2) for each compute node. For each compute node, each T_1 pulls an image from Queue 1, tiles that image and then queues the resulting tiles to Queue 2. The first available T_2 on that compute node, pulls those tiles from Queue 2, and counts the seals.

Note that Design 2 load balances T_1 tasks across compute nodes but balances T_2 tasks only within each node. For example, suppose that T_1 on compute node A runs two times faster than T_1 on compute node B. Since both tasks are pulling images from the same queue, T_1 of A will process twice as many images as T_1 of B. Both T_1 of A and B will execute for around the same amount of time until Queue 1 is empty, but Queue 2 of A will be twice as large as Queue 2 of B. T_2 tasks executing on B will process half as many images as T_2 tasks on A, possibly running for a shorter period of time, depending on the time taken to process each image.

In principle, Design 2 can be modified to load balance also across Queue 2 but in practice, as discussed in §4.3.1, this would produce I/O bottlenecks. Load balancing across T_2 tasks would require for all tiles produced by T_1 tasks to be written to and read from a shared filesystem. Keeping Queue 2 local to each compute node enables using the filesystem local to each compute node.

4.3.2.1 Design 2.A: Uniform image dataset per pipeline

The lack of load balancing of T_2 tasks in Design 2 can be mitigated by introducing a queue in each node from where T_1 tasks pull images. This allows early binding of images to compute nodes, i.e., deciding the distribution of images per node before executing T_1 and T_2. As a result, the execution can be load balanced among all available nodes, depending on the correlation between image properties and image execution time.

Figure 4.1c shows variation 2.A of Design 2. The early binding of images to compute nodes introduces an overhead compared to using late binding via a single queue as in Design 2. Nonetheless, depending on the strength of the correlation between image properties and execution time, design 2.A offers the opportunity to improve resource utilization. While in Design 2 some node may end up waiting for another node to process a much larger Queue 2, in design 2.A this is avoided by guaranteeing that each compute node has an analogous payload to process.

4.4 Experiments: Task Execution Time, Resource Utilization and Overheads

We executed three experiments using GPU compute nodes of the XSEDE Bridges supercomputer. These nodes offer 32 cores, 128 GB of RAM and two P100 Tesla GPUs. We stored the dataset of the experiments and the output files on XSEDE Pylon5 Lustre filesystem. We stored the tiles produced by the tiling tasks on the local filesystem of the compute nodes. This way, we avoided creating a performance bottleneck by performing millions of reads and writes of ≈ 700 KB on Pylon5. We submitted jobs requesting 4 compute nodes to keep the average queue time within a couple of days. Requesting more nodes produced queue times in excess of a week. In addition, we had full control over the nodes during execution and were not shared with other users.

The experiments dataset consists of $3,097$ images, ranging from 50 to $2,770$ MB for a total of ≈ 4 TB of data. The images size follows a normal distribution with a mean value of $1,304.85$ MB and standard deviation of 512.68 MB.

For Design 1, 2 and 2.A described in §4.3, Experiment 1 models the execution time

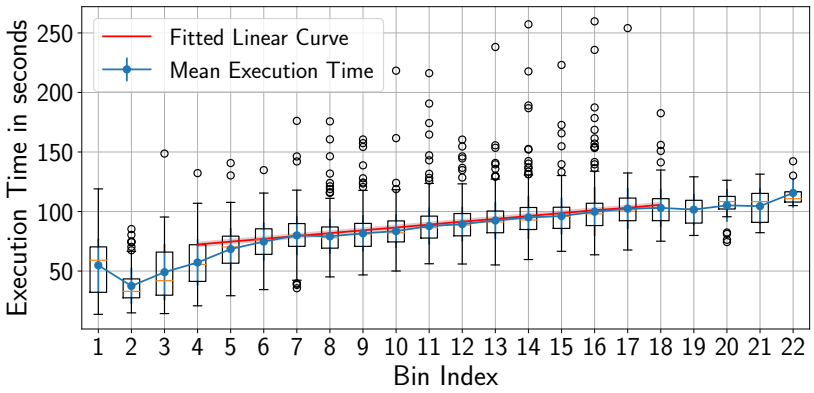


Figure 4.2: Experiment 1, Design 1: Box-plots of T_1 execution time, mean and STD for 125 MB image size bins. Red line shows fitted linear function.

of the two tasks of our use case as a function of the image size (the only property of the images for which we found a correlation with execution time); Experiment 2 measures the total resource utilization of each design; and Experiment 3 characterizes the overheads of the middleware implementing each design. Together, these experiments enable performance comparison across designs, allowing us to draw conclusions about the performance of heterogeneous task-based execution of data-driven workflows on HPC resources.

4.4.1 Experiment 1: Design 1 Tasks Execution Time

Fig. 4.2 shows the execution time of the tiling task—defined as T_1 in §4.3.1—as a function of the image size. We partition the set of images based on image size, obtaining 22 bins with a range of 125 MB each starting from 50 MB up to $2,800$ MB. The average time taken to tile an image in each bin tends to increase with the size of the image. The box-plots show some positive skew of the data with a number of data points falling outside the assumed normal distribution. There are also large standard deviations (STD, blue line) in most of the bins. Thus, there is a weak correlation between task execution time and image size with a large spread across all the image sizes.

We explored the causes of the observed STD by measuring how it varies in relation to the number of tiling tasks concurrently executing on the same node. Fig. 4.3 shows the standard deviation of each bin of Fig. 4.2, based on the amount of used task concurrency.

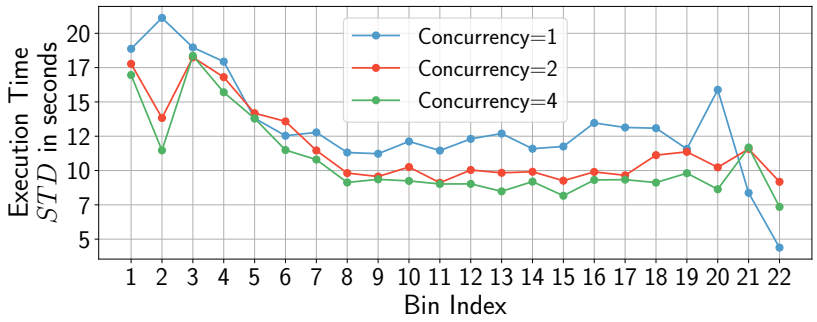


Figure 4.3: STD of T_1 execution time based on image size bin and number of concurrent tasks. Mostly dependent on compute node's performance and invariant across values of task concurrency.

Design	**Fitted Data**	**α value**	**β value**	**R^2 value**	**Figure**
1	T_1	1.92×10^{-2}	60.49	0.97	Fig. 4.2, red line
1	T_2	5.21×10^{-2}	128.53	0.96	Fig. 4.4, green line
2	T_1	3.17×10^{-2}	64.81	0.92	Fig. 4.5, red line
2	T_2	4.71×10^{-2}	95.83	0.95	Fig. 4.6, green line
2.A	T_1	2.74×10^{-2}	49.03	0.94	N/A
2.A	T_2.	4.80×10^{-2}	87.60	0.95	N/A

Table 4.1: Fitted parameter values of Eq. 4.1 using a non-linear least squares algorithm to fit our experimental data.

We observe that STD drops with increased concurrency but remains relatively stable between bins #6 and #20. We attribute the initial dropping to how Lustre's caching improves the performance of an increasing number of concurrent requests. Further, we observe that as the type of task and the compute node are the same across all our measures, the relatively stable and consistent STD observed across degrees of task concurrency depends on fluctuations in the node performance.

Fig. 4.2 indicates that the execution time is a linear function of the image size between bin #4 and bin #18. Bins $1-3$ and $19-23$ are not representative as the head and tail of the image sizes distribution contain less than 5% of the image dataset. We model the execution time as:

$$T(x) = \alpha \times x + \beta \tag{4.1}$$

where x is the image size.

We found the parameter values of Eq. 4.1 by using a non-linear least squares algorithm to fit our experimental data, which are $\alpha = 1.92 \times 10^{-2}$, and $\beta = 60.49$ (see red line in

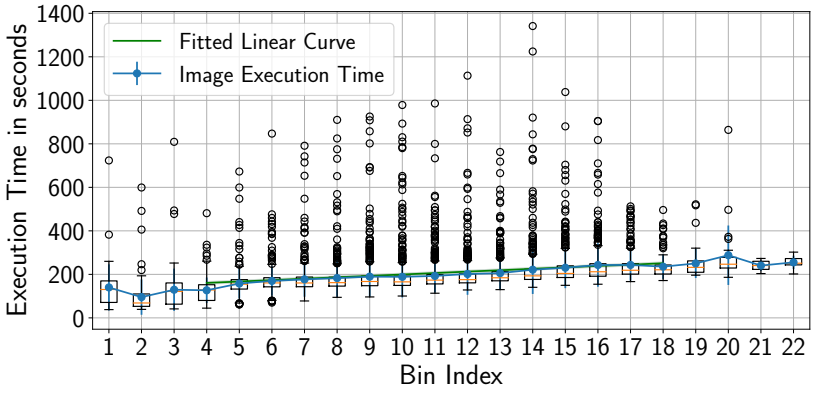


Figure 4.4: Experiment 1, Design 1: Box-plots of T_2 execution time, mean and STD for 125 MB image size bins. Green line shows fitted linear function.

Fig. 4.2). R^2 of our fitting is 0.97, showing a very good fit of the curve to the actual data.

The Standard Error of the estimation, S_{error}, reflects the precision of our regression. The S_{error} is equal to 1.93, shown as the red shadow in Fig. 4.2. From R^2 and S_{error} we conclude that our estimated function is validated and is a good fit for the execution time of T_1 for Design 1.

Fig. 4.4 shows the execution time of the seals counting task as a function of the image size. Defined as T_2 in §4.3.1, this task presents a different behavior than T_1, as the code executed is different. Note the slightly stronger positive skew of the data compared to that of Fig. 4.2 but still consistent with our conclusion that deviations are mostly due to fluctuations in the compute node performance (i.e., different code but similar fluctuations).

Similar to T_1, Fig. 4.4 shows a weak correlation between the execution time of T_2 and image size. In addition, the variance per bin is relatively similar across bins, as expected based on the analysis of T_1. The box-plot and the mean execution time indicate that a linear function is a good candidate for a model of T_2. As in Eq. 4.1, we fitted a linear function to the execution time as a function of the image size for the same bins as T_1.

Using the same method we used with T_1, we produced the green line in Fig. 4.4 with parameter values $\alpha = 5.21 \times 10^{-2}$ and $\beta = 128.53$. R^2 is 0.96, showing a good fit of the line to the actual data, while S_{error} is 5.73, slightly higher than for T_1. As a result, we

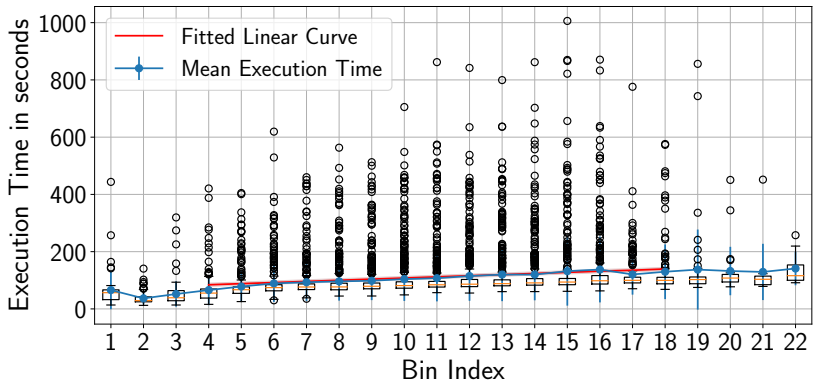


Figure 4.5: Experiment 1, Design 2: Box-plots of T_1 execution time, mean and STD for 125 MB image size bins. Red line shows fitted linear function.

conclude that our estimated function is validated and is a good fit for the execution time of T_2 for Design 1.

4.4.2 Experiment 1: Design 2 Tasks Execution Time

Fig. 4.5 shows the execution time of T_1 as a function of the image size for Design 2. In principle, design differences in middleware that execute tasks as independent programs should not directly affect task execution time. In this type of middleware, task code is independent from that of the middleware: once tasks execute, the middleware waits for each task to return. Nonetheless, in real scenarios with concurrency and heterogeneous tasks, the middleware may perform operations on multiple tasks while waiting for others to return. Accordingly, in Design 2 we observe an execution time variation comparable to that observed with Design 1 but Fig. 4.5 shows a stronger positive skew of the data in Design 2 than Fig. 4.2 in Design 1.

We investigated the positive skew of the data observed in Fig. 4.5 by comparing the system load of a compute node when executing the same number of tiling tasks in Design 1 and 2. The system load of Design 2 was higher than that of Design 1. Compute nodes have the same hardware and operating system, and run the same type and number of system programs. As we used the same type of task, image and task concurrency, we conclude that the middleware implementing Design 2 uses more compute resources than that used for Design 1. Due to concurrency, the middleware of Design 2 competes for

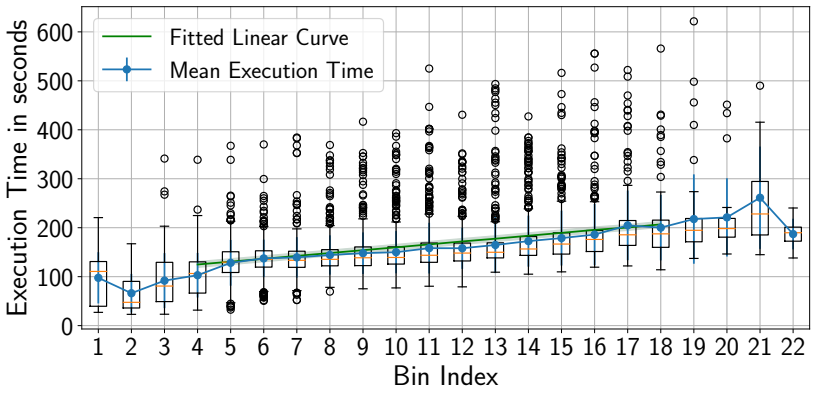


Figure 4.6: Experiment 1, Design 2: Box-plots of T_2 execution time, mean and STD for 125 MB image size bins. Green line shows fitted linear function.

resources with the tasks, momentarily slowing down their execution. This is consistent with the architectural differences across the two designs: Design 2 requires resources to manage queues and data movement while Design 1 has only to schedule and launch tasks on each node.

We fitted Eq. 4.1 to the execution time of T_1 for Design 2, obtaining $\alpha = 3.174 \times 10^{-2}$ and $\beta = 64.81$. The fitting produced the red line in Fig. 4.5. R^2 is 0.92, showing a good fit of the curve to the data and S_{error} is 5.50, validating our estimated function. R^2 and especially S_{error} are worse compared to Design 1, an expected difference based on the positive skew of the data observed in Design 2.

Fig. 4.6 shows that Design 2 also produces a much stronger positive skew of T_2 execution time compared to executing T_2 with Design 1. T_2 executes on GPU and T_1 on CPU but their execution times produce comparable skew in Design 2. This further supports our hypothesis that the long tail of the distribution of T_1 and especially T_2 execution times depends on the competition for main memory and I/O between the middleware implementing Design 2 and the executing tasks.

Fitting the model gives $\alpha = 4.71 \times 10^{-2}$ and $\beta = 95.83$.R^2 is 0.95 and S_{error} of 5.96. As a result, the model is validated and a good candidate for the experimental data. We attribute the difference between these values and those of the model of T_2 for Design 1 to the already described positive skew of execution times in Design 2.

4.4.2.1 Design 2.A

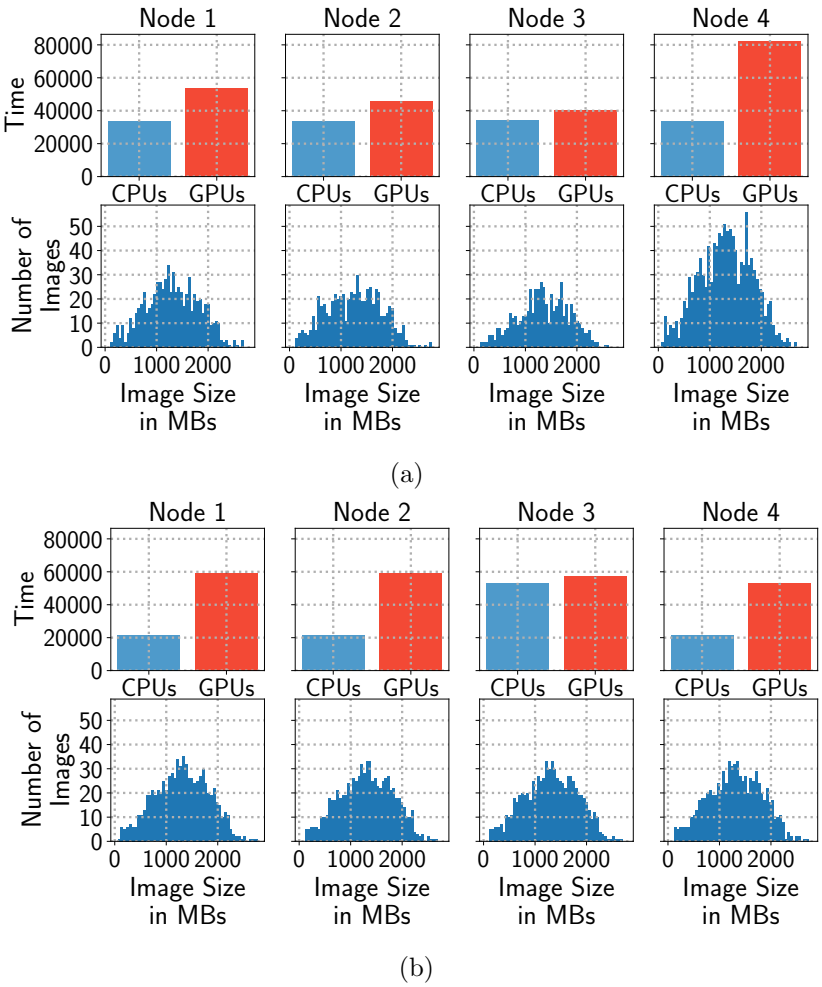
Similarly to the analysis in Design 1 and 2 we fitted data from Design 2.A to Eq. 4.1. For T_1 the fit gives $\alpha = 2.74 \times 10^{-2}$ and $\beta = 49.03$, with R^2 and S_{error} of 0.94 and 3.89 respectively. For T_2 the fit gives $\alpha = 4.8 \times 10^{-2}$ and $\beta = 87.36$, with $R^2 = 0.95$ and $S_{error} = 6.19$. Both models are therefore a good fit for the data and are validated.

