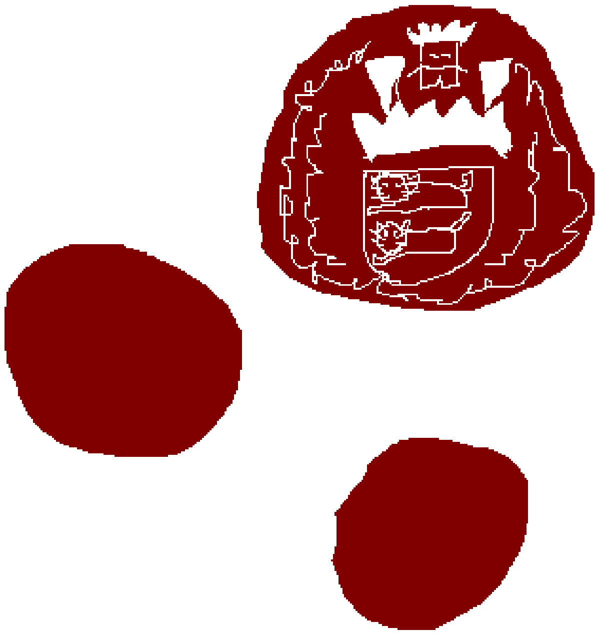


PhD thesis

Rasmus Munk

Grid of Clouds

A model for how resources can be shared amongst organisations

Advisors: Brian Vinter, Kenneth Skovhede & James Avery

Handed in: February 28, 2021.

Acknowledgements

I would like to thank my supervisors, Brian Vinter, Kenneth Skovhede, and James Avery, for being excellent supervisors. Not only for the guidance they have given me throughout the PhD, but also for the motivation they have given me when things were not as much fun. A special thanks also has to be given to the entire eScience group at the Niels Bohr Institute, where I have enjoyed myself a lot. Shout out to my fellow students: David Marchant, Carl Johnsen, Alberte Thegler. In addition to the ERDA/HPC Team, namely Jonas Bardino, Martin Rehr, Hans Henrik Happe, Rune Kildetoft, and Jawad Ahmed. A thanks of course is also given to former colleagues including Mads R. B. Kristensen, Mads Ohm Larsen, Klaus Birkelund, and anyone else I might have forgotten.

Thank you to Toni Cortes and his former team for given me the opportunity to work at the Barcelona Supercomputing Center. In turn, I would also like to thank Xnovo Technology for welcoming me and David Marchant at your work and for our collaboration on cloud technologies here in the last year. A thank you to the team at MAX IV for our collaboration on providing interactive data analysis and high throughput storage, in addition to always giving us a nice welcome in Sweden.

I also give a very special thank you to Karl Nymann and Veronika Skotting for all the fun and great times we have had together. With the numerous office games, actual game times, late nights of fun, sleepovers, road trips, ITU Python Study groups and everything else. Your company have really supported me and has helped with keeping me motivated and in good spirit this last year and a half. I have really appreciate this time that we have had together! Following this, I would also like to thank the most recent addition to the flat, Malika Luzie Dahl for all the great company, meals, alcohol, and board games we have enjoyed together.

Likewise a thank you is given to my old dormitory friends, even though we are all spread out over Denmark at this point, I really appreciate that we still keep in touch. With the yearly summer trip and the obligatory christmas lunch. Thanks to Simon Vinsten, Line Luff Bennetsen, Lars Krag Nygaard, Anne-Sofie Fredslund, Mathias Ladefoged, Ngoc Christensen, Søren Riis Malling. An extra thank you is given to Marlene Kongsted and Casper Nyhuus, for being so kind as to visit me during my time in Barcelona, I really appreciated that!

Also a thank you to Kevin Chalmers, for whom if hadn't reached out on my behalf, this PhD properly never would have happened. A thank you also to the MUMMERING project and all its organisers and participants for both all the work they have put into making this project possible.

Finally I would like to thank my family back in Jutland for always being there for me, and being so understanding when I haven't had the time to travel back home.

This project is developed as part of the MUMMERING[1][2] research project with the aim of providing researchers with a generic tool of the management and processing of scientific sets of data. This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 765604.

[1]*MU*ltiscale, *M*ultimodal, and *M*ultidimensional imaging for *E*nginee*RING* (MUMMERING)

Abstract

When seeking new scientific insights, critical aspects of any scientific inquiry is the acquisition, analysis, and storage of scientific evidence. In today's environment, with the vast growth in data generation, computational platforms have become a key pillar in making such insights feasible. Additionally, with the underlying increase in compute power not coming from faster processors, but rather from the increase in the number of cores within a particular compute platform, the task of speeding up existing applications is no longer achieved by waiting for the next processor. As a complement to this, additional compute capacity is also emerging in specialized hardware platforms, that also leverage the possibility of increasing the number of computational cores within a single device. Nevertheless, this additional capacity on its own does not make up for the projected amount of data that has to be processed in the future. Because of this, the organizing and employment of the existing resources, and how and where data is stored becomes ever important.

Historically, the organising of computational resources have changed as per the technological developments of their time. Both the Grid and Cloud notions seek to organize resources such that they can be effectively used by multiple users.

In this thesis, I explore how the modern computational infrastructures can be used and deployed to provide scientists with the tools they require to achieve new scientific insights at the different stages of inquiry, be they at the acquisition, analysis or storage stage. Specifically, I present how data can be ubiquitously stored and retrieve as part of data acquisition, or within a particular analysis with the introduced MiG Utils library. Furthermore, I present a design solution for how high throughput data generation from large scientific instruments could be optimized by applying in-situ computational kernels to datastream via The HIgh Throughput Storage System.

Beyond investigating the storage and extraction of scientific data for subsequent analysis, another important aspect is also how the organised computational platforms allow the scientist to devise and conduct their analysis. At UCPH, I present the introduced interactive programming services Data Analysis Gateway and MPI Oriented Development and Investigation platforms. These platforms enable scientists and students to define and schedule processing tasks in a web based environment. Furthermore, I present how cloud computational resources can be utilized in a scientific or commercial settings. This is achieved with the introduction of the Cloud Orchestrator, which enables both orchestration, utilization, and job scheduling at a set of supported cloud providers. In the thesis, I will present how this was utilized to enable both a novel JupyterHub Spawner and a traditional neutron ray-tracing simulator to utilize cloud resources to achieve their function. Finally, I present how the Cloud Orchestrator can be used to establish a Grid of Clouds, that is how cloud resources can be shared across a set of providers by establishing a GridCloud extension on top of the Cloud Orchestrator. In doing so, such an extension should be able to establish a mechanism to share resources amongst both public and private clouds, in a loosely coupled network via a decentralized broker. Thereby establishing a Grid of Clouds model, that can be utilized in collaborative environments such as the MUMMERING ITN.

Resumé

I jagten på ny videnskablig indsigt, indgår kritiske aspekter af enhver videnskabelig undersøgelse erhvervelse, analysering og opbevaring af data. I dagens miljø, med den enorme vækst i datagenerering er effektive beregningsplatforme blevet en nøgle komponent til at finde ny indsigt muligt. Derudover følger, at den fremtidige stigning i beregningsevne ikke vil kommer fra hurtigere processorer. I stedet vil stigningen snarere kommer fra et øget antal kerner inden for en bestemt beregningsplatform, hvilket betyder at man ikke længere bare kan forvente at opnå en øget beregningsevne, ved at vente på den næste processor.