The results of our experiments indicate that, on average and across multiple runs, there is a decrease in the execution time of T_1 and an increase in that of T_2 compared to Design 2. Design 2.A requires one queue more than Design 2 for T_1 and therefore more resources for Design 2.A implementation. This can explain the slowing of T_2 but not the speedup of T_1. This requires further investigation, measuring whether the performance fluctuations of compute nodes are larger than measured so far.

As discussed in §4.3.2, balancing of workflow execution differs between Design 2 and Design 2.A. Fig. 4.7a shows that each T_1 task can work on a different number of images but all T_1 tasks concurrently execute for a similar duration. The four distributions in Fig. 4.7a also show that this balancing can result in different input distributions for each compute node, affecting the total execution time of T_2 tasks on each node. Thus, Design 2 can create imbalances in the time to completion of T_2, as shown by the red bars in Fig. 4.7a.

Design 2.A addresses these imbalances by early binding images to compute nodes. Comparing the lower part of Fig. 4.7a and Fig. 4.7b shows the difference between the distributions of image size for each node between Design 2 and 2.A. In Design 2.A, due to the modeled correlation between time to completion and the size of the processed image, the similar distribution of the size of the images bound to each compute node balances the total processing time of the workflow across multiple nodes.

Note that Fig. 4.7 shows just one of the runs we perform for this experiment. Due to the random pulling of images from a global queue performed by Design 2, each run shows different distributions of image sizes across nodes, leading to large variations in the total execution time of the workflow. Fig. 4.7b shows also an abnormal behavior of one compute node: For images larger than 1.5GBs, Node 3 CPU performance is markedly



(a)

(b)

Figure 4.7: Execution time of T_1 (blue) and T_2 (red), and distributions of image size per node for (a) Design 2 and (b) Design 2.A.

slower than other nodes when executing T_1. Different from Design 2, Design 2.A can balance these fluctuations in T_1 as far as they don't starve T_2 tasks.

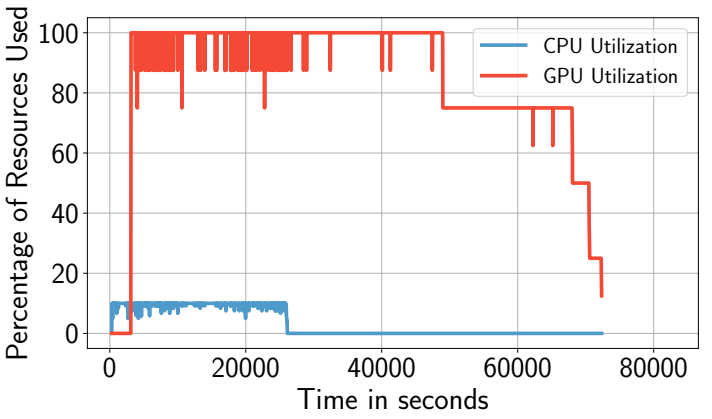
4.4.3 Experiment 2: Resource Utilization Across Designs

Resource utilization varies across Design 1, 2 and 2.A. In Design 1, the RTS (RADICAL-Pilot) is responsible for scheduling and executing tasks. T_1 is memory intensive and, as a consequence, we were able to concurrently execute 3 T_1 on each compute node, using only 3 of the 32 available cores. We were instead able to execute 2 T_2 concurrently on each node, using all the available GPUs. Assuming ideal concurrency among the 4 compute nodes we utilized in our experiments, the theoretical maximum utilization per

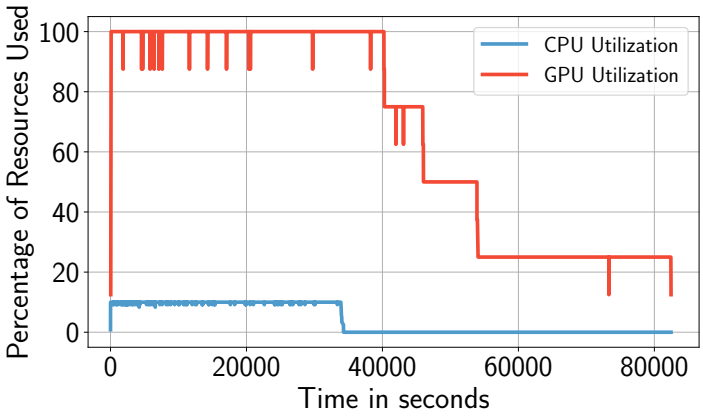
node would be 10.6% for CPUs and 100% for GPUs.

Fig. 4.8 shows the actual resource utilization, in percent of resource type for each design. The actual CPU utilization of Design 1 (Fig. 4.8a) closely approximates theoretical maximum utilization but GPU utilization is well below the theoretical 100%. GPUs are not utilized for almost an hour at the beginning of the execution and utilization decreases to 80% some time after half of the total execution was completed.

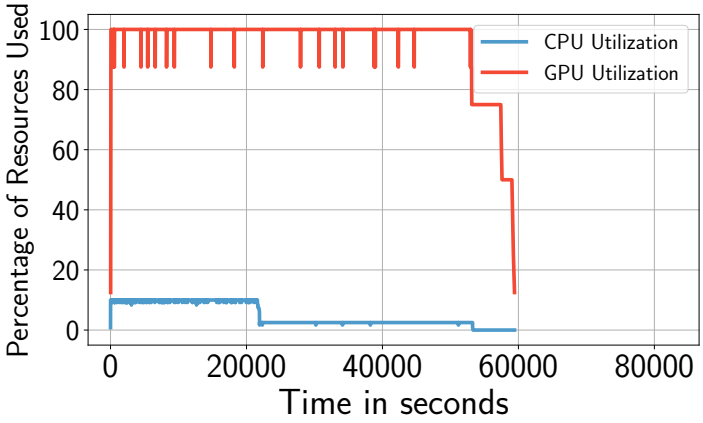
Analysis shows that RADICAL-Pilot's scheduler did not schedule GPU tasks at the start of the execution even if GPU resources were available. This points to an implementation issue and not to an inherent property of Design 1. The drop in GPU utilization is instead explained by noticing that, as explained in §4.3, GPU tasks were pinned to specific compute nodes to avoid I/O bottlenecks. Our experiments confirm that this indeed reduces utilization as some of the GPU tasks on some nodes take longer time to process than those on other nodes.



(a)



(b)



(c)

Figure 4.8: Percentage of CPU and GPU utilization for: (a) Design 1; (b) Design 2, and (3) Design 2.A.

Fig. 4.8b shows resource utilization for a specific run of Design 2. GPUs are utilized almost immediately as images are becoming available in the queues between T_1 and

T_2, and this quickly leads to fully utilized resources. CPUs are utilized for more time compared to Design 1, which is expected due to the longer execution times measured and explained in Experiment 1. In addition, two GPUs (25% GPU utilization) are used for more than $20k$ seconds compared to other GPUs. This shows that the additional execution time of that node was only due to the data size and not due to idle resource time.

Fig. 4.8c shows the resource utilization for a specific run of Design 2.A. For 3 compute nodes out of 4, CPU utilization is shorter than for Design 1 and 2. For the 4th compute node, CPU utilization is much longer as already explained when discussing T_1 execution time for Node 3 in Fig. 4.7, Experiment 1. As already mentioned, the anomalous behavior of Node 3 supports our hypothesis that compute node performance fluctuations can be much wider than expected.

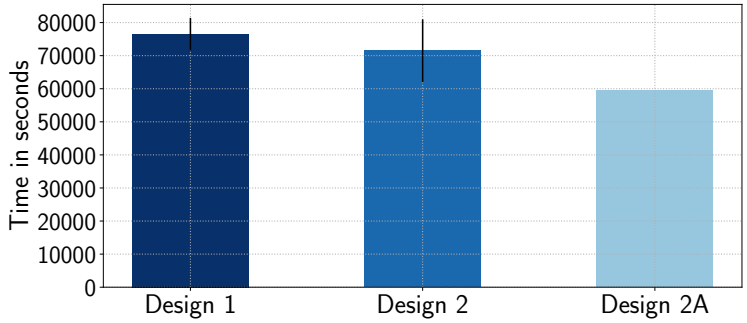
Fig. 4.8c shows that in Design 2.A GPUs are released faster compared to Design 1 and Design 2, leading to a GPU utilization above 90%. As explained in Experiment 1, this is due to differences in data balancing among designs. This shows the efficacy of two design choices for the concurrent execution of data-driven, compute-intense and heterogeneous workflows: early binding of data to node with balanced distribution of image size alongside the use of local filesystems for data sharing among tasks.

Note that drops in resource utilization are observed in all three designs.In Design 1, although both CPUs and GPUs were used, in some cases CPU utilization dropped to 6 cores. Our analysis showed that this happened when RADICAL-Pilot scheduled both CPU and GPU tasks, pointing to an inefficiency in the scheduler implementation. Design 2 and 2.A CPU utilization drops mostly by one CPU where multiple tiling tasks try to pull from the queue at the same time. This confirms our conclusions in Experiment 1 about resource competition between middleware and executing tasks. In all designs, there is no significant fluctuations in GPU utilization, although there are more often in Design 1 when CPU and GPUs are used concurrently.

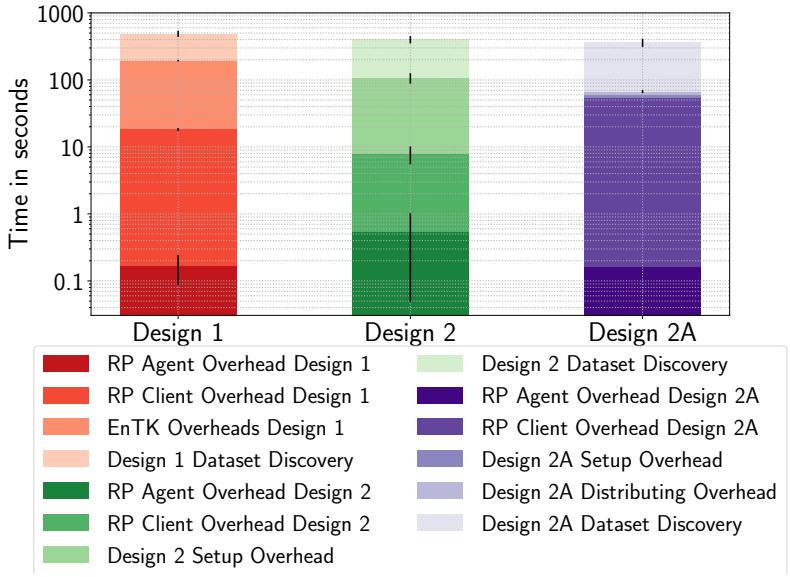
4.4.4 Experiment 3: Designs Implementation Overheads

.

This experiment studies how the total execution time of our use case workflow varies across Design 1, 2 and 2.A. Fig. 4.9a shows that Design 1 and 2 have similar total time to execution within error bars, while Design 2. A improves on Design 2 and both are substantially faster than Design 1. The discussion of Experiment 1 and 2 results explains how these differences relate to the differences in the execution time of T_1 and T_2 tasks, and execution concurrency respectively.



(a)



(b)

Figure 4.9: (a) Total execution time of Design 1, 2 and 2.A. Design 1 and 2 have similar performance, Design 2.A is the fastest. (b) Overheads of Design 1, 2 and 2.A are at least two orders of magnitude less than the total execution time.

Fig. 4.9b shows the overheads of each design implementation. All three designs

overheads are at least two orders of magnitude smaller than the total time to execution. A common overhead between all designs is the "Dataset Discovery Overhead". This overhead is the time needed to list the dataset and it is proportional to the size of the dataset. RADICAL-Pilot has two main components: Agent and Client. RADICAL-Pilot Agent's overhead is less than a second in all designs while RADICAL-Pilot Client's overhead is in the order of seconds for all three designs. The latter overhead is proportional to the number of tasks submitted simultaneously to RADICAL-Pilot Agent.

EnTK's overhead in Design 1 includes the time to: (1) create the workflow consisting of $3.1k$ independent pipelines; (2) start EnTK's components; and (3) submit the tasks that are ready to be executed to RADICAL-Pilot. This overhead is proportional to the number of tasks in the first stage of a pipeline, and the number of pipelines in the workflow. EnTK does not currently support partial workflow submission, which would allow us to submit the minimum number of tasks to fully utilize the resources before submitting the rest. Experiments should be performed to measure the offset between resource utilization optimization and increased time spent in communicating between EnTK and RADICAL-Pilot.

Fig. 4.9b shows that the dominant overheads of Design 2 is "Design 2 Setup Overhead". This overhead includes setting up and starting queues in each compute node, and starting and shutting down both T_1 and T_2 tasks on each node. Setting up and starting the queues accounts for most of the overhead as we use a conservative waiting time to assure that all the queues are up and ready.

Alongside the overheads already discussed, Design 2.A also introduces an overhead called "Design 2.A Distributing Overhead" when partitioning and distributing the dataset over separate nodes. The average time of this overhead is 7.5 seconds, with a standard deviation of 3.71 and is proportional to the dataset and the number of available compute nodes.

In general, Design 2.A shows the best and more stable performance, in terms of overheads, resource utilization, load balancing and total time to execution. Although Design 2 has similar overheads, even assuming minimization of Setup Overhead, it does

not guarantee load balancing as done instead by 2.A.Design 1 separates the execution in independent pipelines that are independently executed by the runtime system on any available resource. Based on the results of our analysis, both EnTK and RADICAL-Pilot can be configured to implement early binding of images to each compute node as done in Design 2.A. Nonetheless, Design 1 would still require executing a task for each image, imposing bootstrap and tear down overheads for each task.

4.5 Discussion and Conclusions

While designs 1, 2 and 2.A successfully support the execution of the workflow described in §4.2, our experiments show that for the metrics considered, Design 2.A is the one offering the better overall performance. Generalizing this result, use cases that are both data- and compute-intense benefit from early binding of data to compute node so to maximize data and compute affinity and equally balance input across nodes. This approach minimizes the overall time to completion of this type of workflows while maximizing resource utilization.

Our analysis also shows the limits of an approach where compute tasks are late bound to compute nodes. In program-based designs, the overhead of bootstrapping programs needs to be minimized, insuring that each task processes as much input as possible (e.g., images). In presence of large amount of data, late binding implies copying, replicating or accessing data over network and at runtime. We showed that, in contemporary HPC infrastructures, this is too costly both for resource utilization and total time to completion.

When executing workflows as part of a computational campaign, it is important to efficiently utilize available resources. Our design evaluation provides us with information on how to implement the workflows of a campaign effectively. In addition, our overheads characterization provides us with a baseline to evaluate production grade workflow implementations.

Chapter 5

Scientific Computational Campaigns on HPC Platforms

In the previous chapters, we discussed the requirements and support of data- and compute-intensive workflows on high performance computing (HPC) resources. When considering actual research, scientific projects require to execute multiple workflows at scale and potentially for long period of time. Usually, workflows are first developed, tested and executed for small scale problems which can be verified theoretically. When ready, tens to thousands workflows are executed, potentially concurrently, at production scale for long periods of time—up to several months—on the target HPC machine(s) while trying to achieve an objective. This poses unprecedented computing challenges, shifting the problem from enabling the effective and efficient execution of a single workflow at scale, to managing the execution of a computational campaign.

The challenges users face when executing a computational campaign include, but are not limited to, deciding an execution plan, submitting, executing and monitoring workflows, and transferring data among resources. Workflow submission requires users to have knowledge of submission systems, which generally differ among HPC resources. Furthermore, users have to connect to those resources to get information about the execution of workflows. Lastly, users have to establish an execution plan which defines when each workflow will be executed and on which HPC resource, with workflows potentially executing concurrently. A campaign manager which, given a computational campaign and a set of resources, creates an execution plan, submits, executes and monitors workflows on HPC resources can address those challenges.

Existing campaign managers make assumptions about the resources and middleware they are utilizing. In addition, they are monolithic software systems, despite their modular design, and tend to be domain specific. For example, PanDA [87], Pegasus [88],

and glideinWMS [89] are not easily extensible to use other capabilities or runtime systems. QCFractal [90] is built to be extensible and be able to interface with different workflow and workload management systems, but remains domain specific.

In response to the limitations of existing campaign managers—monolithic design, domain specific support, and dependence on specific resource and middleware—we design and prototype a new campaign manager (CM). As our CM has to support use cases from different scientific domains, such as molecular dynamics, earth sciences and high energy physics on potentially heterogeneous and diverse resources, the design and functionality space grow exponentially. A prototype allows us to identify the most important components and functionalities to support our use cases. Further, we can quickly implement and test the CM without necessarily considering any performance and data management requirements that may arise when executing an actual campaign on actual production infrastructures. Finally, a prototype allows us to tune parts of the CM with minimal engineering cost.

The CM prototype is domain agnostic as its design does not depend on requirements from any specific scientific domain. In addition, our prototype is designed by following the building blocks approach [18]. In this way, the prototype is agnostic and makes no assumptions about the workflow management framework used to manage the execution of individual workflows, as well as the type of resources used.

The chapter is organized as follows: § **??** discusses the current solutions in executing computational campaigns. In § 5.2, we discuss the supported use cases and the CM requirements. § 5.3 describes the building blocks approach and the design of the CM. In § 5.4 we describe the implementation of the CM prototype and we characterize its baseline performance. We close the chapter with our conclusions in Section 5.5.

5.1 Related Work

Currently, many software systems support computational campaigns. Among those, PanDA [87], DIRAC [91], QCFractal [90], and glideinWMS [89] are the ones with the widest adoption. These campaign managers, apart from GlideInWMS [89], are domain

specific and make assumptions about the underlying software stack and type of workflows to be executed. PanDA WMS [87] mainly supports the distributed data analysis of data produced by the ATLAS experiment [5] at the large hadron collider (LHC). PanDA has been adapted to other use cases and projects, but it requires major tailoring and dedicated support. This is also true for DIRAC [92], which mainly supports the LHCb Monte Carlo simulation production system at CERN. QCFractal [90] is a campaign manager developed to execute large-scale quantum chemistry campaigns. It allows users to define workflows based on the geometry of physical systems and execute calculations, hiding the specifics of the software used. GlideInWMS [89] is a more general purpose campaign manager as it was specifically designed to support different use cases.

Existing campaign managers also support a plethora of computing resources, including Grid resources, HPCs and Cloud. PanDA [87], glideinWMS [89] and DIRAC [91] mainly support grid resources. PanDA has been extended to support HPC [93, 94] and cloud resources [94], while glideinWMS [89] supports only cloud resources. QCFractal [90] supports a set of heterogeneous resources, including local campus clusters, and HPC and cloud resources.

Workflow management frameworks such as Pegasus [88] and Balsam [95] support the concurrent execution of multiple workflows on HPC resources. Both frameworks allow users to submit multiple workflows concurrently for execution but Balsam also provide multitenancy capabilities, where multiple users can submit jobs for execution under the same workflow. Despite this capabilities, both frameworks execute workflows as independent entities but not as part of a computational campaign with a well defined objective.

5.2 Campaign Manager Requirements

The space of data analysis of computational campaigns can be vast with different scientific objectives, types and number of workflows and resources. We use three real exemplar use cases with data analysis workflows to derive the requirements of the CM prototype. The first use case supports quantum chemistry campaigns (UC1), the second

earth science (UC2) and the third data analysis +++campaigns which analyze the data produced for the ATLAS experiment (UC3).

Quantum chemistry defines campaigns which execute workflows that perform chemical computations on molecules to calculate their properties, e.g., energy. These campaigns execute thousands of workflows, with up to 1000 executing at any given point in time, on a number of resources [6]. Workflows execution time varies between half-core hours up to 100-core hours, with different levels of concurrency. In addition, users have access to resources with several capabilities and not every workflow can be executed on every resource. During the campaign definition, users may provide an initial workflow priority and have to be sure that the campaign will finish within the given resource allocation.

Earth science campaigns analyze imagery acquired via sensors from different calendar years [4] to create time series of ecological events, executing one workflow per imagery from a calendar year. Their execution time varies from hours to a couple of days. Due to the volume of the data, users have access to a small number of resources where imagery is stored. While workflows analyze datasets based on the year they represent, a specific execution order is not necessary as long as all datasets are analyzed. The objective of this campaign is to analyze as many years of imagery as possible, thus they target high throughput imagery per unit of time.

The third use case supports the execution of data analysis workflows from the ATLAS experiment [5]. LHC produces data as particles collide inside its detectors. These data are then analyzed by executing hundreds to thousands workflows with different timescales and frequencies [7]. In addition, the analysis needs to happen within specific time boundaries, while the detector is not operational. For example, one of the use cases described in Ref. [7] analyzes thousands of different datasets with workflows running for months every yearly quarter. As a result, the objective of this use case campaign is to finish all of its workflows within a specific amount of time.

Based on these use cases, we derive a set of functional requirements for the CM prototype. Table 5.1 shows a summary of the functional requirements. The table defines a requirement identification number, states the requirement and provides a description of each requirement. In addition to this information, the table shows from which use

REQ ID	Requirement Algorithm	Use Case	Description
1	Support campaign with O(1k) workflows	UC1	The CM should be able to support campaigns with order of thousand workflows. Planning, execution and adaptation should be able to execute with such a campaign.
2	The CM should support at least two different planning algorithms.	G	Users/developers should be able to easily extend the planning capabilities with algorithms.
3	Support from 1 up to 100 resources	UC1-UC2	The CM should be able to execute workflows on multiple resources concurrently. These resources can either be actual or emulated resources.
4	Plan should be derived for heterogeneous and homogeneous static resources in 5 minutes	G	The plan should be derived as soon as the user provides a campaign description. Plan should be derived in less than 5 minutes.
5	Plan should be derived and adapted for heterogeneous/homogeneous dynamic resources in 5 minutes	UC1	The plan should be derived as soon as the user provides a campaign description. Plan should be derived in less than 5 minutes. In case the plan needs to be adapted, it should be adapted in less than 5 minutes.
6	Interface with different WMFs	G	The CM should be able to interface with different WMFs based on the specifics of the campaign.
7	Early bind workflows to resources	UC1	The user may need to bind workflows to specific resources before executing the campaign.
8	Campaign objective is configurable	UC1	While the campaign is executing, the objective may be adjusted based on user preferences.
9	Update the campaign during runtime	UC1-UC2	The user may want to update the campaign while it is executing to add/remove workflows.
10	Resources may be added or removed during runtime	UC1	The user may want to add/remove resources, she has gained/lost access to.

Table 5.1: Campaign manager functional requirements.

case the requirement is derived with "G" being a requirement that supports all three use cases.

5.3 Campaign Manager Design

This section discusses the design principles (§5.3.1) and the architecture (§5.3.2) of the campaign manager prototype.

5.3.1 Building Blocks Design Approach

The building blocks approach, as described in Ref. [18], defines four design principles for software systems. Those are self-sufficiency, interoperability, composability, and extensibility. Adhering to those principles during the design and implementation phase

of a software system allows for greater software sustainability. In addition, it allows users not to tailor their solutions to a specific software ecosystem and take full advantage of the plethora of software that supports scientific computing.

Self-sufficiency and interoperability describe the characteristics of software entities and functionalities. The entities are enough to stand alone and can be reduced to a specific abstraction. The functionalities of each building block are specific to the block and do not overlap with other blocks. As a result, the CM design defines a set of components and that allow it to execute a campaign to HPC resources without providing capabilities to execute the individual workflows of the campaign or acquire the necessary resources.

Composability and extensibility describe the communication and coordination between building blocks and the generality of the generality of the blocks components respectively. Blocks communicate information about their state, events and errors. Based on this information alone building blocks are coordinating with each other. As a result, the CM uses only the state of the workflow execution to create the campaign state. Furthermore, its components are designed to be general enough so that they can extended and interface with different workflow execution engines as well as software systems that execute computational campaigns.

5.3.2 Campaign Manager Components

We define three components as part of the CM prototype (see Figure 5.1): (1) a Planner; (2) an Enactor; and (3) a Bookkeeper. Each of these components supports a basic functionality of the CM, such as executing individual workflows and monitoring the campaign. In accordance to the building blocks approach, the CM functionality is limited in executing computational campaigns. As such, the CM does not provide any functionality to execute individual workflows or acquire resources. Workflow execution is managed by an existing workflow management framework (WMF) on HPC resources. In addition, resource acquisition and management is done by the WMF runtime system.

The Planner component derives an execution plan based on a planning algorithm and calculates the makespan for a campaign on a specific set of resources. The Planner

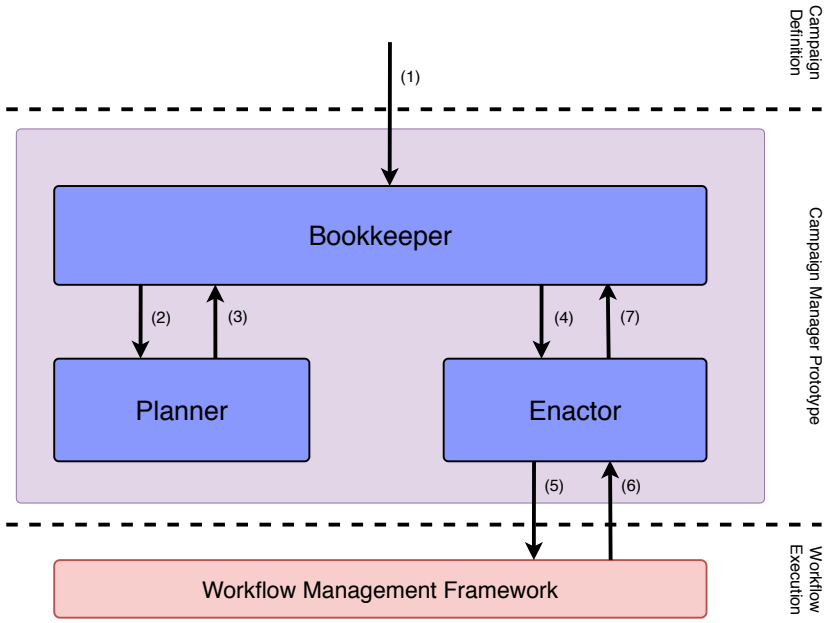


Figure 5.1: Reference Architecture of a Campaign Manager. Basic components of Campaign Manager (CM): (a) Planner, (b) Enactor and (c) Bookkeeper. CM communicates decisions to Workflows Management Framework. CM communicates with HPCs to execute parts of the campaign.

receives information about available resources and the workflows of the campaign from the Bookkeeper component. The Planner does not adhere to a specific planning algorithm, allowing different algorithms to be used, based on the campaign's requirements and objective.

The Bookkeeper component is responsible for monitoring the execution of the campaign. This component holds information about the state of the campaign, the execution plan, the availability of resources, and the campaign's objective. The state of the campaign is based on information about workflow execution provided by the Enactor component. In addition, the Bookkeeper knows the state of the resources utilized by the campaign at runtime and the state of the resources that are planned to be used. Based on this information, the Bookkeeper checks whether the campaign's objective can be achieved. Bookkeeper requests the Planner to update the plan when changes in the campaign happen that affect the effectiveness of a current plan or make the objective not achievable. For example, the availability of one or more resources can change during runtime, requiring a revision of the execution plan.

The Enactor component executes the campaign's workflows by interfacing with a workflow management system (WMF). The Enactor receives a set of workflows from the Bookkeeper, along with the resources on which the workflows will execute at a given point in time. Based on this information, the Enactor submits the workflows to the WMF for execution. The WMF, then acquires resources and executes the workflows.

Initially, the Bookkeeper receives the description of a campaign (Figure 5.1 step 1) and passes information about resources and workflows to the Planner (Figure 5.1 step 2). As soon as a plan is calculated, it is passed to the Bookkeeper (Figure 5.1 step 3). Then, the Bookkeeper passes the workflows and resources descriptions to the Enactor (Figure 5.1 step 4) and the Enactor passes that information to the selected WMF for execution (Figure 5.1 step 5). When a workflow finishes to execute, the Enactor gets notified (Figure 5.1 step 6) and informs the Bookkeeper (Figure 5.1 step 7). Steps 4, 5, 6 and 7 repeat until all the workflows have finished executing.

5.4 Prototype Implementation and Performance Evaluation

Consistent with the design presented in §5.3, we define three main classes to implement the CM prototype. The Bookkeeper class implements the capabilities of the Bookkeeper component, the Enactor class interacts with a selected workflow engine to execute workflows on resources, and the Planner class implements planning algorithms. In addition, the prototype defines a state model for the campaign and the workflows. Figure 5.2 shows the class diagram of the prototype. Each class also defines a set of data structures to support the execution of campaigns.

The states models provide information about the state of the execution. The campaign state model (Figure 5.3a) has six states: (1) *new*, (2) *planning*, (3) *executing*, and (4) *done* or (5) *canceled* or (6) *failed*. A campaign is considered *new* when it is defined and no further action is taken. The campaign transitions to the *planning* state when a plan for its execution is derived. As soon as the execution of the campaign starts, the state of the campaign transitions to *executing*. The final states are possible termination states. When the campaign execution ends and the campaign objective is

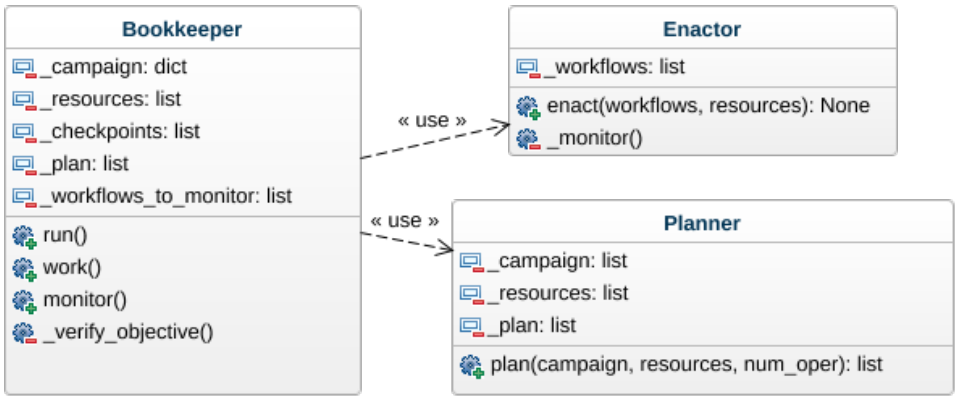
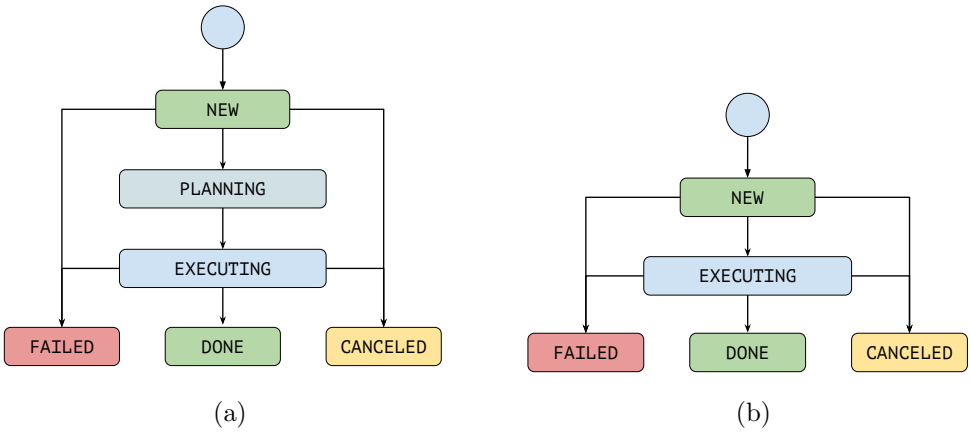


Figure 5.2: Class diagram of campaign manager prototype



(a) (b)

Figure 5.3: State diagrams for : 5.3a) a campaign; 5.3b) each workflow of a campaign.

achieved, the campaign transitions to the *done* state. If the objective cannot be achieved or there is a failure during the execution, the state of the campaign changes to *failed*. Lastly, the campaign state goes to *canceled* when the user cancels the execution of the campaign.

Similar to the campaign state model, the workflows state model (Figure 5.3b) defines five states: (1) *new*, (2) *executing*, and (3) *done* or (4) *canceled* or (5) *failed*. A workflow is in the *new* state when it is received by the Bookkeeper and transitions to the *executing* state when the Enactor submits the workflow for execution to the selected WMF. The workflow transitions to one of the final states—*done*, *canceled* or *failed*—based on the final state the workflow management system reports.

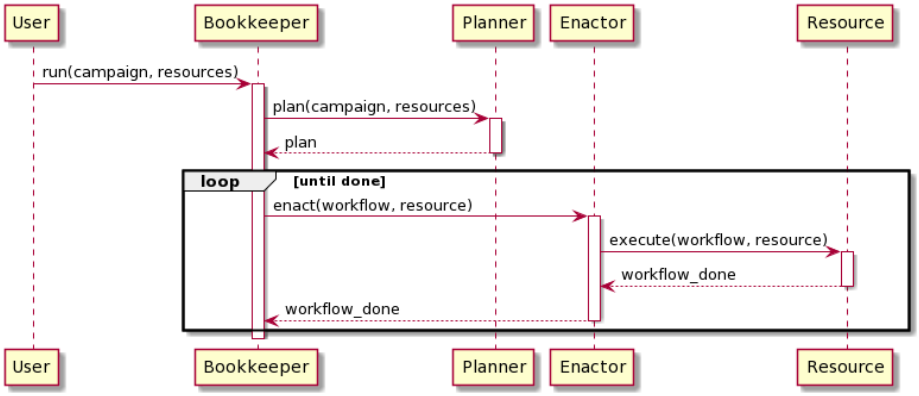


Figure 5.4: Sequence diagram for executing a computational campaign through the campaign manager prototype

The Bookkeeper class defines several methods to execute a campaign. The most important ones are: *run*, *work* and *monitor*. The *run* method initializes the campaign state to *new* and sets up the environment for executing the campaign. *Work* calls the Planner to produce a plan transitioning the state of the campaign to *planning*. After a plan is produced, *work* calls *_verify_objective* to verify if the objective of the campaign can be achieved, and either starts submitting workflows to the Enactor or transitions the campaign to the *failed* state. In addition, *work* pushes the submitted workflows to a data structure that the *monitor* method reads. The *monitor* method checks the state of the workflow which are executing and receives information from the Enactor via callbacks.

The Planner and Enactor classes define the methods for planning the execution of a campaign, executing workflows on resource and monitoring their execution. The Planner defines a *plan* method which implements a planning algorithm and returns a plan. The Enactor's *enact* method submits a workflow to the selected WMF. The *monitor* method of the Enactor checks periodically the state of the workflows that are executing and pushes state updates to the Bookkeeper via a callback.