Som et supplement til dette, vil der også opstå yderligere beregningskapacitet i specialiserede hardwareplatforme, der også udnytter muligheden for at øge antallet af beregningskerner inden for en enkelt enhed. Dog selv med disse fremskridt, vil denne ekstra kapacitet ikke alene være tilstrækkelig til at behandle den forventede mængde data der bliver genereret i fremtiden. På grund af dette, bliver organisering og anvendelsen af de eksisterende ressourcer, og hvordan og hvor data lagres, stadig vigtigere.

Historisk har organiseringen af beregningsressourcer ændret sig i henhold til den underliggende teknologiske udvikling. Både med Grid- og Cloud-modellerne søges der at organisere ressourcer, så de effektivt kan bruges på tværs af organisatinoner og brugere.

I denne afhandling undersøger jeg, hvordan de moderne beregningsinfrastrukturer kan bruges og implementeres til at give forskere de værktøjer, de har brug for til at opnå ny videnskabelig indsigt på de forskellige stadier af en videnskablig undersøgelse. Det er hvad enten det er i anskaffelses-, analyse- eller opbevaringsstadiet af processen. Specifikt præsenterer jeg, hvordan data kan lagres og hentes som en del af dataopsamling eller inden for en bestemt analyse med det introducerede MiG Utils-bibliotek. Desuden præsenterer jeg en designløsning til, hvordan generering af data med høj kapacitet fra store videnskabelige instrumenter kunne optimeres ved at anvende in-situ beregningskerner på data strømme via det forslåede The HIgh Throughput Storage System.

Ud over at undersøge lagring og udvinding af videnskabelige data til efterfølgende analyse, er et andet vigtigt aspekt også, hvordan de organiserede beregningsplatforme tillader forskeren at udtænke og udføre deres analyse. På KU præsenterer jeg de introducerede interaktive programmeringstjenester Data Analysis Gateway og MPI Oriented Development and Investigation. Disse platforme gør det muligt for forskere og studerende at definere og planlægge behandlingsopgaver i et webbaseret miljø. Desuden præsenterer jeg, hvordan cloud-beregningsressourcer kan udnyttes i videnskabelige eller kommercielle omgivelser. Dette opnås med introduktionen af Cloud Orchestrator biblioteket, som muliggør både orkestrering, udnyttelse og jobplanlægning hos et sæt understøttede cloud udbydere. I afhandlingen vil jeg præsentere, hvordan dette blev brugt til at muliggøre både en ny JupyterHub Spawner og en traditionel neutronstråle sporing simulator til at udnytte cloud ressourcer til at opnå deres funktion. Endelig præsenterer jeg, hvordan Cloud Orchestrator muligt kan bruges til at etablere et net af clouds, det er, hvordan cloud ressourcer kan deles på tværs af et sæt udbydere og organisationer ved at etablere em GridCloud-udvidelse oven på Cloud Orchestrator. Dermed skulle en sådan udvidelse være i stand til, at etablere en mekanisme til at dele ressourcer mellem både offentlige og private clouds i et løst koblet netværk, via en decentral mægler. Til at etablere dette, forslås der en model for at oprette et Grid of Clouds netværk, der kan bruges til at dele resourcer på tværs af organisationer i tidsbegrænsede projekter såsom MUMMERING ITNet.

Contents

Chapter 1

Introduction

In today's science environment, the importance of computational platforms cannot be overstated. From the use of computational simulations, modelling and analysis to pursue new scientific discoveries in fields such as astronomy [128], climate [56] and physics [156], the need for computational capacity is not going away.

In addition to wanting to either increase the resolution or speed of the current computational tasks, the challenge for future system developments and scientific projects is also the continual growth in data generation. For instance, International Data Corporation (IDC) predicts that the worlds data collection will grow from 45 Zettabytes (ZB) in 2019 to 175ZB in 2025 [152]. This is also reflected in the scientific sphere, where both educational and scientific institutions will experience tremendous growth, both in the amount of data that is going to be generated and subsequently processed. For instance, facilities such as the European Synchrotron Radiation Facility (ESRF) [42] and the European XFEL (EuXFEL) [44] expect that their output will increase from 8 petabyte (PB) in 2019, to 50 PB by 2023. To meet this challenge, the existing computational capacity has to increase as well.

However, at the individual machine level, the rate at which processors execute operations have not seen exponential improvement since the mid 2000s, topping out at a clock rate of about 4 GHz. Instead, the development of the multi-core architecture with parallel running processors at the same executing rate has been the main method in which additional compute capacity has been achieved up until recently. This in addition to utilizing accelerator technologies such as GPUs that employ a massive amount of parallel processors at a lower rate compared to high-end CPUs. However, from this point on, the coming increases in computational power is mainly provided by the utilization of heterogeneous system architectures with multiple computational types including, System on a Chip (SOC)s and Accelerated Processing Units (APU)s such as CPUs, GPUs, DSPs, and FPGAs. Gains in performance from SOCs and APUs will mainly come from an increasing number of individual cores within each device. Enabling an increasing amount of calculations that can be executed in parallel within a given time frame. Furthermore, novel architecture developments such as co-locating compute capacity with data storage thereby minimizing the distance in which data has to travel before being computed is another prospect that could have great benefit at the individual machine level.

Looking beyond the single device, another strategy that has been employed since the introduction of computer networks, is to employ multiple machines to increase the number of operations that can be conducted in parallel at a given time. Since the early days of computational machines, the organising of these connected resources have changed as different technologies and architectures have emerged. This includes distributed architectures such as Grid federated networks from the early 2000s which was inspired by the early electricity grids, where distributed resources across organisations become a shared commodity. This was followed by developments in the early 2010s, with the introduction of the Cloud computing concept, which is heavily inspired by the Grid notion. In contrast though, Cloud computing puts its emphasises on providing resources that are dynamically established through on-demand elastic scaling of compute instances. This was both made possible and cost effective through technologies such as virtualisation.

To establish the initial Grids [50], middleware frameworks and tools such as the Globus Toolkit [53] have been developed as a do-it-yourself distributed toolkit, and have historically been the de-facto standard for establishing Grids [53]. However, Grids have only seen limited success outside of the scientific arena, in that no commercial introduction produced the same wide scale success as established scientific Grids. For instance, the European Grid Infrastructure (EGI) currently hosts 1E6 computing cores and +740 PB of disk [40] and are used in many Pan-European scientific projects. Part of the reason for this, was due to commercial vendors seeing little usage in allowing external entities to utilize their infrastructure. That, in addition to the limitation that Cloud Computing eventually solved, including on-demand and scaling computational capacity via technologies such as virtualisation. Such innovations allowed for the dynamic expansion of isolated compute environments across underused resources.

Given this, the Globus Toolkit for instance has been deprecated since 2018 [53]. Furthermore, the ability for scientific organisation and institutions to establish Grids is also limited by this development. However, this does not mean that the Grid architecture cannot be of use today. In particular the sharing of computational resources is still of great usage to institutions with limited budgets and commercial constraints like public institutions. In addition, the ability utilize a combination of institutional and commercial resources could have a potentially great benefit.