Figure 5.4 shows the sequence diagram of the execution. Before the execution of a campaign, the Bookkeeper gets as input a campaign (a set of workflows) and a set of resources. As the user requests to execute the campaign, the Bookkeeper passes to the Planner the information it needs. When the plan is ready, the execution of the

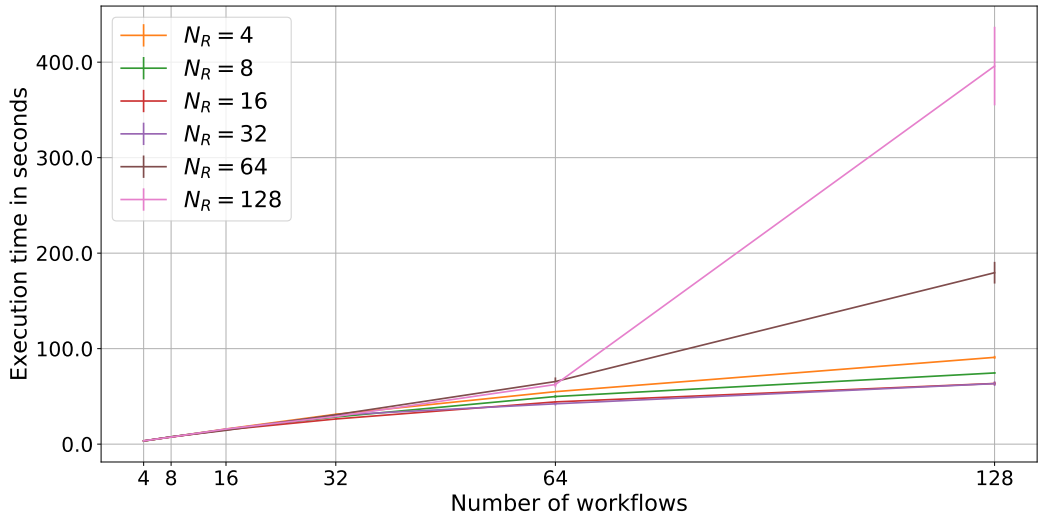


Figure 5.5: Execution time of simulating the execution of computational campaign with different number of workflows on different number of resources, N_R.

campaign begins with the Bookkeeper passing a workflow to the Enactor, along with a list of resources to use for the execution. As the Enactor executes the workflow, it pushes state updates to the Bookkeeper via callbacks. The Bookkeeper continues to pass workflows to the Enactor as resources become available, until there are no more workflows to execute.

Instead of an actual workflow management framework (WMF), the Enactor utilizes a modified version of the workload emulator proposed in Ref. [96]. The emulator allows us to define heterogeneous resources and emulate the execution of workloads on those resources. A workload is a set of independent tasks that can be executed concurrently. We modified the emulator to emulate the execution of a campaign which is a set of independent workflows. In addition, we extended the emulator execution capabilities with SimPy [97], a discrete time simulator. SimPy executes discrete simulations by producing events as time progresses. These events provide information about which workflows are executing or have finished to execute. The Enactor polls these events and pushes to the Bookkeeper the state of each workflow. As a result, we are able to simulate the execution of a campaign and validate the correctness of our design and its implementation.

We validated the correctness of our design by executing: 1) a campaign with an

achievable objective and 2) a campaign which objective cannot be achieved. In both cases, we provided the prototype with a campaign, a set of resources, an execution plan and an objective. The campaign manager executes the campaign and we validate that the campaign and workflows have reached the expected state. We created a validation suite which executes a set of tests and verifies the final state of the campaign and workflows. When the execution of the campaign can achieve the computational objective, the campaign and workflows transitioned to the *done* state as expected. Similarly when the execution of the campaign cannot achieve the computational objective, the campaign and workflows transitioned to the *failed* state.

We executed campaigns of different sizes on different number of resources where the objective of the campaign could be achieved and measured the execution time of the simulation. We verified that the simulation executed in a viable amount of time, as shown in Figure 5.5. We observed that the execution time of the simulation increases when more than 64 resources are utilized concurrently. Both the Enactor and the Bookkeeper have monitoring methods that traverse the list of executing workflows. In addition, the Enactor uses callbacks to inform the Bookkeeper about the state of the workflows. Above 64 resources, these operations start to dominate the execution time of the simulation. The prototype executed 128 workflows on 128 resources in $\approx$7 minutes, an amount of time we considered acceptable for experimental purposes.

Currently, the CM prototype is implemented in Python as an installable module. The source code is open and available on Github [98]. The CM currently offers a simple API for users who want to simulate the execution of a campaign. The Bookkeeper and Resources modules can be imported (seen under the example in [98]). The campaign currently is represented as a list of workflows where each workflow is a dictionary with the workflow description, a unique identifier per workflow and the runtime of the workflow on an 1 PetaFLOP resource. Resources are a list of dictionaries with a unique identifier, the performance and a label for each resource. In order to simulate the execution of the campaign, users pass the campaign and resources to the Bookkeeper and call the *run* method.

5.5 Conclusions

In this chapter, we motivated, discussed and validated a prototype for a software system that supports the execution of computational campaigns on HPC resources based on a set of use cases. We described the design and the basic components of the CM, a Bookkeeper, a Planner and an Enactor. Based on the selected design, we implemented a software prototype that simulates the execution of a campaign on resources. The prototype allows us to understand and correctly implement the necessary functionality of the CM.

The CM produces an execution plan for a computational via the Planner class. The Planner is able to support several planning algorithms. Although using any planning algorithm would satisfy the requirement of creating an execution plan, it is necessary to utilize an algorithm that is able to achieve the campaigns objective. In the next chapter, we investigate planning algorithms and evaluate the plans they produce for executing computational campaigns.

Chapter 6

Evaluating Computational Campaigns Planning Algorithms on HPC Platforms

Computational campaigns enact an execution plan to achieve a computational objective for given requirements and constraints. A computational objective is a set of values for a set of metrics that must be satisfied. Some of these metrics are time to completion or throughput defined as number of workflows executed per unit of time. Requirements, generally, describe the minimum amount and type of resources needed to execute each workflow of the campaign, while constraints are the conditions that bound the execution, such as resource availability, resource capacity or costs.

The objective of a campaign can be translated to a computational objective function that satisfies a lower or upper bound of a metric. Among the many metrics that can be considered, the most common one is the total time taken to execute all workflows of a campaign, also known as makespan. An execution plan of a campaign is a mapping between workflows and resources on which to execute those workflows. Calculating the makespan of a campaign for a given plan means verifying if said execution plan satisfies the computational objective function, within the given constraints.

We focus on computational campaigns that can concurrently utilize several high performance computing (HPC) infrastructures, i.e., supercomputers. Further, we consider infrastructures with homogeneous and heterogeneous resources, depending on the type of computational capabilities they offer, e.g., number of operations per second. Each HPC infrastructure can offer a set of homogeneous or heterogeneous resources and a set of HPC infrastructures can also be homogeneous or heterogeneous.

Uncertainty about the makespan of a campaign arises from two main reasons: resource dynamism and workflow runtime estimation. HPC resources are governed by policies

that affect their performance, such as power regulating policies [99] and filesystem and network multitenancy [100]. As a consequence, resource performance changes over time (i.e, is dynamic) which, in turn, creates uncertainty about the makespan of a campaign. Further, accurately estimating the runtime of each workflow of a campaign is challenging at best, impossible at worse. Users offer an estimation of each workflow runtime in the form of a range of possible values which, in turn, also creates uncertainty about the makespan of a campaign.

There is a plethora of algorithms which derive an execution plan by calculating the makespan of a workflow offline [13], i.e., before the workflow's execution. Planning for a campaign with N_W workflows is equivalent to planning for a workflow with N_t tasks. Offline algorithms usually take information about resource performance and task/workflow runtime as input, and use that information to produce a plan. As a result, the plan produced can be significantly affected by the uncertainty described above.

To the best of our knowledge, there is not a methodology that allows to compare planning algorithms on dynamic heterogeneous resources and in presence of workflows runtime estimation uncertainty. We select three algorithms, each representing a family of algorithms, and compare their performance in respect to the makespan of the plans they produce as well as how sensitive they are on resource dynamism and workflows runtime uncertainty. Based on the comparison, we make a case about which algorithm to select given a computational campaign, the execution environment, constraints and objective.

The chapter is organized as follows: §6.1 discusses related work. §6.2 discusses the calculation of the makespan of a campaign given the set of assumptions of this work. Section §6.3 discusses three makespan calculation algorithms and §6.4 compares their performance. Finally, section §6.5 presents our conclusions and a conceptual framework that will allow users to best select a planning algorithm. The chapter offers the two main contributions: (i) an experimental methodology to compare the performance of planning algorithms that is independent of the considered use case and computing framework; and (ii) a conceptual framework for selecting planning algorithms based on the algorithmic, campaign and resources characteristics.

6.1 Related Work

Several methods compare and evaluate algorithms that calculate and optimize the makespan of a workflow [13], including queuing networks [101, 102], domain specific languages [103, 104], and machine learning [105, 106]. Queuing networks are of limited use because they require from the user to provide a queuing network equivalent to the campaign. In the case the campaign contains only independent workflows, a single queuing system with multiple servers would be sufficient, but a campaign with complex dependencies among workflows may require expertise outside the user's domain to define the equivalent queuing network. Furthermore, using queuing methods to derive the makespan of a campaign requires to search for possible mappings and keep the one that optimizes the makespan. Domain specific languages approaches either require description of the resource usage of workflows [103], or to execute part of the campaign to obtain an execution "skeleton" [104]. When executing a campaign, workflows may require days to execute to obtain execution time information, and users rarely know the resource usage of their workflows to provide accurate enough information.

Some work has been done to evaluate different planning algorithms. Ref. [107] offers a survey of different algorithms for planning the execution of workflows on Grid resources. Ref. [108] experimentally compares a set of makespan-centric, heuristic-based planning algorithms. Algorithms are ranked based on how robust they are when task runtimes are uncertain. This approach is very close to our approach but we do not limit our evaluation to a specific type of algorithm and we also measure how planning algorithms perform in presence of resource dynamism and workflow runtime estimation uncertainty.

6.2 Calculating the Makespan of a Campaign

The way workflows of a given campaign are mapped to resources can affect the makespan calculation. Figure 6.1 shows an example of a campaign with workflows of different size and execution times, and the makespans that two different mappings produce. The makespan of the campaign on the left sub -figure is 20, while that on the right it is 16. In addition, Figure 6.1 also shows the relevance of the size of the workflows, i.e., the

number of resources they require, and that resources may be underutilized.

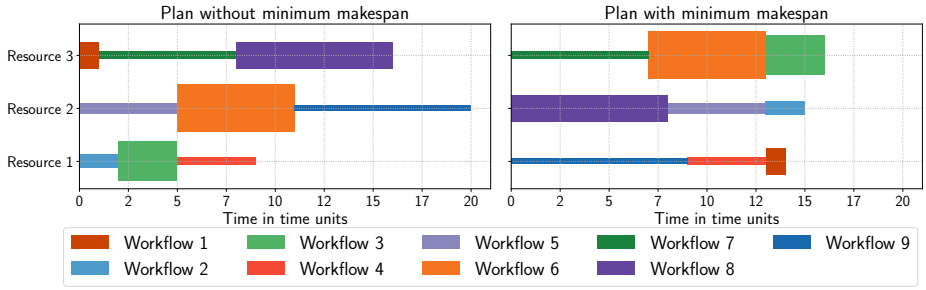


Figure 6.1: Comparison of different campaign execution plans. Makespan and resource utilization is different, based on workflow mapping on resources.

We make a set of assumptions when defining a model for the calculation of a campaign makespan: (1) a workflow is an atomic unit; (2) a workflow resource request is always sufficient to execute the workflow; (3) a resource is an aggregate of computing capabilities; (4) every workflow of a given campaign can be executed on the given resources; (5) random resource selection is based on a uniform distribution; (6) only one workflow can be executed on a resource at any point in time; and (7) workflows can be homogeneous or heterogeneous in time—the amount of time they are executing—and space—the amount of resources they require.

We denote a computational campaign as $C = [w_i : 1 \leq i \leq N_C]$, where w_i is a workflow and N_C is the total number of workflows, $R = [r_j : 1 \leq j \leq N_R]$ is a set of available resources, where r_j is a resource and N_R is the total number of available resources before the execution of the campaign. $M(C, R)$ is a mapping function which produces a map $[(w_i, r_j) : 1 \leq i \leq N_C, r_j \in R]$ of workflows onto resources. In addition, we denote the execution time of a workflow as Tx_{w_i}, the makespan of campaign C as TTX_C, and the makespan of campaign C for a given mapping function M as $TTX_C(M)$.

With a single resource, i.e., $N_R = 1$, the workflows of a campaign are executed sequentially, regardless of the execution order or whether the workflows are homogeneous or heterogeneous. As a result the makespan of the campaign is:

$$TTX_C = \sum_{i=1}^{N_C} Tx_{w_i} \tag{6.1}$$

With multiple resources, i.e., $1 < N_R < N_C$, the workflows of a campaign can be executed concurrently. With homogeneous resources, this is semantically equivalent to executing the campaign on a single resource large enough to allow concurrent workflow execution, where each workflow executes on a resource partition. Because of assumptions #4 and #7, the execution of spatially homogeneous or heterogeneous workflows has the same makespan. A random mapping of workflows onto resource has a makespan:

$$TTX_C(Random) \geq \frac{1}{N_R} \sum_{i=1}^{N_C} Tx_{w_i} \tag{6.2}$$

Given multiple homogeneous resources, when executing workflows that are heterogeneous in time and that can be spatially homogeneous or heterogeneous, the makespan of the campaign for a given mapping function M is:

$$TTX_C(M) = \max_{r_j \in R} \Big\{ \sum_{w_i \in M(C,r_j)} Tx_{w_i} \Big\} \tag{6.3}$$

Relaxing the assumption that resources are heterogeneous, in the performance they offer, affects the result of the mapping function M. As a result, Eq. 6.3 holds in calculating the makespan of the campaign.

6.3 Planning Algorithms

We investigate three algorithms: (i) the Heterogeneous Earlier Finish Time algorithm (HEFT) [19]; (ii) a genetic algorithm [20]; and (iii) a simple heuristic algorithm. These algorithms produce an execution plan before the actual execution of the campaign, and require workflow runtime and resource performance as input. Table 6.1 shows a summary of the algorithms characteristics.

6.3.1 Heterogeneous Earlier Finish Time (HEFT) algorithm

List scheduling algorithms represent a family of heuristic-based algorithms to schedule workflows [109, 110]. These algorithms assign priorities to tasks based on a heuristic and then order tasks based on their priority. To place tasks on resources, they traverse the

	HEFT	Genetic Algorithm	L2FF	Random
Decision Policy	Deterministic	Convergence Criteria	Deterministic	Deterministic
Initial State	Blank	Semi-random	Blank	Blank
Initial Information				
Workflow Operations	Yes	Yes	Yes	No
Resource Performance	Yes	Yes	Yes	No
Produced Knowledge	Resource availability	Resource availability	None	None

Table 6.1: Basic characteristics of selected planning algorithms.

Algorithm 4 Heterogeneous Earliest Finish Time (HEFT) algorithm

1: **procedure** HEFT(W,R) ▷ W and R are a set of workflows and resources respectively
2: `Calculate the computation cost` w_{tx}^{ij} `of each workflow for all resources`
3: `Assign` $rank_i = \overline{w_i} = \sum_{j=1}^{|R|} w_{tx}^{ij}/|R|$
4: `Sort workflows in decreasing order of` $rank_i$
5: **while** unscheduled workflows **do**
6: `Select the first workflow` $\tilde{w}$ `from the sorted list`
7: **for** $\forall r_j$ in R **do**
8: `Compute earliest finish time for` $\tilde{w}$ `on` r_j, $eft_{\tilde{w},r_j}$
9: **end for**
10: `Assign` $\tilde{w}$ `on` r_k `with` $\min(eft_{\tilde{w},r_j})$
11: **end while**
12: **end procedure**

ordered list of tasks and assign a task to a resource that can execute it. HEFT is a list scheduling algorithm [109] and calculates the makespan of a workflow on heterogeneous resources to produce its plan.

Pegasus [88] and ASKALON [111] utilize HEFT to derive workflow execution plans amongst other algoirthms, such as Round Robin, a genetic algorithm and a myopic algorithm. HEFT has been shown to provide better performance in terms of makespan minimization compared to other mapping algorithms [19, 108], e.g., genetic algorithms [111]. Furthermore, there is initial research to extend HEFT to dynamic resources [112], and to resources that provide CPU and GPUs [113].

HEFT makes two assumptions when used on workflows: (1) any task in a workflow can be executed on all available resources; and (2) all resources are initially available. HEFT's complexity is proportional to the number of dependencies among tasks and the number of resources offered. As we are interested in executing computational campaigns, our HEFT extension provides an execution plan based on workflows as atomic units instead of tasks. Algorithm 4 shows HEFT for placing independent workflows on resources.

HEFT has three main algorithmic characteristics: (1) creates a priority list of workflows; (2) produces an expectation of when resources are available; and (3) uses a deterministic heuristic. Initially, HEFT calculates the average execution time of a workflow on all resources and creates a priority list with the longest workflow first. In addition, HEFT initializes a vector with information about where each workflow is when a resource is available. Then, HEFT calculates when a workflow will finish at each resource and places that workflow on the resource that will finish it earlier. HEFT ends when all workflows are placed on a resource.

6.3.2 Genetic Algorithm

Genetic algorithms are another family of algorithms that can be used to calculate the makespan of workflows [109]. Genetic algorithms start from an initial set of possible solutions, and iterate and improve them across multiple iteration. In our case, genetic algorithms follow the following procedure: (i) create an initial population, where the population is a set of possible plans; (ii) evaluate the population members based on a fitness function, which calculates the makespan of the workflow; (iii) reproduce by selecting population members, either randomly or based on their fitness value, and generate a new set of possible plans; and (iv) mutate, where randomly selected tasks from a random population member are reassigned to resources.

The selected genetic algorithm [20] was developed to support the placement of independent tasks on heterogeneous resources. It assumes that all tasks that can be executed on all available resources, are independent and indivisible. These assumptions are consistent with the assumptions we made for the workflows in a campaign and we therefore extend this algorithm to support scientific campaigns. The pseudocode of the selected genetic algorithm is shown in Algorithm 5.