The challenge however, is how such a Grid should be established. This has typically involved the deployment and configuration of a complex service stack on an existing computing cluster. Followed by the continuous support and maintenance required to keep the Grid operational. This often implies that vast resources in terms of personal hours have to be dedicated to keep the Grid operational. Examples of such includes Advanced Resource Connector (ARC) [41], The Globus Toolkit [53], Distributed Interactive Engineering Toolbox (DIET) [34] and the Minimum Intrusion Grid (MiG) [19] to name a few.

The ability to define distributed Grid resources at a DIY level for ephemeral or burst infrastructures is limited in these frameworks. The reason being that they are developed to define and deploy continuous services and infrastructures. This is coupled with the introduced complexity that is necessary to monitor and ensure a continuous infrastructure. Meaning that they often require teams of people to define and provide Grid resources. In addition, they often do not allow the expansion of the classic institutional grid resources to be expanded commercial cloud resources, there at some development in this area, such as DIET supporting EC2 instances [34] but this capability is still at the prototyping level. Furthermore, the ability to migrate running computational tasks between Grid providers is not something that is enabled by any known Grid middleware system.

Therefore, the aim of this work, is to enable scientist, and potentially educational and scientific institutions to establish and share a resource pool of compute capabilities across organisational boundaries with inspiration from both the previous and recent Grid initiatives. Thereby enabling the establishment of a Grid of multiple clouds that can be exploited for both small and large scale scientific inquiries. This would be especially beneficial to temporary projects and collaborations between institutions that do not have the scale that deem it worthy to apply for huge grid resources such as the EGI. A high-level overview of such a Grid of Clouds between four participating organisations can be seen in Figure 1.1.

In addition, a challenge that often arises when computational tasks are to be scheduled on remote resources, is the staging and extraction of data. Since the early days, this has been solved by either manually copying the data back and forth via some network, giving access to a shared data repository such as network or parallel filesystems, object storage solutions, or enabling access to other forms of external data repositories. Beyond the manual approach, middleware frameworks have employed multiple strategies to provide this capability. A common approach is to decouple the definitions of inputs from the code itself and let them be provided by the middleware framework as part of the job description that is being scheduled. ARC for instance allows for the staging through input and output files definitions that specifies which files are to be copied to and from the job. However, this approach does come with some limitations, most importantly, each identical job description has to define its own inputs. Meaning that if multiple Grids were to define the same job, they would each have to specify the same set of data staging definitions.

As a response to this, I will present a prototype through the mig_utils library on how data staging could be coupled as part of an implementation, to make the defined computation ubiquitous, in that wherever it is executed the data would be pulled and utilized for that particular computation. This approach by its nature in general does impose a lower speed at which jobs can be executed, because of the increased distance and

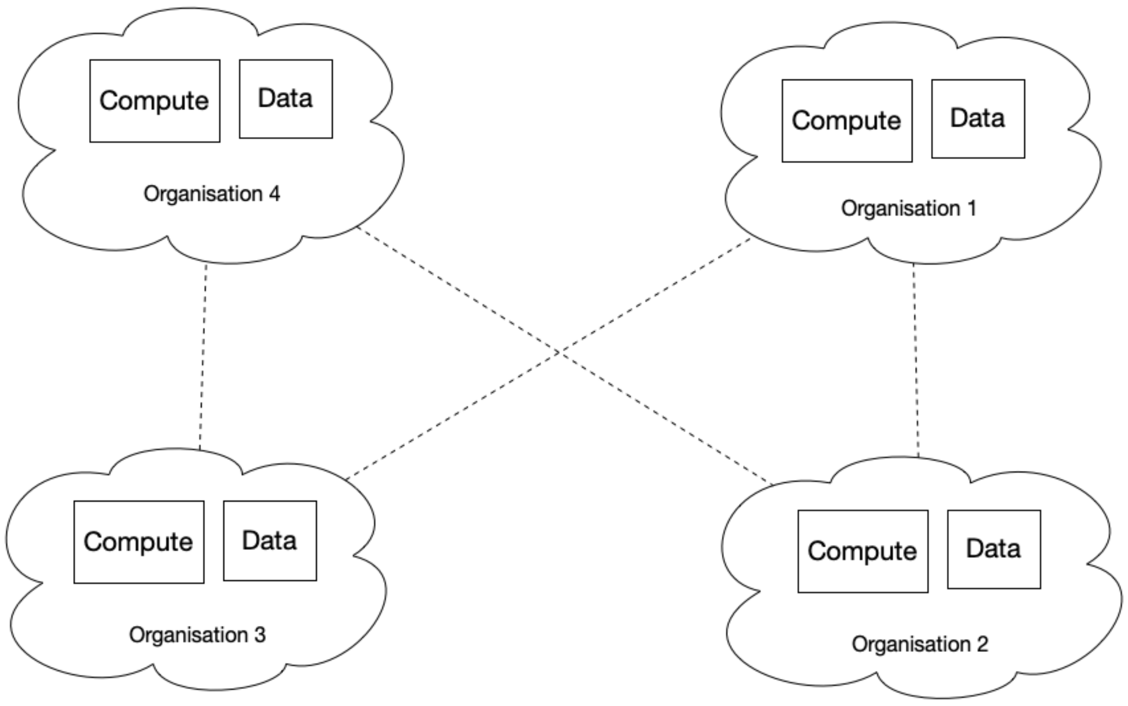


Figure 1.1: A high-level overview of an established Grid of Clouds between four participating organisations. Compute = Cloud Computational Resources, Data = The organisations data storage service.

potentially lower bandwidth data has to travel before reaching where it was requested. Therefore another model was also developed to target the high throughput requirement that organisations also have, namely the design of a HIgh throughput Storage System (HISS), that is designed to act as a big funnel for staging datasets from vast producers such as a scientific instrument temporarily.

1.1 Publications

This thesis is based on a number of publications that have been published as work progressed during the PhD. In this Section a brief overview of these are given. They are listed in the order in which they were published.
Rasmus Munk, Brian Vinter. MUMMERING Platform Idea's & Ubiquitous Data Analysis
In Communicating Process Architectures 2017 & 2018 WoTUG-39 & WoTUG-40

Rasmus Munk, Artur Barczyk, Zdenek Matej, Brian Vinter. From instrument to publication: A First Attempt at an Integrated Cloud for X-ray Facilities
At 6th Cloud Storage Services for File Synchronization and Sharing Conference

David Marchant, Rasmus Munk, Elise O. Brenne, Brian Vinter. Managing Event Oriented Workflows
XLOOP 2020 at Super Computing 2020

Rasmus Munk, David Marchant, Brian Vinter. Cloud enabling educational platforms with corc
At 8th Workshop on Cloud Technologies in Education (CTE 2020)

David Marchant, Rasmus Munk, Brian Vinter, Further Developments in Event Oriented, Emergent Workflows*
Euro-Par 2021, Workshop

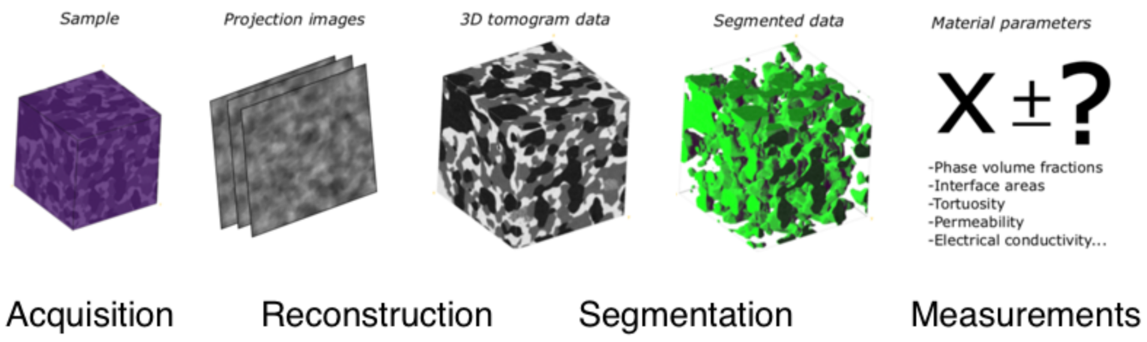


Figure 1.2: An overview of the tomography workflow [22].

1.2 Thesis Outline

This thesis is organized as follows: Chapter 2 will provide the background knowledge necessary to understand the thesis. Chapter 3 describes the research and results in developing the MiG Utils data sharing library and the HISS model. Chapter 4 describes the research and results in developing a set of interactive computational platforms to enable access to computational resources. Chapter 5 describes the research and results in developing the Cloud Orchestrator library, and cloud enabling novel and existing applications including the MultipleSpawner and the McStas simulator. Furthermore, presenting how a set of institutional Clouds could be connected in a Grid model for sharing resources. Chapter 6 presents the future work to be done from here and Chapter 7 contains the final conclusions.

1.3 Thesis Statement

At the outset of the PhD, there was no clear problem statement for what had to be researched. Instead the scientific thesis had to be discovered and developed as part of the PhD project. This was done in the context of the MUMMERING research project. MUMMERING was established with the aim of creating research tools that could empower scientists with access to the wealth of 3D imaging modalities for applications in materials engineering across the tomography workflow. Furthermore, it was established to train 15 Early Stage Researcher (ESR) students [2], across 25 participating organisations spanning nine different European countries. Each of the ESR's were employed in a specific domain of the tomography pipeline, which is illustrated in Figure 1.2. What is common across each of these tomography domains, is that each of them is complicated by the challenge of handling and processing vast quantities of 3D datasets in order to conduct scientific experiments. Furthermore, the challenge of using and producing such datasets is only made worse, when they are utilized across distributed collaborations like MUMMERING. In particular, when large datasets are physically spread out across multiple entities across large geographical distances, the sharing, access, and utilisation of such datasets becomes a challenge that appears in both the runtime cost because of distance latency and the complexity of locating and utilising them.

1.3.1 Facilitating the distribution large datasets

In this thesis, the problem of providing and facilitating access to large scale distributed datasets was investigated. Specifically, the problem of enabling usage of said datasets, without the user having to be aware of the physical location of said datasets during their execution. This was achieved by coupling the logical location and destination of the input and output datasets for an analysis, with the analysis itself. This coupling was made possible in MUMMERING with the introduction of the MiG Utils library. The library enables the novel ubiquitous execution of experiments in Python by enabling a programming model were the input and output data are coupled with the analysis. This is in contrast to typical data frameworks such as Hadoop, where the staging of input and output datasets are decoupled from the analysis. In addition, another contribution made in this thesis is the introduction of the HISS model. This model shows how in-situ processing of datasets could occur to speedup the post processing of generated datasets at scientific instruments such as MAX IV or EuXFEL by applying pre-compiled computational kernels on FPGA devices. A subsequent successful implementation of said model, could potentially save executing time by conducting pre-filtering computations before the generated experimental data is stored.

1.3.2 Providing computational platforms

Another problem that became evident early on in MUMMERING, and is likely to arises in similar multi organisation projects, is providing computational resources. Specifically, a challenge that often arises is the ability to utilize the organisations computational resources, without the continued involvement of computer experts. In this area, this thesis presents how the introduced the Data Analysis Gateway (DAG) and Message Passing Interface (MPI) Oriented Development and Investigation (MODI) services provides computational platform capabilities to institutions such as UCPH or MAX IV. By introducing these services, the institutions have been able to provide computational resources in an interactive manner at their own initiative. The novelty in these services, lies in the complementary libraries that enable their integration into the institutions platforms. For instance, the jhub-authenticator, enables the DAG service to be integrated with a data management solution by introducing the ability to provide mounting credentials to the JupyterHub Authenticator. Furthermore, the ldap-hooks library novelty is that it makes it possible for the JupyterHub service to conduct LDAP operations on an existing SLURM compute cluster, prior to spawning the user session, thereby ensuring the automatic integration with said cluster. Further details on this is presented in Chapter 4. These contributions, enables the ability to provide out of the box interactive data processing platforms that can utilize existing institutional compute platforms, such as a SLURM compute cluster. Their usage has already been found at UCPH, where they have been used teach courses in Computing, Applied Statistics, and Physical Oceanography.

1.3.3 Dynamic scheduling of workflows

Beyond the ability to perform computations in an interactive manner. Another problem that became apparent in MUMMERING, was that in multi organisational scientific collaborations, is how computational tasks should be organised and scheduled. Specifically, in scientific areas such as tomography, analysis or simulations is made up of several processing steps. To organise such computation, researchers often use scientific workflows with their added tools to organise these [81]. However, traditionally scientific workflows have applied a static approach, that defines individual workflow as a directed acyclic graph that fully describes the necessary steps required to complete the workflow. This approach has been useful in some scientific areas, but there are specific areas, such as tomography, which could benefit from more dynamic workflows where changes can occur during the execution. In this thesis, the contribution to the dynamic workflow system, Managing Event Oriented Workflows (MEOW) will be presented. This system was designed and implemented in collaboration with the main contributor David Marchant. This collaboration produced an implementation of MEOW with subsequent example use cases of a dynamic workflow, that could not be produced so easily in a traditional static workflow system. The full extend of the contribution to this work is presented in Chapter 4

1.3.4 Establishing and using grid resources

Beyond the ability to share and exploit datasets across organisations in collaborative environments. The ability to schedule experiments on computational resources is similarly a prerequisite. In this thesis, a contribution was made towards how an individual scientist can acquire and share computational resources across organisations. Specifically with the introduction of the corc toolbox/framework, an individual scientists is able to acquire and utilize computational resources and data related tasks at an individual level across its supported providers. Currently, this includes the Oracle Cloud Infrastructure (OCI) and in part Amazon Web Services (AWS). Thereby enabling a scientists to utilize multiple cloud providers to conduct their research. The novelty lies in the simplicity in which the tool exposes the ability to orchestrate resources, without having to require a team of computer experts to support the infrastructure. This does not come without certain drawbacks, including that the corc does not guarantee nor does it monitor that the defined infrastructure is consistent over time. Another instance of corc's usage, is shown with the introduction of the MultipleSpawner for JupyterHub and the enabling of cloud capabilities to the McStas simulator. In Chapter 5, the MultipleSpawner contribution is presented. This novel Spawner enables the JupyterHub service to orchestrate user requested resources at an external cloud provider, which to my knowledge has not been possible before with any of the current JupyterHub Spawners. As a first case example, it archives this by orchestrating a designated Virtual Machine at the OCI, before scheduling a Jupyter Notebook instance via the CloudSSHSpawner. In addition, by cloud enabling the McStas simulator with corc by introducing few altercations to the McWeb platform, it highlights its ability to empower research companies like Xnovo Technologies with greater computing scale. This is in contrast to what would otherwise require a substantial procurement and computer expertise to provide a similar computational infrastructure.