The members of the initial population are constructed either randomly or semi-randomly. Specifically for each individual member of the population, a percentage of the workflows are randomly assigned to resources, where the assignment is drawn from a uniform distribution and the rest of the workflows are assigned based on an earlier finish time (EFT) heuristic, similar to the one used by HEFT. The size of the population,

Algorithm 5 Genetic Algorithm

```
1: procedure GA(W, R)                    ▷ W and R are a set of workflows and resources respectively
2:     Initialize population
3:     while Conergence Criteria not met and #Gen < Total_Generations do
4:         Selection
5:         Reproduce
6:         Randomly mutate
7:     end while
8: end procedure
```

i.e., potential plans, is set to 20 members. Ref. [114] shown that a population size of 20 members reduces the computational load of the genetic algorithm without significantly impacting the final result.

Genetic algorithms use a fitness function to calculate the fitness of a population member to its environment, returning a value between 0 and 1. In our case, the fitness function calculates the distance of the makespan of a plan from the ideal makespan. This distance is defined as:

$$E = \sqrt{(\sum_{j=1}^{N_C} Tx_{w_j,r_j} - IM)^2} \tag{6.4}$$

where N_C is the number of workflows and IM is the ideal makespan. The ideal makespan is equal to:

$$IM = \frac{\sum_{i=1}^{N_C} Tx_{w_i}}{\sum_{j=1}^{N_R} r_j} \tag{6.5}$$

The fitness of a plan is the equal to $F = 1/E$ when $E > 0$, otherwise to $F = 1$.

The algorithm selects members to reproduce based on their fitness values. The fitness value of each member defines the probability of that member to be selected. When the selection is complete, the algorithm uses cyclic rotation [115] to generate new population members and replaces those with the lowest fitness. The algorithm stops evolving either when a member has fitness equal to 1 or after a specified number of iterations.

Our genetic algorithm has the following characteristics: (1) creates partial plans randomly; (2) produces an expectation of when resources are available; (3) evolves plans randomly; and (4) terminates based on a convergence criteria

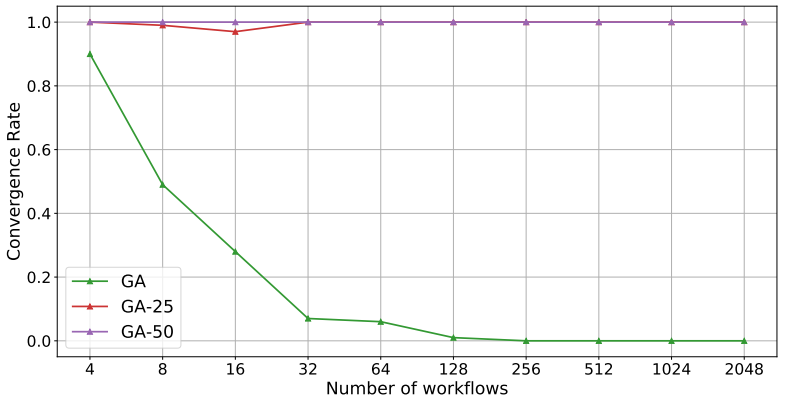
Genetic Algorithm Convergence Rate for Different Configurations

During the initialization phase of the genetic algorithm, we can select the percentage of workflows that will be assigned through the EFT heuristic. This is true for every member, i.e., plan, of the genetic algorithm population. We expect that the percentage of non-random placement affects the final plan selected by the genetic algorithm. We executed an experiment to measure the convergence rate of the genetic algorithm when planning a campaign with homogeneous workflows on homogeneous resources. In this case, distributing equally the workflows on resources produces a plan with the ideal makespan as defined in Eq. 6.5.

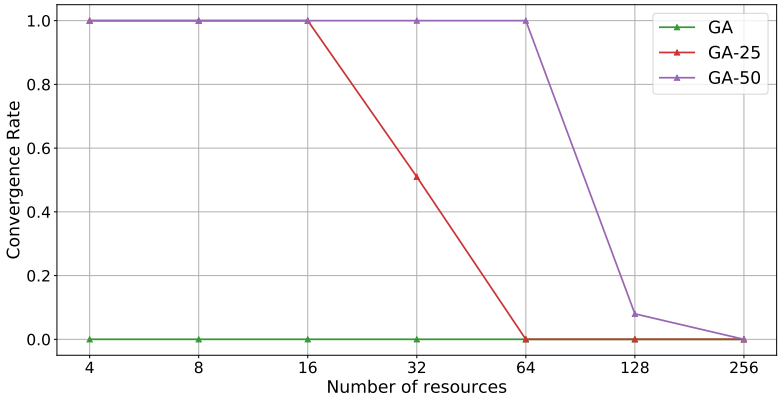
We use three configurations of our genetic algorithm: GA, GA-25 and GA-50. These configurations differ by the percentage of workflows non-randomly assigned to resources during the initialization of the population. GA makes no random workflow assignment to resources while GA-25 assigns 25% of the workflows non-randomly and 75% randomly, and GA-50 assigns 50% of the workflows non-randomly and 50% randomly.

Figure 6.2a shows the convergence rate of the genetic algorithm as the number of workflows changes while the number of resources is constant. When the population is initialized randomly, GA does not always equally distribute workflows to resources, as the convergence rate is always less than 1. As a result, GA does not always produce a plan with the ideal makespan. On the contrary, GA-25 and GA-50 have a convergence rate of 1. The convergence rate of GA-25 shows a drop for 8 and 16 workflows, but remains above 0.97 which allows us to conclude that GA-25 does produce plans with the ideal makespan.

Figure 6.2b shows the convergence rate of the genetic algorithm as the number of resources changes while the number of workflows is constant. When the population is randomly initialized, the genetic algorithm does not produce a plan that has the ideal makespan. The convergence rate of GA-25 is 1 up to 16 resources and drops to 0 for more than 32 resources. GA-50 convergence rate is 1 up to 64 resources and drops to less than 0.2 for 128 resources and equal to 0 for 256 resources. The drop is due to the random initial placement of workflows on resources. As the number of resources increase,



(a)



(b)

Figure 6.2: Convergence rate of Genetic algorithm for homogeneous campaigns on static homogeneous resources based on random initialization percentage. 6.2a) Different campaign sizes on 4 resources; 6.2b) Campaign with 1024 workflows and different number of resources.

the probability for GA to place less than N_C N_R workflows on a resource decreases.

Based the convergence rate, we expect that GA will not provide on average a plan with the ideal makespan whether the campaign size or the number of resources changes. GA-25 will provide plans with the ideal makespan when the number of resources is less than 16, regardless the number of workflows. Finally, GA-50 will provide plans with ideal makespan up to 64 resources. Note that less than ideal plans may still be desirable compared to those produced by other algorithms.

6.3.3 Longest to Fastest First Available Resource Algorithm

The last algorithm we considered is an heuristics algorithm that places the longest workflow on the fastest first available resource (L2FF) [96]. This algorithm sorts the workflows based on the number of operations or workflow runtime estimation, and sorts resources based on their performance. Then it places each workflow on the first fastest available resource, starting from the longest workflow. A resource is considered available when it has less or equal number of workflows than any other resource. When workflows and resources are sorted, the placement is equivalent to a modulo operation between the position of the workflow in the sorted list and the number of resources. Algorithm 6 show the pseudocode for this algorithm.

Algorithm 6 Longest to Fastest First (L2FF)

1: **procedure** L2FF(W, R) ▷ W and R are a set of workflows and resources respectively.
2: $W_{sorted} = sort(W)$
3: $R_{sorted} = sort(R)$
4: **for** w in W_{sorted} **do**
5: Assign w to r_k where $k = w_{idx} \mod N_R$
6: **end for**
7: **end procedure**

L2FF has the following algorithmic characteristics: (1) creates a priority list of workflows; and (2) uses a deterministic heuristic. Contrary to HEFT and GA, L2FF does not produce an expectation of when a resource is available because it does not calculate when each resource will be available based on the duration on the workflow that is executed on that resource. This reduces considerably the burden on the user and has the potential to simplify the design and implementation of the campaign manager. Depending on its performance, it may therefore be the preferable option to adopt.

6.4 Performance Evaluation of Planning Algorithms

We execute three experiments to evaluate and analyze the performance of the selected algorithms. We use random plans as the baseline of our experiments. The Random planner randomly selects a resource and places a workflow to that resource. In addition,

it requires no information about workflow runtime and resource performance.

The first experiment measures the makespan of a plan that uses HEFT, GA, L2FF and Random for different campaign and resource sizes. The second experiment measures the sensitivity of the makespan to resource dynamism, i.e., resource performance changes over time. We define sensitivity as the difference between the makespan measured via executing a campaign and the expected makespan given by the execution plan. Last, we measure the sensitivity of the makespan to workflow runtime uncertainty. This set of experiments provides a methodology to compare planning algorithms and decide which algorithm is more suitable, based on the computational requirements of a campaign, e.g., workflow size, resource performance and workflow runtime estimate.

We make a set of assumptions about resource performance and workflow runtime. These assumption are based on the use cases we support and the capabilities of the high performance computing resources we have access to. Resources can be homogeneous or heterogeneous and static or dynamic. Resources are homogeneous when they have the same performance in terms of operations per second, heterogeneous otherwise. Resources are static when their performance does not change over runtime and dynamic when it does. When resources are homogeneous, we assume that their performance is equal to 1 petaFLOP. We use four existing HPC resources as the basis for generating heterogeneous resources. These resources are PSC Bridges, SDSC Comet, TACC Stampede2 and TACC Frontera with performance of 1.3 [116], 2.7 [117], 10.6 [118] and 23.5 [118] petaFLOPS respectively.

Workflow execution runtime is based on the use case described in §4. We measured the runtime of the use case by executing the actual workflows on PSC Bridges. We adjusted the mean execution time and its variance to an resource with performance of 1 petaFLOP and obtained a mean runtime of 75000 seconds and a variance of 6000 seconds. For the purpose of our experiments, we assume that workflow runtimes of a campaign are drawn by a normal distribution with a mean and variance based on this use case.

6.4.1 Experiment 1: Measuring Makespan on Static Resources

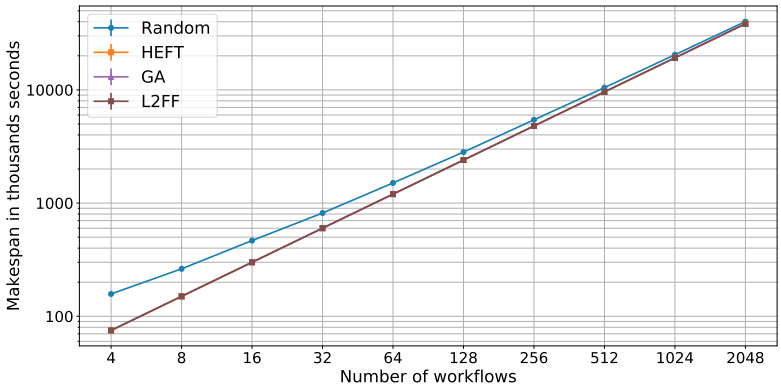
In our first experiment, we assume static resources and we use HEFT, GA, L2FF and Random to measure the makespan for different campaign and resource sizes. We use two configurations: in the first, we increase the size of the campaign from 4 to 2048 workflows, keeping the number of resources constant to 4; in the second configuration, we increase the number of resources from 4 to 256 resources and we fix the campaign size to 1024 workflows.

We measure the makespan of executing a homogeneous campaign with workflows that all have a runtime of 75000 seconds, on homogeneous resources with performance of 1 petaFLOP. In this case, an algorithm produces the minimum makespan when it equally distributes workflows to resources. Figure 6.3 shows the makespan for plans which use HEFT, GA (50% of the workflows are placed randomly), L2FF and Random (RA). Figure 6.3a varies the campaign size from 4 to 2048 workflows with 4 resources, and figure 6.3b varies the number of resources from 4 to 256 resources with a campaign size equals to 1024 workflows.

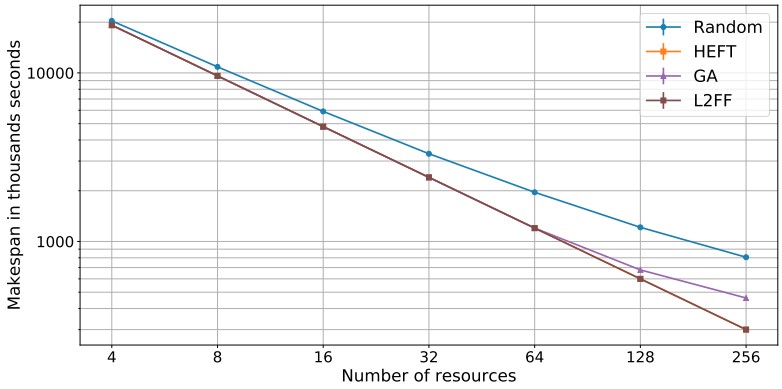
HEFT and L2FF provide better makespan than Random and GA. HEFT places a workflow to the resource that will finish it earlier. L2FF places equal number of workflows per resource for all available resources. As a result, both algorithms are able to equally distribute the workflows to the resources. The genetic algorithm shows worse makespan than HEFT and L2FF for more than 64 resources (Fig. 6.3b), verifying our expectation based on its convergence rate.

Figure 6.4 shows the makespan of the three algorithms for a campaign with heterogeneous workflows on heterogeneous resources. Workflows are heterogeneous when they have different runtime and resources are heterogeneous when they have different performance. Specifically, workflow runtimes are drawn randomly from a normal distribution with mean of 75000 seconds and variance of 6000 seconds. Heterogeneous resources are drawn randomly from the four resources described before.

HEFT provides up to an order of magnitude better makespan than L2FF and GA whether the campaign size (Fig. 6.4a) or the number of resources (Fig. 6.4b) changes.



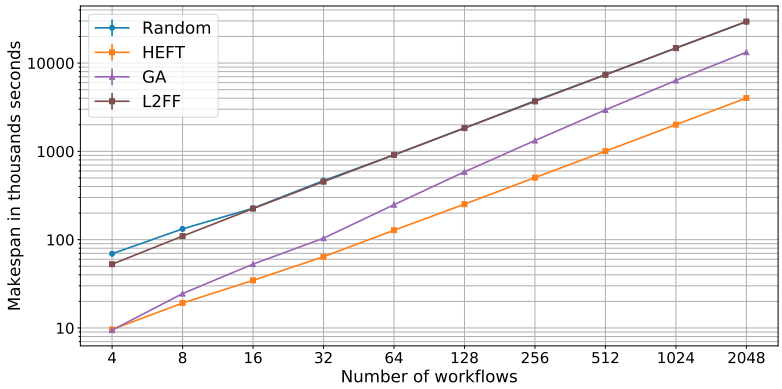
(a)



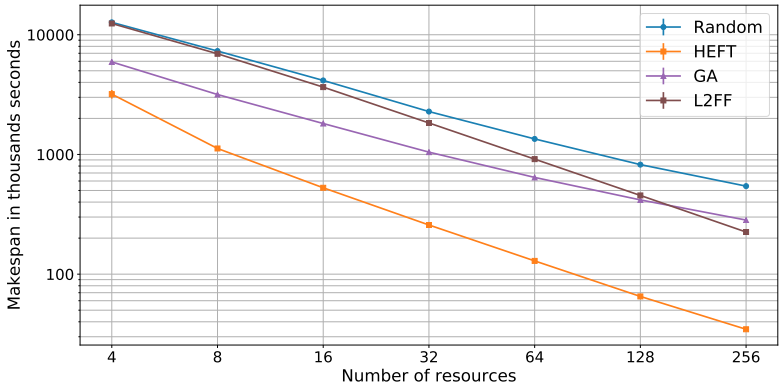
(b)

Figure 6.3: 6.3a Makespan of increasing number of homogeneous workflows on homogeneous resources. 6.3b Makespan of homogeneous campaign on different number of homogeneous resources.

In addition, GA produces plans with at least two times better makespan that L2FF when resources are constant and the number of workflows changes. As the number of resources increase, the difference between the plans that GA and L2FF produce reduces with L2FF providing a plan with better makespan than GA for 256 resources. For up to 64 resources, the heuristic of GA based on earlier finish time, reduces the effect of random initialization. This is not the case for 128 and 256 resources though, as the random part of the initialization may place very few or no workflows on some resources. L2FF places the same number of workflows per resource and, as a result, overutilizes less performant resources. HEFT instead ranks the workflows and places each workflow on the resource that will finish it earlier. As a result, HEFT tries to place each workflow on the resource that causes the minimum makespan increase for the given set of resources.



(a)



(b)

Figure 6.4: 6.4a) Makespan of 4 up to 2048 heterogeneous workflows on 4 heterogeneous resources; 6.4b) Makespan of a heterogeneous campaign with 1024 workflows on different 4 up to 256 heterogeneous resources.

In summary, we conclude that resource heterogeneity dominates the makespan performance among the considered algorithms. On homogeneous resources, HEFT, GA and L2FF plans produce plans with similar makespan to each other. On heterogeneous resources, HEFT makespan is at least an order of magnitude smaller than L2FF makespan and is more than two times smaller than GA makespan independent of workflow homogeneity or heterogeneity.

A common characteristic between HEFT and GA is that they create an expectation of when resources will be available, in contrast to L2FF which considers a resource available based on the number of workflows. Further, HEFT is deterministic, while GA is not, and HEFT always performed better than GA. Thus, the characteristics of the selected algorithms that affect the makespan performance, in order of importance,

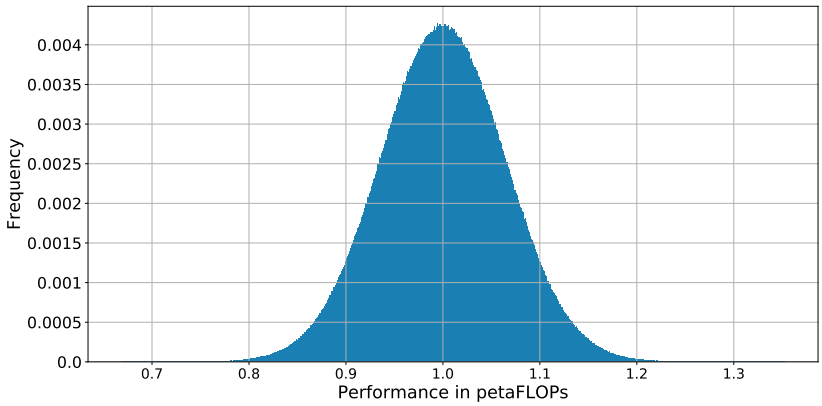


Figure 6.5: Resource performance distribution of a dynamic resource with 1 petaFLOP average performance.