1.3.5 Summary

Overall, the work in this thesis has been informed by the challenges faced in providing both data and computational resources in a multi-organisational context such as the MUMMERING project. Thereby aiming to provide both access and accessibility to the underlying resources necessary, in order for them to conduct their scientific inquiries without having to become computer experts. This resulted in the contribution of the MiG Utils library, the HISS model, the DAG/MODI interactive services with their complementary libraries, the dynamic MEOW system, and the corc framework/toolbox that cloud enabled the McStas and MultipleSpawner applications.

Chapter 2

Providing Computational Resources

As highlighted in Chapter 1, the way in which computational resources are organised is dictated by the available technology and the challenges of their time. This Chapter will give a brief overview over how distributed architectures like Grids and Clouds have come about and how different approaches and developments have provided computational resources in a Grid or Cloud architecture. This Chapter will not however be a complete survey of these developments, but to give a highlight of some of the most popular and common approaches in providing computational resources, including their benefits and drawbacks. To begin with a brief outline will be given for how the notion of the Grid has developed, followed by its common components, how in practice they have been established, developments that have provided Grid infrastructure, and how Cloud Computing has build upon this context.

2.1 Distributed Architectures

There are numerous architectural approaches in which organisational compute resources can be organised both physically and logically. For starters at the physical level, distributed resources can be deployed on opposite sides of the planet, while being interconnected through the internet. Alternatively, they could also be installed in the same physical location, being interconnected via dedicated high-bandwidth low-latency links.

The choice amongst these depends on numerous factors. Including the goal of the particular problem that the resources are to be employed against. For instance, low latency is very important when dealing with problems that require a tight coupling between the individual processes. This is typically the case in talkative applications such as large scale climate and astronomy simulations, where there is a as a large amount of messages are exchanged between processes to conduct the simulation. This makes latency a very important performance factor in these applications, that is the delay in time it takes to transmit a message to another process. In such a scenario, a classic HPC oriented architecture with fully connected nodes would be appropriate to utilize due to its emphasis on being physically close with high bandwidth and low latency interconnects between nodes. Pivoting to loosely coupled applications the importance of latency flips. Here a set of distributed compute resources spread out over great physical distances is not as much of a worry, thereby enabling the use of a greater compute capacity than is available at a particular institution.

Similarly, in terms of logical organisation, the computational resources can be organised in multiple ways, including a graph, tree, layered, and flat architecture or possibly in a combination of multiple logical structure. As with the physical organisation, the choice depends on the problem that is to be solved or service that is to be provided, that is what requirements are there in terms of availability and scalability to the provided system.

Given this development from the early days of homogeneous compute infrastructures, to today's varied landscape of multiple compute providers and platforms, the complexity in both defining, utilizing and maintaining these resources is a substantial challenge even for computer science and infrastructure professionals. This typically leads to a cascade of complicated distributed network architectures and software stacks that have to be both provided and maintained to be operational. The areas involved cover both the interconnect and management of the resources themselves in addition to the required software dependencies enabling their use, while attempting to expose a minimal amount of complexity to the user. Establishing such internal infrastructures, has typically been dedicated to an operations team that ensures that the installation, configuration and maintenance is taken care of. However, after this has been established at an individual institution, the point of concern stops at the organisational boundary. Within the institution, a typical infrastructure stack is composed of three layers as shown in Figure 2.1, namely, the Hardware, Infrastructure and Service Layer. The Hardware Layer is concerned with the architectural design and installation of physical hardware to serve a particular need. For instance, commodity servers, such as off-the-shelf desktop computers, typically do not have the same high requirement for low latency interconnects as HPC oriented servers. Therefore, their installation will differ, typically in a simple design that ensure availability compared to performance if they are to host a critical service for the institution.

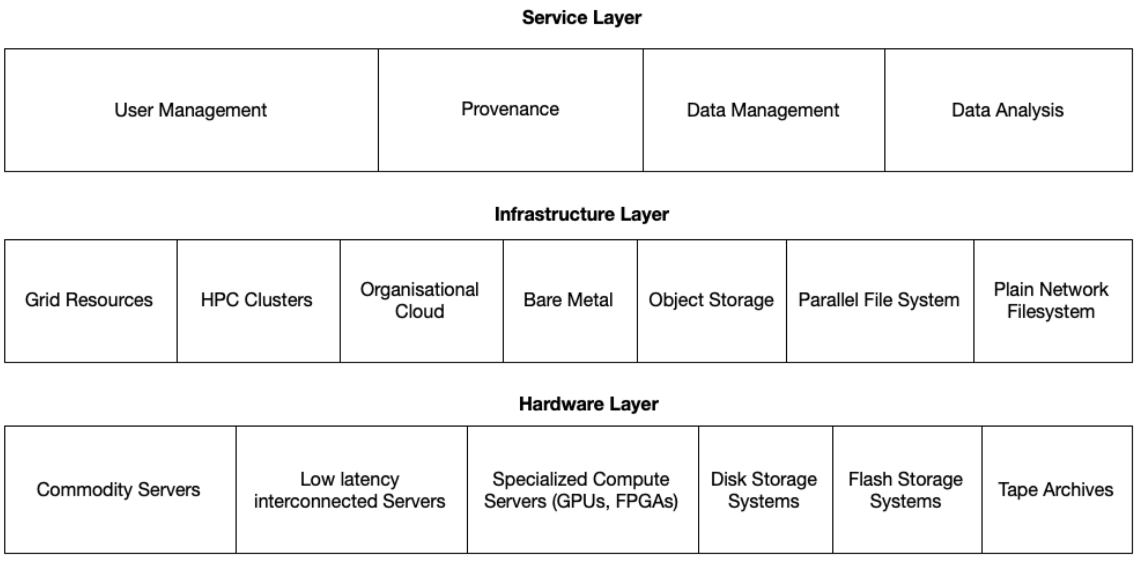


Figure 2.1: An example of an scientific compute stack.

This is where the federated Grids comes into play, because they act as a gateway for access to internal organisational resources. However, with the existing federated Grids, the internal compute nodes are directly registered and managed by the external Grid.

2.2 The Grid model

Each era of computational developments has defined how the innovations of the day should be organised to face the challenges of its time. This has been the case from the early days of independent mainframes in the 1950s with their shared mainframe environments, to the establishment of network connected research computers during the 60s and 70s [58]. This was followed by the emergence of interconnected powerful desktop devices in the 80s and 90s, to the present day of the reemergence of cluster computational such as High Performance Computing (HPC) centers across organisations [88].

With the combined development of high-speed interconnected networks, capable desktop compute capabilities, and the requirements of applications in the 90s, the emergence of new computing infrastructure designs started to develop in the scientific community. One of such designs was the establishment of compute clusters based on commodity hardware from powerful desktop computers across scientific, federal and educational organisations [88]. To effectively leverage these acquired resources, a push was made to make these resources accessible across such distributed institutions via federated networks.

The idea was to allow access to computing resources on demand through flexible, secure and coordinated sharing of resources amongst a dynamic set of individuals, institutions and resources [50] [49]. A so called Grid of on-demand computational resources, analogous to how the electrical power grid operates, where individuals are given access to a shared commodity, with all the benefits and drawbacks that follows from a large scale federated network of resources. In addition to mere access to greater computational capacity through a federated network, the Grid notion also implied other capabilities. This included how such resources should be coordinated in a non-centralized manner across a federated network of multiple organisational domains that are controlled by different administrators. This is often provided via so called Virtual Organisations (VO), that spans the entire Grid, enabling members and users with the essential ability to share resources, data, etc, across different Grid members. VOs enable such sharing by establishing a logical grouping that spans every member of the Grid. Users that are then part of such VOs are typically able to associate shared organisational resources as part of a particular VO. Furthermore, the Grid model imposes that resources and entities should be interconnected “using standard, general-purpose protocols and interfaces“ [49], in addition to delivering some quality of service so that resource usage can be coordinated to meet defined standards of for instance throughput or response time, to serve multiple users at a given point.

This led to the establishment of numerous Grid infrastructures, for example the European Grid Infrastructure (EGI), the Open Science Grid (OSC) and The Laser Interferometer Gravitational Wave Observatory (LIGO) Data Grid [75]. A timeline of some of the important developments and Grid establishments can be seen in Figure 2.2. However, at the end of the 2000s, the Grid model had only achieved success in specific scientific fields such as high energy physics [73], climate [4], and gas methane research [45] to assist in solving large scale problems in pan-national environments.

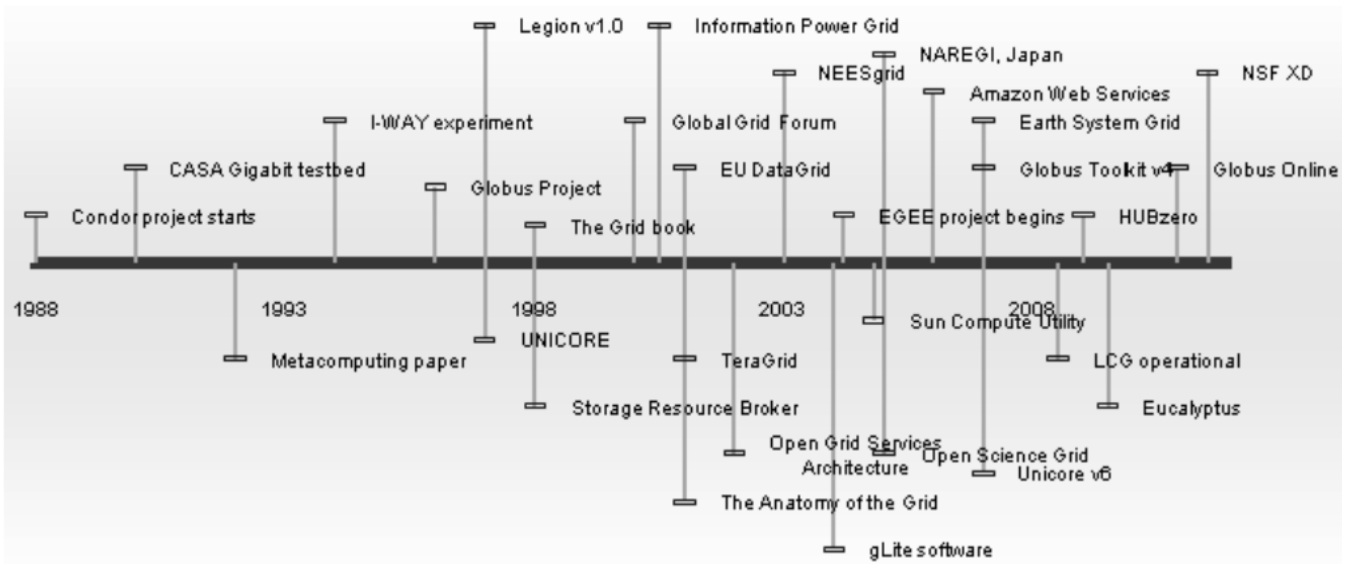


Figure 2.2: 30 representative events in Grid development [49].

However, these successes did however not carry over into the commercial space. Part of the issue, was that the developing Grids did not have of enough utilization to establish an sufficient scale to be economical. Instead since the middle of the 2000s till the present, Grid oriented platforms in the commercial space have been superseded or complemented by introduction of Cloud computing platforms [49]. Multiple contributory factors are the reason for this; foremost is that they had the required scale and demand to establish a cost effective architecture that had been the crux of the Grid initiatives. The early frontrunner in the public arena has been the AWS [8] [182], which due to business necessity validated the establishment of a large scale web service. As part of this development, the requirements for AWS produced duplicate service functionalities across the organisation, which validated the creation of common services and APIs. Furthermore, the fluctuating annual demand of compute resources meant that periods would occur with under-usage, or in contrast huge spikes. The most classic example of which are big online shopping days in the United States like Black Friday. As a result of this AWS established a set of common services that could be used by outside users in an on-demand manner. For instance Simple Queue Service (SQS) for inter-service messaging, Simple Storage Service (S3) for object oriented storage, and Elastic Compute Cloud (EC2) as a Infrastructure-as-a-Service (IaaS) platform. These were some of the first releases of such services [21] [182]. Enabling worldwide access to them in 2004 and 2006 with the addition of similar such services from other public Cloud Computing providers [51].

However, in the world of large scale scientific research, Cloud Computing capabilities such as Infrastructure-as-a-Service (IaaS) have not taken off in a similar manner as in the commercial space. As presented by [111], the challenges have historically included lower raw performance, especially when scaling compute-intensive workloads with a high degree of inter-process communication as found by [197] where certain HPC applications suffered a slowdown of 50 times that of a dedicated HPC cluster due to the high degree of small inter-process messages between compute nodes. Instead, HPC applications can benefit from access to cloud allocated resources if they exhibit other characteristics. For instance, decent performance can be achieved with embarrassingly parallel and bandwidth-limited applications because they do not utilize an extensive communication pattern that inherently relies on low-latency connections. Furthermore, what is often disregarded when comparing application performance between infrastructures is as defined by [78] the turnaround time from queuing an application until it is being scheduled and completed. The authors show that in public cloud environments such as EC2 produce better turnaround times than that of a classic HPC clusters at Lawrence Livermore National Laboratory (LLNL). This is in part due to oversubscribing of HPC resources, which could lead to a total turnaround of more than a factor of four when compared to a comparable cloud instance.

Taking this to the present date where computational landscape is composed of a heterogeneous set of private and public infrastructures, compute platforms, and services to expose a given compute capacity. This includes classic institutional computational clusters or clouds; national or pan-national HPC centers; distributed Grid resources; and public cloud computing services. Adding to this mixture of compute capacities the internals also offer heterogeneous configurations from bare-bone CPU based servers to APU enabled systems including GPUs, FPGAs and TPUs. For instance, at scientific instruments such as MAX IV [85], and EuXFEL where computational resources are provided through classic internally shared institutional compute clusters. Namely the Lund University Compute Center [77] and the DESY compute centers [33]. This is also the case at educational institutions like The University of Copenhagen (UCPH), that has its own HPC center associated within the Science faculty. These computing resources are typically exposed internally either via traditional remote shell access to a login node, or through some web-based portal to submit batch oriented jobs to the internal system. In contrast to these local institutional systems, facilities such as the Large Hadron Collider (LHC) [24] have compute requirements that simple cant be met by an individual institution. Since its inception in the early 2000s, the LHC has instead been a main motivational component for established the worlds largest Computing Grid [25] [73]. This network of compute capacity is made up of the European Grid Infrastructure (EGI) and the Open Science Grid (OSG), spanning 41 countries and 170 compute centers in a hierarchical structure serving 8000 physicists in accessing and analysing data in near real-time.