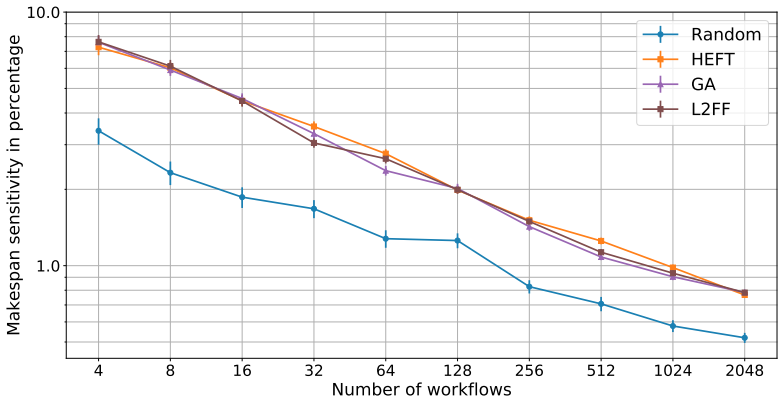
are: (1) estimating when a resource will be available; (2) deterministic or randomized heuristics; and (3) creating a workflow priority list based on workflow runtime

6.4.2 Experiment 2: Sensitivity of Makespan to Resource Dynamism

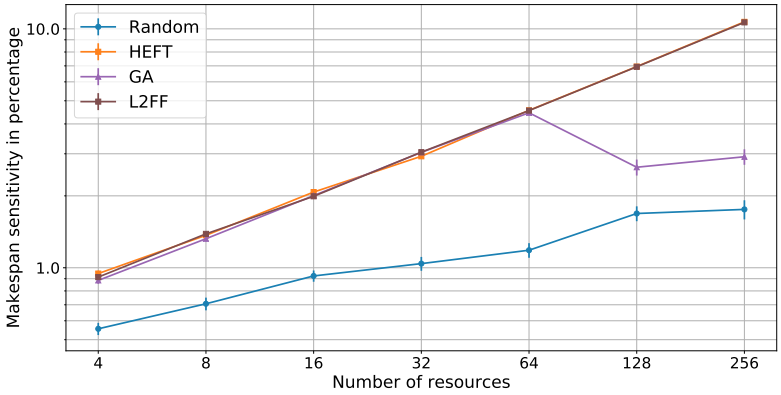
High performance computing (HPC) resources show performance variations for multiple reasons, including power constrained operations [99], network and shared filesystem congestion [100] and performance degradation [119]. Following established literature, the rate at which the performance of a resource changes is days to years [120]. Thus, we model resource dynamism as the daily change in the performance of a resource. We draw the performance values from a normal distribution, with a mean value equal to the performance of that resource and sigma to 6 % of the mean. Figure 6.5 shows the histogram of a dynamic resource performance with a mean value of 1 petaFLOP.

In experiment 2, we measure the sensitivity of the makespan using HEFT, L2FF, GA and Random to resource dynamism on homogeneous and heterogeneous resources. We define makespan sensitivity $Sens$ as the difference between the makespan measured via executing a campaign, $TTX_{C,Exec}$, and the expected makespan given by the execution plan, $TTX_C(M)$, thus $Sens = TTX_{C,Exec} - TTX_C(M)$. Based on the results of experiment 1, we expect $Sens$ to be similar between algorithms on homogeneous resources, as the three algorithms produce similar plans. On heterogeneous resources, we expect HEFT and L2FF to show higher sensitivity than GA and Random, as GA

and Random may produce a plan that does not utilize a resource which performance results to be significantly slower than expected at runtime. Figures 6.6 and 6.7 show the results for homogeneous workflows and homogeneous resources, and for heterogeneous workflows and heterogeneous resources respectively.



(a)



(b)

Figure 6.6: Sensitivity of makespan in percentage of homogeneous workflows on homogeneous resources. 6.6a increasing number of workflows and 4 resources; 6.6b 1024 workflows and increasing number of resources.

As per our expectation, the sensitivity of makespan among the three algorithms is similar and it is between 1 % and 10 % (see Fig. 6.6a and 6.6b). In addition, we observe a drop in GA sensitivity at 128 resources. GA produces different plans at every realization of the experiment, since it places a number of workflows to resources randomly and, as a result, some plans are affected less from resource dynamism. In addition, sensitivity drops as the number of workflows increases and the number of resources is constant (Fig. 6.6a). Sensitivity increases when the number of resources

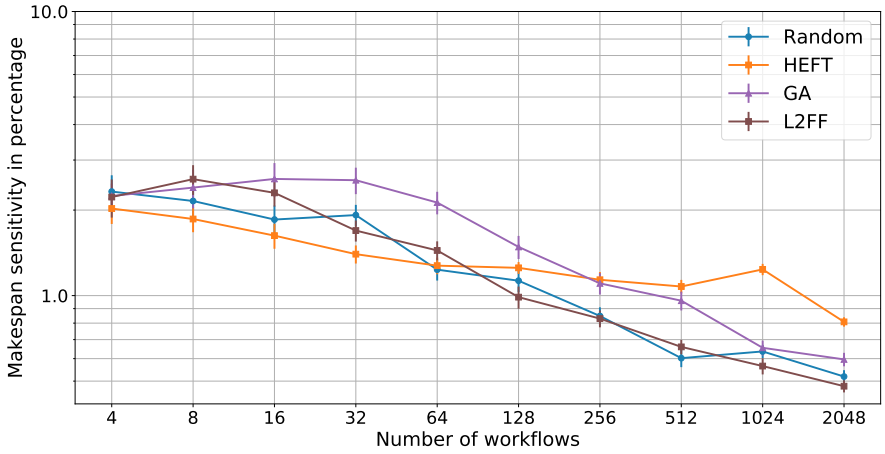
increases and the number of workflows is constant (Fig. 6.6b). Thus, the sensitivity decreases when the average number of workflows per resource increases, regardless for how workflows are placed onto resource by a plan.

Figure 6.7 shows the makespan sensitivity as a percentage for a campaign of heterogeneous workflows and heterogeneous resources. Makespan sensitivity shows similar values—from 3% to $< 1\%$—for all three algorithms in figure 6.7a, except for HEFT with more than 512 workflows contrary to our expectation. We explain this behavior by noticing that, as more workflows are placed on each resource, it becomes more probable that the effect of a potential slowdown of a workflow is reduced. This is because some workflows will finish faster and therefore reduce the sensitivity of the plan.

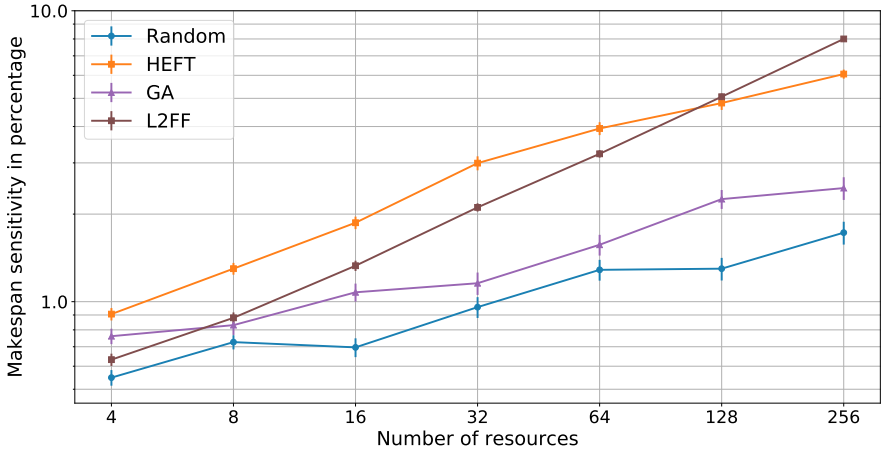
Figure 6.7b shows different sensitivity values for each algorithm. HEFT is at least two times more sensitive than GA and L2FF. HEFT places workflows on resources so that all resources are used for the least possible amount of time. As a result, a slow down on a resource will affect the makespan with HEFT more than that with GA and L2FF, which produce plans that overutilize some resources. Further, the sensitivity values for all algorithms are significantly lower than those observed when homogeneous resources were used, as a potential slowdown on a performant resource will not significantly affect the makespan of the campaign. In addition, we see that: (1) makespan sensitivity decreases as the number of workflows increases; and (2) makespan sensitivity increases as the number of resources increases. This further supports our conclusion that makespan sensitivity decreases as the average number of workflows per resource increases.

Based on the results in figure 6.7b, we see that the number of resources is a more dominant factor than resource heterogeneity for makespan sensitivity. HEFT-based plans are more sensitive that GA, L2FF and Random up to 64 resources. L2FF becomes more sensitive for more than 128 resources and is more sensitive to changes on less performant resources.

Experiment 2 measured how sensitive plans are when using HEFT, GA, L2FF and Random in presence of resource dynamism. On homogeneous resources, all algorithms show similar levels of sensitivity. GA sensitivity drops from more than 64 resources as some resources are overutilized. On heterogeneous resources, HEFT is generally



(a)



(b)

Figure 6.7: Sensitivity of makespan in percentage of heterogeneous workflows on heterogeneous resources. 6.7a increasing number of workflows and 4 resources; 6.7b 1024 workflows and increasing number of resources.

more sensitive as it tries to place a workflow to a resource with the least impact on the campaign makespan. L2FF becomes more sensitive than HEFT after 128 resources as it places equal number of workflows to resources, progressively overutilizing less performant resources. In addition, sensitivity is proportional to the number of resources and inverse proportional to the number of workflows, since it decreases as the number of workflows increases when using constant number of resources and increases when number of resources increases and campaign size is constant. Finally, we observed that a 6 % performance variation in the resources causes less than 10 % variation in makespan, regardless if resources are homogeneous or heterogeneous.

6.4.3 Experiment 3: Sensitivity of Makespan to Workflow Runtime Estimation Uncertainty

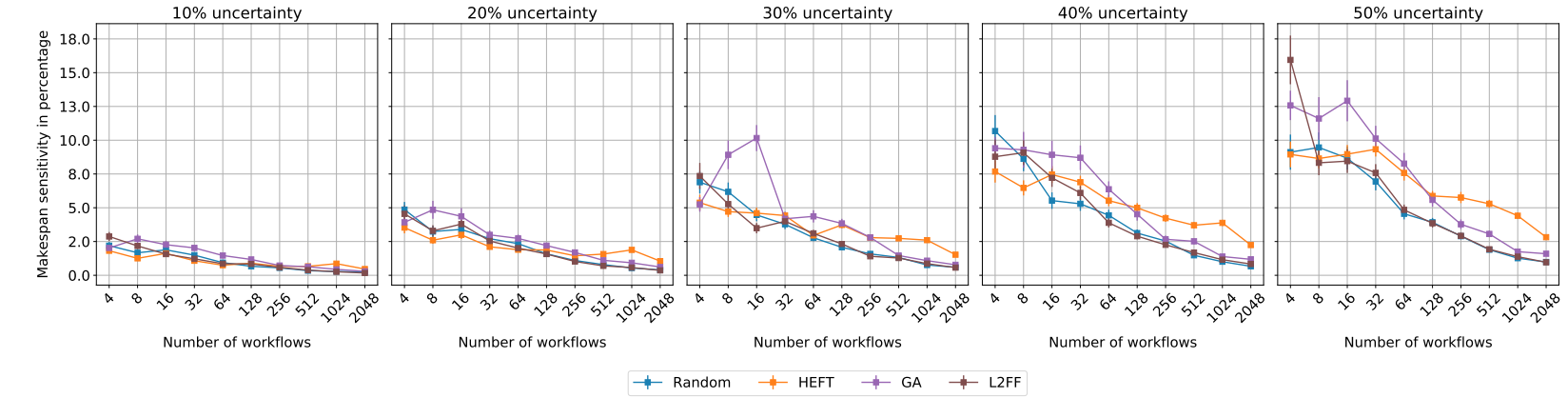
All selected algorithms, HEFT, GA and L2FF, require an estimation about the runtime of each workflow of the campaign. Users usually derive this information from empirical data, creating uncertainty in the workflow runtime estimation. This uncertainty has the potential to affect the makespan of the campaign, as the runtime estimation is used to either derive information about workflows runtime on resources (HEFT and GA) or rank workflows (L2FF). Similarly to experiment 2, we want to understand how different levels of uncertainty affect the makespan of the plans produced by HEFT, GA and L2FF

We introduce workflow runtime estimation uncertainty as the difference between the estimated and actual runtime of a workflow on a 1 petaFLOP resource. Specifically, we denote as u the level of uncertainty between $[0, 1]$, u' the uncertainty for a workflow drawn randomly from the range $[-u, u]$, and Tx_w the mean estimated runtime of a workflow. Thus, the actual runtime of a workflow is $Tx_{w'} = Tx_w \times (1 - u')$.

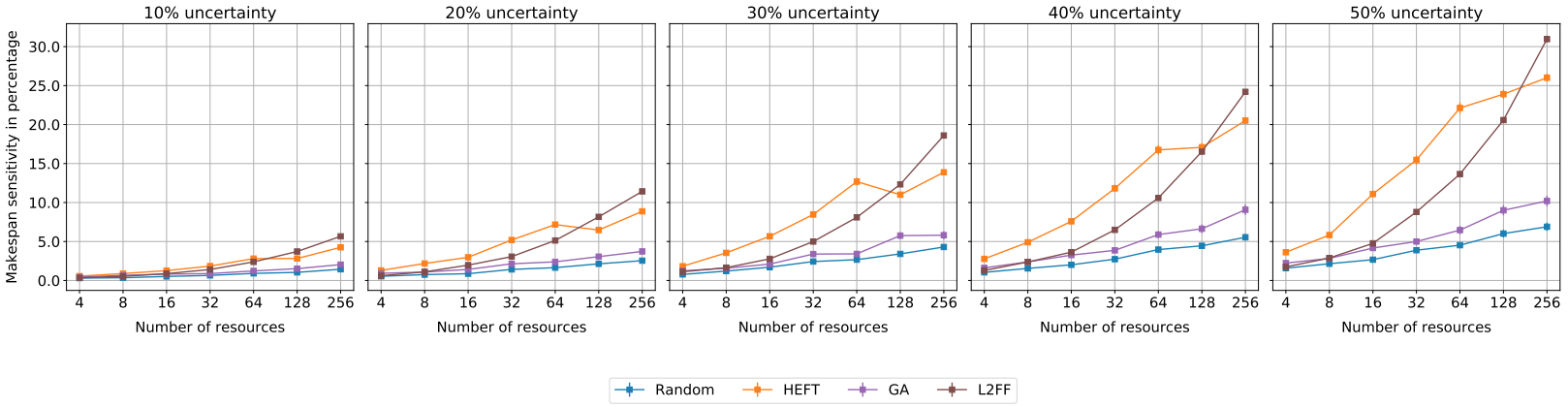
In experiment 3, we measure the sensitivity of makespan, as defined in experiment 2, to workflow runtime estimation uncertainty for level of uncertainty from 10 % to 50 %. As all algorithms produce similar plans on homogeneous resources, exposing the plans to the same variation should show the same effect, regardless of the used algorithm. This is supported by the fact that we measure the sensitivity after the execution of a campaign, and compare it to the expected makespan produced by the plan. Accordingly, we limit experiment 3 onlt to heterogeneous workflows executing on heterogeneous resources.

We expect sensitivity to workflow runtime estimation uncertainty to decrease as the number of workflows increases, and to increase as the number of resources decreases. As the number of workflows increases, more workflows are placed on a resource. As a result, on one hand it is more probable that a potential increase of the makespan due to a longer than expected workflow is hidden by other workflows than are shorter than expected. On the other hand, as the number of resources increases, fewer workflows are placed on each resource and a potentially larger workflow will have significant impact on the overall makespan. This is especially true when heterogeneous resources are used and

a long workflow is placed on a less performant resource. Figure 6.8 shows the results of experiment 3 with varying number of workflows (figure 6.8a) and varying number of resources (6.8b).



(a)



(b)

Figure 6.8: 6.8a Makespan sensitivity for different levels of uncertainty and different number of workflows on 4 heterogeneous resources; 6.8b Makespan sensitivity for different levels of uncertainty and different number of resources for 1024 workflows static resources.

As per our expectation, sensitivity to workflow uncertainty decreases when the number of workflows increase (Fig. 6.8a) and increases as the number of resources increase (Fig. 6.8b). This result, along with experiment 2, let us to conclude that as the average number of workflows per resource increases, the sensitivity value will decrease.

Sensitivity to makespan runtime uncertainty shows a wide range of values for HEFT and L2FF, but a significantly smaller range for GA and Random. Specifically, the sensitivity of plans using HEFT is between almost 0 % for 2048 workflows on 4 resource and 25 % for 1024 workflows on 256 resources. The sensitivity of plans using L2FF show a similar range to the one seen when using HEFT, with L2FF reaching 35 % for 1024 workflows on 256 resources. GA plans show increased sensitivity for a small number of workflows when the number of resources is constant ($\approx$13 %), and quickly drop close to 0 %.

HEFT ranks and L2FF sorts all the workflows based on runtime information, while GA uses that information only to map 50% of the workflows of each plan to resources, for each generation. When the runtime information is uncertain, any type of ordering will not necessarily be correct as the runtime estimation is not the same as the actual runtime. Since GA utilizes such information for half of the workflows at every generation, while the other half is placed on resources randomly, GA is less sensitive to workflow runtime uncertainty than HEFT and L2FF.

HEFT shows higher sensitivity than GA and L2FF above 128 workflows and 4 resources, and when the number of resources changes with 1024 workflows. In addition, we see higher sensitivity for plans that use HEFT and L2FF than plans that use GA when the number of resources increases. Both HEFT and L2FF create priority lists for the workflows and GA does not. As a result, HEFT and L2FF place large workflows on less performant resources. GA does not prioritize workflows and places a subset of workflows to resources randomly. As a consequence, the random order of the workflows with the random placement reduces the effect of the workflow runtime uncertainty.