2.2.1 Grid Components

When defining what constitutes a Grid, many different opinions merge. Particularly what components and behaviour are required before a federated network of resources can be referred to as a Grid. The following statement by Ian Foster makes a good attempt in making such a definition. “A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities.” [48]. This definition was later extended in [50] to include, that Grids are concerned with “Coordinated resources sharing and problem solving in dynamic, multi-institutional virtual organisations“. That is, a Grid provides a shared infrastructure between organisations to exploit cross institutional resources to perform large scale problem solving in virtual groupings. To provide this capability, a Grid is typically made up of the following components:

- Authentication and Authorization
- Resource and Service registration and discovery
- Scheduling of computational jobs across resources

- Data management to and from the Grid

2.2.1.1 Authentication and Authorization

Authentication is the term used to describe the process in which an entity is validated to be who they claim to be. This process typically involves providing some secret information that the service can verify is as expected. In the Grid setting, this becomes additionally complex and important, because trust has to be established between various decentralized resources across different organisations with different security models and procedures. Authorization on the other hand is concerned with the level of privileges or access a properly authenticated entity is allowed to obtain. A classic example of this is the definition of User Groups that dictates what rights a particular user has. For instance, an authenticated user's rights could be divided into either administrative, basic, or guest capabilities.

Much effort has already been devoted to provide proper authentication and authorization mechanisms to Grid infrastructures. This includes authentication through digital certificates, tickets, usernames, passwords and beyond [64]. Examples of such solutions includes Globus Grid Security Infrastructure (GSI), Kerberos, and Single Sign-On (SSO) architectures such as Shibboleth and OpenID. In addition, Public Key Infrastructure (PKI) were used in GSI as an authentication scheme to establish trust between federated organisations through the validation of X.509 certificates by Certificate Authorities.

In this thesis, the scope will not include an extensive review of potential authentication and authorization techniques, but instead the proposed Grid framework relies on the authentication and authorization techniques established by the existing Clouds and Infrastructures the proposed framework will utilize.

2.2.1.2 Providing Grid Resources

When providing Grid resource, middleware frameworks have generally relied on the local administrative body of a particular Grid member to be responsible for provisioning, configuring, and maintaining their resources. The middleware frameworks were instead responsible for providing the interconnect fabric between the components as mentioned in Section 2.2.1.

When establishing Grid resources, the task has traditionally involved the following: First, validating that the Grid resource can be created, meaning that a particular compute resource can join the Grid, while adhering to the organisations policies, that is whether doing so violates policies such as security, data protection, or privacy. Second, exploring the technical requirements for establishing a compute resource; this can typically involve the provisioning and configuration of a prototype node in a restricted environment to validate that the provided specifications actually match up with reality. If possible, this can also include the registration of the prototype resource to test the capability and subsequent usage of the resource. For instance, it could be used to carry out some computational task, such as running a small scale ocean or neutron ray-trace simulation, to explore how a potential experimental workflow would be conducted. Given that this process produces a satisfactory result, the organisation will typically have to map out which and how resources they would want to provide to the Grid should it be established. This is often a daunting task in terms of formal procedures and required paperwork that has to be completed before the resource can actually be established as a Grid resources. The task of establishing the resource, for instance a Virtual Machine (VM) for computational work, is often known these days as resource orchestration by the providers administrative team.

2.2.1.3 Orchestration

Orchestration is about providing an automated method to configure, manage and coordinate computer systems, such as establishing compute resources through a bundled and easy workflow [151]. Through orchestration, an organisation or individual such as a university or a researcher is able to establish a complex infrastructure via a well defined workflow. As I presented in [105], each task is carried out to establish the required component to provide the defined infrastructure. An example of a simplified orchestration workflow can be seen in Figure 2.3. To orchestrate the requested resource, a valid operating system Image, a Shape that defines the specification of

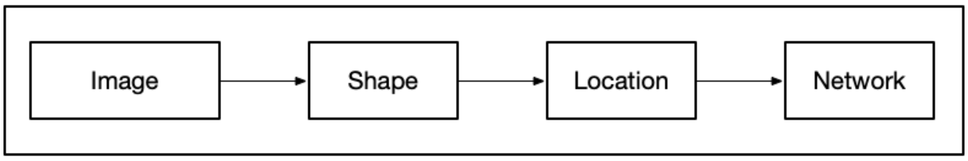


Figure 2.3: Workflow for orchestrating a compute node. [105].

the compute resource, a Location of where the resource should be orchestrated and how it should be connected to the providers Network [105].

In the context of a federated network like a Grid, the orchestration would ideally involve the automated provisioning of the computational resource, the configuration of said resource, and ensure that the resource is correctly reachable via a configured network infrastructure [105]. In terms of providing orchestration, multiple tools have been developed that enables this. This thesis will not present a complete survey over every orchestration tool, but instead will present the themes and common approaches that have been employed to automate the tasks necessary to establish a complex computing infrastructure such as establishing a set of Grid resources at an institution like a Research Instrument. Examples of such automation tools includes Ansible [12], Puppet [146], and TerraForm [166]. As presented in [105], each of these tools do not provide full orchestration capability on their own, instead they each have been born out of the specialty they are designed to perform.

2.2.1.4 Configuration

Primarily, Ansible is a configuration tool, that helps automating the configuration of existing systems after they have been provisioned, that is the compute resource have been created and is running with the operating system ready to be configured. In addition, Ansible is a so-called agent-less application, meaning that it does not require the nodes that it wants to configure to have an active agent running. Instead Ansible only requires to be installed and executed on a single node which can be a designated configuration server. In terms of design model, this is also known as the push model, where the server pushes the data it wants to apply to the client in question. This requires that the designated server that on which Ansible is executed can reach the specified node via the connected network to reach the exposed port.

To configure a particular node, Ansible only requires to be told which connection and authentication method it should use to gain access to the node in question. The chosen connection method then subsequently dictates which authentication methods are available. By default, Ansible uses OpenSSH [13] [123] to connect to a particular node. Subsequently, public key authentication via SSH keys are the default authentication mechanism used when utilizing an OpenSSH connection [13]. When Ansible configures a node, it expects to be told which instructions it should execute on the target node. This set of instructions are defined in a file called a Playbook [14]. A Playbook is at its core a YAML Ain't Markup Language (YAML) [196] structured file, as the name indicates, is a Markup Language formatted file. An example of this structure can be seen in Listing 2.1, where a user with the attributes **name** and **mail** is defined. As this demonstrates, YAML is arguably easy for humans to read and write, due it its simple structure and limited amount of required formatting. It is based on the same principal as the Python programming language, in that indentation is used to define same level structures.

```yaml
user:
  name: Rasmus Munk
  mail: rasmus.munk@nbi.ku.dk
```

Listing 2.1: YAML syntax example.