We conclude that algorithms like HEFT and L2FF that prioritize workflows tend to be more sensitive to workflow runtime estimation uncertainty than algorithms that do not create a priority list, such as GA and Random. Workflow runtime uncertainty

can make the order decided by HEFT and L2FF invalid and, as a result, long workflow may be placed on less performant resource, increasing makespan. HEFT and L2FF are more sensitive than GA and random when the number of resources increases: 25 % and 35 % compared to less than 10 % for 1024 workflows and 256 resources. Further, all algorithms show similar sensitivity when the number of workflows increases. Sensitivity remains proportional to the number of resources and inverse proportional to the number of workflows. This, in conjunction to the results of experiment 2, allows us to conclude that the sensitivity slope is independent of the type of uncertainty.

6.5 Conclusions

In this chapter, we discuss and compare algorithms to plan the execution of a computational campaign on HPC resources. Specifically, we discuss the characteristics of three planning algorithms: HEFT, GA and L2FF. Our experimental methodology allows us to compare plans in terms of makespan performance, and their sensitivity to resource dynamism and workflow runtime uncertainty.

Our analysis shows that there are three algorithmic properties which affect the performance and sensitivity of planning algorithms. The first property is whether the algorithm estimates resource availability. HEFT and GA estimate when resources are available and show better makespan performance on homogeneous and heterogeneous resources than L2FF and random. The second property is whether the algorithm uses a deterministic or randomized heuristic to place workflows on resources. GA does not produce a plan that equally distributes workflows on resources when all workflows were initially placed randomly to resources. In addition, when using more than 128 resources, GA shows worse performance than L2Ff as it underutilized some of the resources. The third property is whether the algorithm creates a workflow priority list based on workflow runtime. HEFT and L2FF sort workflows based on their runtime, placing first the longer workflows on their lists. This priority list allows L2FF to place longer workflows to more performant resources and, as a result, L2Ff outperforms GA with more than 128 resources.

Our analysis of the makespan sensitivity shows that deterministic algorithms are more sensitive to resource dynamism or workflow runtime uncertainty than non-deterministic algorithms. HEFT and L2FF provide more sensitive plans than GA and Random. Further, sensitivity is proportional to the number of resources and inverse proportional to the number of workflows. Sensitivity to resource dynamism or workflow runtime uncertainty: (1) decreases as the number of workflows increases and the number of resources is constant ; and (2) increases when number of resources increases and the number of workflows is constant. Both HEFT and L2FF are more sensitive to workflow runtime uncertainty than GA since they prioritize workflows based on their runtime estimation.

We conclude that users should select different planning algorithms to derive an execution plan, depending on the properties of the campaign and resources. When resources are homogeneous, a planner like L2FF provides a plan with good makespan, is very simple to engineer and is not significantly more sensitive than algorithms like HEFT or GA. When resources are heterogeneous and the computational objective is to produce the best possible makespan, an deterministic algorithm like HEFT that derive information about resource availability and creates a priority list is a better candidate. If the objective of a campaign is to use a plan that is not very sensitive to resource dynamism or workflow runtime uncertainty, a non-deterministic algorithm like GA is a better candidate than the other two types of algorithms. Although GA may provide plans with worse makespan than an algorithm like HEFT, its plans are less sensitive to workflow runtime uncertainty.

Chapter 7

Conclusions

This dissertation investigated the execution of computational campaigns with data- and compute-intensive workflows on HPC resources. Our contributions to the field offer to domain scientists a better understanding of the capabilities needed to efficiently and effectively execute computational campaigns on HPC resources. Further, we provide the computational tools, methodologies and experimental understanding needed to design a software manager to plan and execute computational campaigns on high performance computing (HPC) platforms.

This dissertation was motivated by three exemplar use cases. These use cases have similar characteristics in terms of campaign size and resources they utilize. However, our results and solutions are independent of the scientific domain, size of the campaign, resources and computational objective of the campaign. We believe that using our campaign manager along with its planning capabilities is beneficial for any campaign that has a finite allocation on the resources it utilizes to achieve its computational objective.

7.1 Contributions

Computational campaigns on HPC resources can execute compute- or data- intensive workflows. Compute-intensive workflows are at the epicenter of high-performance scientific computing and account for the production of an immense amount of data. Compute intensive workflows execute either a single, very large and long-running executable or an ensemble of smaller compute-intensive tasks [8]. As the need for analyzing data on HPC resources increases, the efficient and effective execution of data-intensive workflows on HPC becomes important. Contrary to compute-intensive

workflows, data-intensive workflows execute a large number of short-running tasks in multiple stages which can be I/O, memory and compute bound.

As data-intensive workflows and applications are not well supported on HPC infrastructures, we showed how the Pilot-Abstraction can be used as an integrating concept between HPC resources and existing data analytics workflows management frameworks in chapter 2. We developed RP-YARN, extending RADICAL-Pilot, an implementation of the Pilot-Abstraction, to support Hadoop and Spark on HPC resources (see § 2.2). Our analysis in § 2.3.2 showed that RP-YARN reduces the time to completion of a data-intensive workflow. We concluded that the Pilot-Abstraction can increase HPC resources utilization when executing data analytics workflows with data analytics frameworks.

Existing Data analytics frameworks provide different programming abstractions and resource capabilities. Selecting a suitable framework becomes crucial to increase resource utilization and reduce development effort. In chapter 3, we investigated the use of three task-parallel frameworks—Spark, Dask and RADICAL-Pilot—to implement a range of algorithms for MD trajectory analysis, typically used in data analysis workflows. We found that, for embarrassingly parallel applications with coarse grained-tasks, the framework selection does not affect the performance. As shown in § 3.4.1, all three frameworks showed similar performance and speedups. For fine-grained data-parallel applications, a data-parallel framework is more suitable. § 3.4.2 shows our investigation of a MapReduce style algorithm and how Spark and Dask showed better performance than RADICAL-Pilot, with Spark offering linear speedups.

In addition to performance gains, developers should take into account usability and programmability aspects when selecting a task parallel framework. Spark and Dask, as shown in § 3.4, required to tune the number of tasks to achieve the desired performance (programmability aspect). The programming language of the selected framework affects its usability as it may introduce overheads due to transferring data between different languages space, e.g., between Python and Java as required by PySpark. In § 3.5, we provide a conceptual framework that enables application developers to evaluate and select task parallel frameworks with respect to application requirements. The conceptual

model, in conjunction with the Pilot-Abstraction, provide a methodology for application developers to maximize resource utilization while reducing the engineering effort needed to develop and execute data analysis workflows on HPC resources.

Some data-analysis workflows and applications are both data- and compute-intensive and, as such, are not well supported by frameworks specifically designed for data parallelism. In Chapter 4, we investigated three functionally equivalent designs to execute a data- and compute-intensive workflow. We characterized the designs by measuring task execution time (§4.4.1 and §4.4.2), resource utilization (§4.4.3), workflow time to completion and design overheads (§ 4.4.4). Based on our analysis, we found that late binding data to compute nodes, especially in the presence of large amount of data, implies copying, replicating or accessing data over network and at runtime. We showed that, in HPC infrastructures, this is too costly both for resource utilization and total time to completion. Further, we found that early binding of data to compute and equally balancing input across nodes reduced time to completion and increased resource utilization. This result provides information on the design properties that workflow management frameworks should have to efficiently and effectively execute data-driven and compute-intensive workflows. As a consequence, application developers should develop their workflows with a framework that allows to early bind and balance data to compute nodes.

Having established our understanding on how to effectively and efficiently support the execution of data- and compute-intensive workflows on HPC, we discussed the design of a campaign manager in Chapter 5. We discussed three actual computational campaigns, eliciting the requirements for a campaign manager. Based on those requirements, we designed and implemented a campaign manager prototype which is domain-agnostic and adheres to the building blocks design approach. We utilized a discrete event simulator (§5.4) to simulate the execution of a computational campaign on HPC resources. We validated the correctness of our design by executing campaigns of different sizes on different number of resources (§5.4).

As the campaign manager derives an execution plan for a computational campaign and given resources, in Chapter 6 we investigated three planning algorithms—HEFT, a genetic

algorithm (GA) and L2FF—to plan the execution of a campaign. We experimentally compared their plan performance in terms of makespan and how sensitive the plans are to resource dynamism and workflow runtime uncertainty (§6.4). As computational campaigns utilize heterogeneous resources, in §6.4.1 we showed that HEFT provides at least two times, and up to an order of magnitude better makespan than GA and L2FF. Further, we found that plans using HEFT are on average more sensitive to resource dynamism than plans using GA and L2FF (§6.4.2) and workflow runtime uncertainty (§6.4.3). Based on our experimental analysis, we discussed the algorithmic characteristics that affect the plan performance and sensitivity to resource dynamism and workflow runtime estimation uncertainty. Finally, we provided a conceptual model that users can use to select a planning algorithm based on their campaign characteristics, the resources they target and their computational objective.

This work has immediate impact on use cases from earth science domains that analyze very high resolution satellite imagery. Scientists from those domains want to execute computational campaigns with multiple workflows that analyze imagery from different calendar years and multiple satellite and terrestrial sources. New imagery becomes available at a constant low rate stream that, in turn, require constant analysis. Utilizing the work done to develop a campaign manager, scientists will be able to continuously and effectively executing workflows that analyze imagery as it becomes available.

7.2 Future Work

The work of this dissertation is invariant of the scientific domain and HPC resources that a campaign can utilize. As we move forward to support actual computational campaigns on HPC resources, we plan to pursue several lines of research and software development.

First, our campaign manager prototype and specifically its planner, assumes that all workflows are able to execute on all available resources. In reality, workflows may have different resource requirements, number and type of resources, and cannot execute on all

resources. We plan to extend the set of algorithms of the planner with algorithms that define resource requirements for workflows, and evaluate their makespan performance and sensitivity to resource dynamism.

Second, the planning algorithms we considered are early binding, i.e., they early bind workflows to resources. Late binding algorithms respond to events during runtime and their execution plan evolves as the campaign executes. Late binding algorithms can offer the same performance as early binding algorithms on homogeneous resources and we believe that they can offer similar performance to HEFT on heterogeneous resources. As a result, we plan to characterize the performance of diverse late binding algorithms and compare them to HEFT's performance.

Third, the amount of resources a workflow utilizes to minimize time to completion may not necessarily be known before runtime. In addition, there is initial research that utilizes machine learning (ML) algorithms and workflow time to completion historic data for workflow performance predictions [121]. We plan to investigate such ML approaches to provide the resource requirements of each workflow in a computational campaign which will then be used to plan the execution of a campaign. This approach has the potential to reduce the individual workflow time to completion and as a consequence increase the overall resource utilization across all the resources a campaign utilizes.

Fourth, our campaign manager prototype does not support the actual execution of campaigns but only their simulation. As we have an understanding of the requirements to support the execution of a campaign, we want to investigate the requirements to utilize an actual workflow management framework. RADICAL-EnTK will be our first choice, as it fits the requirements of the target use cases, and utilizes a pilot framework as runtime system. Further, we want to investigate the API or user interface requirements so at the definition of a computational campaign is independent of the workflow management framework used.

Finally, our campaign manager prototype supports the execution of a single campaign per user. As we move forward supporting actual computational campaigns, the campaign manager may be used in an multi-tenant environment. Currently, the campaign manager prototype can support multiple users where its user execute their campaign from their

user space. It does not support, though, the case where multiple users have access to the same campaign. As a result, we want to investigate the requirements of supporting multiple users through a single campaign manager executable independent of the number of campaigns executing at any given point in time.

References

[1] T. Cheatham and D. Roe. The impact of heterogeneous computing on workflows for biomolecular simulation and analysis. *Computing in Science Engineering*, 17(2):30–39, 2015.

[2] J. Dakka, K. Farkas-Pall, M. Turilli, D. W. Wright, P. V. Coveney, and S. Jha. Concurrent and adaptive extreme scale binding free energy calculations. In *2018 IEEE 14th International Conference on e-Science (e-Science)*, pages 189–200, Oct 2018.

[3] B.C. Gonçalves, B. Spitzbart, and H.J. Lynch. Sealnet: A fully-automated pack-ice seal detection pipeline for sub-meter satellite imagery. *Remote Sensing of Environment*, 239:111617, 2020.

[4] Ioannis Paraskevakos, Matteo Turrili, Bento Collares Gonçalves, Heather J. Lynch, and Shantenu Jha. Workflow design analysis for high resolution satellite image analysis. *CoRR*, abs/1905.09766, 2019.

[5] The atlas experiment. `http://www.atlas.ch`.

[6] Daniel Smith, Doaa Altarawy, Lori Burns, Matthew Welborn, Levi N. Naden, Logan Ward, Sam Ellis, and Thomas Crawford. The molssi qcarchive project: An open-source platform to compute, organize, and share quantum chemistry data, Mar 2020.

[7] Mikhail Borodin, Kaushik De, Jose Garcia Navarro, Dmitry Golubkov, Alexei Klimentov, Tadashi Maeno, David South, and Alexandre Vaniachine. Big Data Processing in the ATLAS Experiment: Use Cases and Experience. In *Procedia Computer Science*, volume 66, pages 609–618. Elsevier B.V., jan 2015.

[8] V. Balasubramanian, M. Turilli, W. Hu, M. Lefebvre, W. Lei, R. Modrak, G. Cervone, J. Tromp, and S. Jha. Harnessing the power of many: Extensible toolkit for scalable ensemble applications. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 536–545, May 2018.

[9] Apache Hadoop. `http://hadoop.apache.org/`, 2019.

[10] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[11] 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.

[12] J. L. Hellerstein, K. J. Kohlhoff, and D. E. Konerding. Science in the cloud: Accelerating discovery in the 21st century. *IEEE Internet Computing*, 16(4):64–68, 2012.

[13] Pingping Lu, Gongxuan Zhang, Zhaomeng Zhu, Xiumin Zhou, Jin Sun, and Junlong Zhou. A review of cost and makespan-aware workflow scheduling in clouds. *Journal of Circuits, Systems and Computers*, 28(06):1930006, 2019.

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

[15] Matteo Turilli, Mark Santcroos, and Shantenu Jha. A comprehensive perspective on pilot-job systems. *ACM Comput. Surv.*, 51(2), April 2018.

[16] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In Kathryn Huff and James Bergstra, editors, *Proceedings of the 14th Python in Science Conference*, pages 130 – 136, 2015.

[17] Andre Merzky, Matteo Turilli, Manuel Maldonado, Mark Santcroos, and Shantenu Jha. Using pilot systems to execute many task workloads on supercomputers. In *Job Scheduling Strategies for Parallel Processing*, pages 61–82, Cham, 2019. Springer International Publishing.

[18] M. Turilli, V. Balasubramanian, A. Merzky, I. Paraskevakos, and S. Jha. Middleware building blocks for workflow systems. *Computing in Science Engineering*, 21(4):62–75, July 2019.

[19] H. Topcuoglu, S. 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.

[20] Andrew J. Page and Thomas J. Naughton. Dynamic task scheduling using genetic algorithms for heterogeneous distributed computing. *Proceedings - 19th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2005*, 2005, 2005.

[21] Katherine Yelick, Susan Coghlan, Brent Draney, and Richard S. Canon. The Magellan Report on Cloud Computing for Science. Technical report, U.S. Department of Energy Office of Science Office of Advanced Scientific Computing Research (ASCR), December 2011.

[22] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, 2007.

[23] K. Grolinger, M. Hayes, W. A. Higashino, A. L'Heureux, D. S. Allison, and M. A. M. Capretz. Challenges for mapreduce in big data. In *2014 IEEE World Congress on Services*, pages 182–189, 2014.

[24] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient

distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[25] Mllib. `https://spark.apache.org/mllib/`, 2014.

[26] Ron O. Dror, Robert M. Dirks, J.P. Grossman, Huafeng Xu, and David E. Shaw. Biomolecular simulation: A computational microscope for molecular biology. *Annual Review of Biophysics*, 41(1):429–452, 2012. PMID: 22577825.

[27] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. Mdanalysis: A toolkit for the analysis of molecular dynamics simulations. *Journal of Computational Chemistry*, 32(10):2319–2327, 2011.

[28] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. In Sebastian Benthall and Scott Rostrup, editors, *Proceedings of the 15th Python in Science Conference*, pages 98 – 105, 2016.

[29] Daniel R. Roe and III Thomas E. Cheatham. Ptraj and cpptraj: Software for processing and analysis of molecular dynamics trajectory data. *Journal of Chemical Theory and Computation*, 9(7):3084–3095, 2013. PMID: 26583988.

[30] Tiankai Tu, C. A. Rendleman, D. W. Borhani, R. O. Dror, J. Gullingsrud, M. O. Jensen, J. L. Klepeis, P. Maragakis, P. Miller, K. A. Stafford, and D. E. Shaw. A scalable parallel framework for analyzing terascale molecular dynamics simulation trajectories. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12, Nov 2008.

[31] Andre Merzky, Matteo Turilli, Manuel Maldonado, and Shantenu Jha. Design and performance characterization of RADICAL-Pilot on titan. *Arxiv*, 2018. `https://arxiv.org/abs/1801.01843`.

[32] Apache Hadoop Project. Hadoop on demand. `http://hadoop.apache.org/docs/r0.18.3/hod.html`, 2008.

[33] William Clay Moody, Linh Bao Ngo, Edward Duffy, and Amy Apon. Jummp: Job uninterrupted maneuverable mapreduce platform. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–8, 2013.

[34] Al Chu. Magpie. `https://github.com/chu11/magpie`, 2015.

[35] Sriram Krishnan, Mahidhar Tatineni, and Chaitanya Baru. Myhadoop - hadoop-on-demand on traditional hpc resources. Technical report, San Diego Supercomputer Center, 2011.

[36] Richard Shane Canon, Lavanya Ramakrishnan, and Jay Srinivasan. My cray can do that? supporting diverse workloads on the cray xe-6. In *Cray User Group*, 2012.

[37] Md Wasi-ur Rahman, Xiaoyi Lu, Nusrat Sharmin Islam, and Dhabaleswar K. (DK) Panda. Homr: A hybrid approach to exploit maximum overlapping in mapreduce over high performance interconnects. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 33–42, New York, NY, USA, 2014. ACM.

[38] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[39] A. Treikalis, A. Merzky, H. Chen, T. S. Lee, D. M. York, and S. Jha. Repex: A flexible framework for scalable replica exchange molecular dynamics simulations. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 628–637, Aug 2016.

[40] Anjani Ragothaman, Sairam Chowdary Boddu, Nayong Kim, Wei Feinstein, Michal Brylinski, Shantenu Jha, and Joohyun Kim. Developing eThread Pipeline Using SAGA- Pilot Abstraction for Large-Scale Structural Bioinformatics. *BioMed Research International*, 2014.

[41] Soon-Heum Ko, Nayong Kim, Shantenu Jha, Dimitris E. Nikitopoulos, and Dorel Moldovan. Numerical Experiments of Solving Moderate-velocity Flow Field Using a Hybrid Computational Fluid Dynamics Molecular Dynamics Approach. *Mechanical Science and Technology*, 1(28):245–253, 2014.

[42] Niall Gaffney, Christopher Jordan, and Weijia Xu. Wrangler introduction. `https://portal.tacc.utexas.edu/user-news/-/news/101680`, 2014.

[43] Llama – low latency application master. `http://cloudera.github.io/llama/`, 2013.

[44] Apache Slider. `http://slider.incubator.apache.org/`, 2014.

[45] Andre Merzky, Ole Weidner, and Shantenu Jha. Saga: A standardized access layer to heterogeneous distributed computing infrastructure. *SoftwareX*, 1-2:3 – 8, 2015.

[46] Arun C. Murthy, Vinod Kumar Vavilapalli, Doug Eadline, Joseph Niemiec, and Jeff Markham. *Apache Hadoop YARN: Moving Beyond MapReduce and Batch Processing with Apache Hadoop 2*. Addison-Wesley Professional, 1st edition, 2014.

[47] Stampede. `https://portal.tacc.utexas.edu/user-guides/stampede`, 2015.

[48] V. Balasubramanian, I. Bethune, A. Shkurti, E. Breitmoser, E. Hruska, C. Clementi, C. Laughton, and S. Jha. Extasy: Scalable and flexible coupling of md simulations and advanced sampling techniques. In *2016 IEEE 12th International Conference on e-Science (e-Science)*, pages 361–370, 2016.

[49] Lisandro Dalcín, Rodrigo Paz, and Mario Storti. Mpi for python. *Journal of Parallel and Distributed Computing*, 65(9):1108 – 1115, 2005.

[50] Daniel R. Roe and Thomas E. Cheatham III. Parallelization of cpptraj enables large scale analysis of molecular dynamics trajectory data. *Journal of Computational Chemistry*, 39(25):2110–2117, 2018.

[51] Shujie Fan, Max Linke, Ioannis Paraskevakos, Richard J. Gowers, Michael Gecht, and Oliver Beckstein. PMDA - Parallel Molecular Dynamics Analysis. In Chris Calloway, David Lippa, Dillon Niederhut, and David Shupe, editors, *Proceedings of the 18th Python in Science Conference*, pages 134 – 142, 2019.

[52] Semen O. Yesylevskyy. Pteros 2.0: Evolution of the fast parallel molecular analysis library for c++ and python. *Journal of Computational Chemistry*, 36(19):1480–1488, 2015.

[53] Konrad Hinsen, Eric Pellegrini, Sławomir Stachura, and Gerald R. Kneller. nmoldyn 3: Using task farming for a parallel spectroscopy-oriented analysis of molecular dynamics simulations. *Journal of Computational Chemistry*, 33(25):2043–2048, 2012.

[54] Cameron Mura and Charles E. McAnany. An introduction to biomolecular simulations and docking. *Molecular Simulation*, 40(10-11):732–764, 2014.

[55] Sean L. Seyler, Avishek Kumar, M. F. Thorpe, and Oliver Beckstein. Path similarity analysis: A method for quantifying macromolecular pathways. *PLoS Comput Biol*, 11(10):1–37, 10 2015.

[56] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

[57] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python. `"http://www.scipy.org/"`, 2001–.

[58] Daniel P. Huttenlocher, Gregory A. Klanderman, and William J Rucklidge. Comparing images using the hausdorff distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(9):850–863, 1993.

[59] Shantenu Jha, Judy Qiu, André Luckow, Pradeep Kumar Mantha, and Geoffrey Charles Fox. A tale of two data-intensive paradigms: Applications, abstractions, and architectures. *Proceedings of 3rd IEEE Internation Congress of Big Data*, abs/1403.1528, 2014.

[60] Supun Kamburugamuve, Pulasthi Wickramasinghe, Saliya Ekanayake†, and Geoffrey C. Fox. Anatomy of machine learning algorithm implementations in mpi, spark, and flink. In *Technical Report*, Indiana University, Bloomington, 2017.

[61] Shantenu Jha, Daniel S. Katz, Andre Luckow, Neil Chue Hong, Omer Rana, and Yogesh Simmhan. Introducing distributed dynamic data-intensive (d3) science: Understanding applications and infrastructure. *Concurrency and Computation: Practice and Experience*, pages e4032–n/a, 2017. e4032 cpe.4032.

[62] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris

Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(34):1–7, 2016.

[63] 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.

[64] Stephen M. Omohundro. Five balltree construction algorithms. Technical report, International Computer Science Institute, 1989.

[65] Scikit-Learn: Nearest Neighbors. `http://scikit-learn.org/stable/modules/neighbors.html`, 2016.

[66] Hoang Vo, Jun Kong, Dejun Teng, Yanhui Liang, Ablimit Aji, George Teodoro, and Fusheng Wang. Mareia: a cloud mapreduce based high performance whole slide image analysis framework. *Distributed and Parallel Databases*, Jul 2018.

[67] Zhao Zhang, Kyle Barbary, Frank A. Nothaft, Evan R. Sparks, Oliver Zahn, Michael J. Franklin, David A. Patterson, and Saul Perlmutter. Kira: Processing Astronomy Imagery Using Big Data Technology. *IEEE Transactions on Big Data*, pages 1–1, 2016.

[68] Yuzhong Yan and Lei Huang. *Large-Scale Image Processing Research Cloud.* 2014.

[69] Raul Ramos-Pollan, Fabio A. Gonzalez, Juan C. Caicedo, Angel Cruz-Roa, Jorge E. Camargo, Jorge A. Vanegas, Santiago A. Perez, Jose David Bermeo, Juan Sebastian Otalora, Paola K. Rozo, and John E. Arevalo. BIGS: A framework for large-scale image processing and analysis over distributed and heterogeneous computing resources. In *2012 IEEE 8th International Conference on E-Science*, pages 1–8. IEEE, oct 2012.

[70] Antonella Galizia, Daniele D'Agostino, and Andrea Clematis. An MPI–CUDA library for image processing on HPC architectures. *Journal of Computational and Applied Mathematics*, 273:414–427, jan 2015.

[71] Amir Gholami, Andreas Mang, Klaudius Scheufele, Christos Davatzikos, Miriam Mehl, and George Biros. A framework for scalable biophysics-based image analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 19:1–19:13, New York, NY, USA, 2017. ACM.

[72] Rafael Vescovi, Ming Du, Vincent de Andrade, William Scullin, Gürsoy Doğa, and Chris Jacobsen. Tomosaic: efficient acquisition and reconstruction of teravoxel tomography data using limited-size synchrotron X-ray beams. *Journal of Synchrotron Radiation*, 25:1478–1489, aug 2018.

[73] Steve Petruzza, Aniketh Venkat, Attila Gyulassy, Giorgio Scorzelli, Frederick Federer, Alessandra Angelucci, Valerio Pascucci, and Peer-Timo Bremer. Isavs: Interactive scalable analysis and visualization system. In *SIGGRAPH Asia 2017 Symposium on Visualization*, SA '17, pages 18:1–18:8, New York, NY, USA, 2017. ACM.

[74] George Teodoro, Tony Pan, Tahsin M. Kurc, Jun Kong, Lee A.D. Cooper, Norbert Podhorszki, Scott Klasky, and Joel H. Saltz. High-throughput Analysis of Large Microscopy Image Datasets on CPU-GPU Cluster Platforms. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 103–114. IEEE, may 2013.

[75] K Ullas Karanth. Estimating tiger panthera tigris populations from camera-trap data using capture—recapture models. *Biological conservation*, 71(3):333–338, 1995.

[76] David Western, Rosemary Groom, and Jeffrey Worden. The impact of subdivision and sedentarization of pastoral lands on wildlife in an african savanna ecosystem. *Biological Conservation*, 142(11):2538–2546, 2009.

[77] Heather J. Lynch, Richard White, Andrew D. Black, and Ron Naveen. Detection, differentiation, and abundance estimation of penguin species by high-resolution satellite imagery. *Polar Biology*, 35(6):963–968, Jun 2012.

[78] Benjamin Kellenberger, Diego Marcos, and Devis Tuia. Detecting mammals in uav images: Best practices to address a substantially imbalanced dataset with deep learning. *Remote Sensing of Environment*, 216:139 – 153, 2018.

[79] Andrei Polzounov, Ilmira Terpugova, Deividas Skiparis, and Andrei Mihai. Right whale recognition using convolutional neural networks. *CoRR*, abs/1604.05605, 2016.

[80] Mohammad Sadegh Norouzzadeh, Anh Nguyen, Margaret Kosmala, Alexandra Swanson, Meredith S. Palmer, Craig Packer, and Jeff Clune. Automatically identifying, counting, and describing wild animals in camera-trap images with deep learning. *Proceedings of the National Academy of Sciences*, 115(25):E5716–E5725, 2018.

[81] Adriana Fabra and Virginia Gascón. The convention on the conservation of antarctic marine living resources (ccamlr) and the ecosystem approach. *The International Journal of Marine and Coastal Law*, 23(3):567–598, 2008.

[82] Helmut Hillebrand, Thomas Brey, Julian Gutt, Wilhelm Hagen, Katja Metfies, Bettina Meyer, and Aleksandra Lewandowska. Climate change: Warming impacts on marine biodiversity. In *Handbook on Marine Environment Protection*, pages 353–373. Springer, 2018.

[83] Keith Reid. Climate change impacts, vulnerabilities and adaptations: Southern ocean marine fisheries. *Impacts of climate change on fisheries and aquaculture*, page 363, 2019.

[84] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[85] Sean Gillies et al. Rasterio: geospatial raster i/o for Python programmers, 2013–.

[86] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[87] T Maeno. PanDA: distributed production and distributed analysis system for ATLAS. *Journal of Physics: Conference Series*, 119(6):062036, jul 2008.

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

[89] I Sfiligoi. glideinWMS—a generic pilot-based workload management system. *Journal of Physics: Conference Series*, 119(6):062044, jul 2008.

[90] QCFractal: A platform to compute, store, organize, and share large-scale quantum chemistry data. `http://docs.qcarchive.molssi.org/projects/qcfractal/en/latest/`, 2019.

[91] Adrian Casajus, Ricardo Graciani, Stuart Paterson, Andrei Tsaregorodtsev, and the Lhcb Dirac Team. DIRAC pilot framework and the DIRAC workload management system. *Journal of Physics: Conference Series*, 219(6):062049, apr 2010.

[92] A Tsaregorodtsev, V Garonne, J Closier, M Frank, C Gaspar, E van Herwijnen, F Loverre, S Ponce, R. Graciani Diaz, D Galli, U Marconi, V Vagnoni, N Brook, A Buckley, K Harrison, M Schmelling, U Egede, A Bogdanchikov, I Korolko, A Washbrook, J P Palacios, S Klous, J J Saborido, A Khan, A Pickford, A Soroko, V Romanovski, G N Patrick, G Kuznetsov, and M Gandelman. DIRAC - Distributed Infrastructure with Remote Agent Control. *CERN Document Server*, (0306060):8p, Jun 2003.

[93] K De, A Klimentov, T Maeno, P Nilsson, D Oleynik, S Panitkin, A Petrosyan, J Schovancova, A Vaniachine, and T Wenaus. The future of PanDA in ATLAS distributed computing. *Journal of Physics: Conference Series*, 664(6):062035, dec 2015.

[94] De, K., Klimentov, A., Maeno, T., Mashinistov, R., Nilsson, P., Oleynik, D., Panitkin, S., Ryabinkin, E., and Wenaus, T. Accelerating science impact through big data workflow management and supercomputing. *EPJ Web of Conferences*, 108:01003, 2016.

[95] Michael A. Salim, Thomas D. Uram, J. Taylor Childers, Prasanna Balaprakash, Venkatram Vishwanath, and Michael E. Papka. Balsam: Automated scheduling and execution of dynamic, data-intensive hpc workflows, 2019.

[96] Vivekanandan Balasubramanian. *A programming model and execution system for adaptive ensemble applications on high performance computing systems*. Rutgers University, 2019.

[97] SimPy. `https://simpy.readthedocs.io/en/latest/`, 2020.

[98] RADICAL Campaign Manager. `https://github.com/radical-project/campaign_manager`, 2020.

[99] Yuichi Inadomi, Tapasya Patki, Koji Inoue, Mutsumi Aoyagi, Barry Rountree, Martin Schulz, David Lowenthal, Yasutaka Wada, Keiichiro Fukazawa, Masatsugu Ueda, Masaaki Kondo, and Ikuo Miyoshi. Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, volume 15-20-November-2015. IEEE Computer Society, nov 2015.

[100] Kevin A. Brown, Nikhil Jain, Satoshi Matsuoka, Martin Schulz, and Abhinav Bhatele. Interference between i/o and mpi traffic on fat-tree networks. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, New York, NY, USA, 2018. Association for Computing Machinery.

[101] Yan Yao, Jian Cao, and Shiyou Qian. Throughput-guarantee resource provisioning for streaming analytical workflows in the cloud. *Computer Supported Cooperative Work and Social Computing*, pages 213–227, 2019.

[102] Liang Bao, Chase Wu, Xiaoxuan Bu, Nana Ren, and Mengqing Shen. Performance Modeling and Workflow Scheduling of Microservice-based Applications in Clouds. *IEEE Transactions on Parallel and Distributed Systems*, pages 1–1, 2019.

[103] Christopher D. Carothers, Jeremy S. Meredith, Mark P. Blanco, Jeffrey S. Vetter, Misbah Mubarak, Justin LaPre, and Shirley Moore. Durango: Scalable synthetic workload generation for extreme-scale application performance modeling and simulation. In *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '17, pages 97–108, New York, NY, USA, 2017. ACM.

[104] Ketan Maheshwari, Eun-Sung Jung, Jiayuan Meng, Vitali Morozov, Venkatram Vishwanath, and Rajkumar Kettimuthu. Workflow performance improvement using model-based scheduling over multiple clusters and clouds. *Future Gener. Comput. Syst.*, 54(C):206–218, January 2016.

[105] Carl Witt, Marc Bux, Wladislaw Gusew, and Ulf Leser. Predictive performance modeling for distributed batch processing using black box monitoring and machine learning. *Information Systems*, 82:33–52, may 2019.

[106] Sarunya Pumma, Wu-chun Feng, Phond Phunchongharn, Sylvain Chapeland, and Tiranee Achalakul. A runtime estimation framework for ALICE. *Future Generation Computer Systems*, 72:65–77, jul 2017.

[107] Marek Wieczorek. Scheduling of scientific workflows in the ASKALON grid environment. *SIGMOD Record*, 34(3):56–62, sep 2005.

[108] Louis-Claude Canon, Emmanuel Jeannot, Rizos Sakellariou, and Wei Zheng. *Comparative Evaluation Of The Robustness Of DAG Scheduling Heuristics*, pages 73–84. Springer US, Boston, MA, 2008.

[109] Fangpeng Dong and Selim G Akl. Scheduling algorithms for grid computing: State of the art and open problems. Technical report, Technical report, 2006.

[110] Wikipedia contributors. List scheduling — Wikipedia, the free encyclopedia, 2018. [Online; accessed 15-April-2020].

[111] T. Fahringer, R. Prodan, Rubing Duan, F. Nerieri, S. Podlipnig, Jun Qin, M. Siddiqui, Hong-Linh Truong, A. Villazon, and M. Wieczorek. Askalon: a grid application development and computing environment. In *The 6th IEEE/ACM International Workshop on Grid Computing, 2005.*, pages 10 pp.–, Nov 2005.

[112] F. Dong and S. G. Akl. Pfas: A resource-performance-fluctuation-aware workflow scheduling algorithm for grid computing. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–9, March 2007.

[113] K. R. Shetti, S. A. Fahmy, and T. Bretschneider. Optimization of the heft algorithm for a cpu-gpu environment. In *2013 International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 212–218, Dec 2013.

[114] A. Y. Zomaya and Yee-Hwei Teh. Observations on using genetic algorithms for dynamic load-balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(9):899–911, 2001.

[115] I. M. Oliver, D. J. Smith, and J. R. C. Holland. A study of permutation crossover operators on the traveling salesman problem. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*, page 224–230, USA, 1987. L. Erlbaum Associates Inc.

[116] PSC Bridges. `https://www.psc.edu/bridges/user-guide/system-configuration`, 2020.

[117] SDSC Comet. `https://www.sdsc.edu/support/user_guides/comet.html`, 2020.

[118] Top 500 Supercomputers List. `https://www.top500.org/lists/top500/list/2020/06/`, 2020.

[119] Linping Wu, Xiaowen Xu, Yong Wei, and Xu Liu. A survey about quantitative measurement of performance variability in high performance computers. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10561 LNCS, pages 76–86. Springer Verlag, 2017.

[120] David Skinner and William Kramer. Understanding the causes of performance variability in HPC workloads. In *Proceedings of the 2005 IEEE International Symposium on Workload Characterization, IISWC-2005*, volume 2005, pages 137–149, 2005.

[121] Alok Singh, Shweta Purawat, Arvind Rao, and Ilkay Altintas. Modular performance prediction for scientific workflows using machine learning. *Future Generation Computer Systems*, 114:1 – 14, 2021.