Ansible, as indicated, is not a tool that is able to provide full orchestration capabilities, it does not establish a primary method for instantiating resources, but instead focuses on configuring existing resources that have already been established. Nevertheless, due to its simple architecture, it is a good candidate for introducing configuration management with minimal intrusion and footprint on the target orchestration and resources. The reason is that it only requires a single port to access a node, and that it simply applies a set of instructions on said resources makes it quite attractive in a federated model where a uniform architecture and state management is not feasible.

2.2.1.5 Puppet

Puppet, like Ansible, is also mainly a tool for automating administrative tasks such as configuration of computer systems. In contrast to Ansible, Puppet is based on a master-agent architecture. This means that every node that is to be configured has to be installed with the client application before it can be configured by Puppet. Another substantial difference is that Puppet is designed to maintain the specified configuration state of the node instead of the simple apply and leave alone model that Ansible applies. Maintaining a particular configuration on every node, is made possible via the added complexity of using the agent based model. Having an agent daemon on a resource, imposes its own set of additional complexities and requirements. For instance, the health, correctness, and version of each agent daemon has to be maintained in order for the configuration management to work as expected. Furthermore, since the agents have to retrieve the configuration from the master node, the nodes are required to 'pull' any configuration that they have to apply. This has the added caveat that the server has to be reachable by the agents on the designated port, potentially requiring this port to be opened across multiple organisational firewalls. Alternatively, the configuration traffic could be forwarded to and from the nodes via other secure networking mechanisms, such as Secure Shell (SSH) tunneling, or Virtual Private Networks (VPN)s. However, this only introduces more complexity onto the underlying architecture, thereby increasing the risk of failure and the required maintenance required to keep it operational.

In terms of orchestration, Puppet allows for the definition of tasks and plans that can be applied on-demand [147]. Nevertheless, Puppet's orchestration capabilities are limited to controlling how configuration updates are applied, or how node data should be changed. [148]. Therefore, Puppet, like Ansible is designed to control the state of resources after they have been established. Similarly, this makes it infeasible to use for orchestrational tasks beyond ensuring the state of established resources.

2.2.1.6 TerraForm

TerraForm is a infrastructure deployment tool developed in Go by Hashicorp [166]. It is quite different to both Ansible and Puppet. Foremost, TerraForm is not about configuring existing resources, but is about building infrastructures in a cloud setting. To create an infrastructure, TerraForm expects a set of .tf files that describes in declarative form how the infrastructure should look like after being provisioned. Upon provisioning, the descriptions in the .tf files are then translated into state files in a JavaScript Object Notation (JSON) structure. These state files are TerraForm's way of maintaining records that describe the desired state of the specified infrastructure. This has the implication, that in a multi-user environment, the state files have to be kept in a shared location. Meaning that due to the sharing constraint, consistency of the file immediately becomes an issue that has to be handled. The reason is that if multiple processes tries to update the same state file at a given time, the outcome will become an inconsistent state file that may not reflect every change that should have been applied. For example, when a series of processes are tasked with applying a set of changes to the infrastructure, the last process to commit its changes will determine the end-result of the state file. To mitigate this issue, TerraForm provides the capability to manage the state file through a set of remote backends that ensure that any change is applied without causing consistency problems. Specifically, TerraForm utilizes locks to guarantee that only one process can change the state file at a given time, thereby removing the possibility of multiple changes being applied at the same instant. Examples of such remote backends includes, Amazon S3, Azure Storage, Google Cloud Storage, and Terraform Pro. In terms of security, TerraForm does not provide a mechanism to encrypt the state file before it is being stored at the remote site, but relies on the encryption capabilities of the selected remote backend. Implicit trust is thereby given to the responsible storage provider, including that they wont access your data without permission, nor that they will fail to protect it from potential breaches.

TerraForm supports infrastructure orchestration and maintenance across different cloud providers. This includes all the major public compute clouds such as AWS, Azure, Google Cloud Platform, Oracle Cloud Infrastructure and beyond [57]. When constructing a particular defined infrastructure, TerraForm defines a core workflow of multiple stages. The two most important of these stages are the Plan and Apply stage. The Plan stage is where the user-defined .tf files are read and the target infrastructure is derived from the combined information. The result of this stage is a plan that is ready to be applied to the defined cloud providers. Subsequently, when the Apply stage is carried out, TerraForm will provision the specified infrastructure.

In addition, TerraForm supports orchestration across cloud provider infrastructures and is not limited to a single cloud during the Apply Phase. This capability, makes it suitable to utilize when the need is to provision and maintain a complex compute infrastructure across multiple cloud environments. Overall, TerraForm is a tool targeted towards teams of infrastructure operators, system administrator, and DevOps teams that are tasked with maintaining large and complex cloud infrastructures. As found by [72], the complexity of the tool introduces a steep learning curve for its usage. Also, when transitioning between different cloud providers, TerraForm's infrastructure descriptions are not portable. Which means that TerraForm might be used for orchestrating resources at different cloud providers, but is not easily capable of transitioning an existing infrastructure to a different cloud provider without substantial effort. This lack of portability imposes a substantial barrier when transitioning infrastructures across clouds. Additionally, TerraForm is only concerned with provisioning resources, it is not built to support job scheduling and data staging.

2.2.1.7 Configuration and Provisioning

In terms of configuring resources both Ansible and Puppet can provide this. In addition, since this is not an exhaustive list, there are multiple other projects that could be considered to provide configuration management. However, these two do highlight some of the overall traits that most configuration management tools provide. That is the agent-less vs. server-agent model, push vs. pull data flow, maintaining or applying configurations.

Putting it all together, from the two presented options, Ansible has the simple architecture model in that it does not require an agent to be running the node in question. Furthermore, the ability to push changes simplifies the owning entities responsibilities to ensure that configurations can be applied successfully. Specifically, the owner of the nodes that are to be configured is as a default, responsible for ensuring that they are reachable by the configuration server on the designated port. In contrast, by using the pull technology, the owner of the server can be forced to expose ports in multiple firewalls to allow a range of client nodes to access the designated port to retrieve their configuration. Similarly, it is simple to apply configurations in contrast to maintaining them, because there is no continual monitoring that has to happen, in order for the configuration manager to ensure that the current configuration is correct. Therefore, it follows that in only applying configuration, as Ansible does, the owner of the nodes cannot be sure that the individual configured nodes are kept in a correct state. This concern is bigger, when the resource in question is utilized by multiple users, or multiple purposes that introduces configuration changes.

The choice of configuration manager therefore, is all up to circumstance, and the particular situation in which it has to be applied. If the target infrastructure is tightly coupled and well controlled system by a single owner or organisation. The more leeway there is to use more complex configuration managers such as Puppet, because it ensures greater control and assurance that the participating nodes are in a correctly configured state while the infrastructure is running.

However, in terms of providing a Grid of Clouds, the expectation is that multiple resources will be distributed across multiple organisations and security boundaries. This implies that it is infeasible to expect that the entire resource Grid is tightly integrated, connected, or homogeneous. Therefore it can be hard to expect that a configuration manager like Puppet is suitable to be utilizes in such a decoupled architecture. Instead, simpler and looser coupled configurations seems more feasible in that it does not require specialized firewall rules or the maintenance of numeros agent clients on Grid resources. This however, could validate further investigation because a deep analysis of every tool was not conducted during this work.

2.2.1.8 Discovery of Resources

In a distributed architecture, the discovery of resources is crucial in order to commit the requested action to the correct resource or service. Multiple different approaches have been applied to provide this capability. In a Grid structure the challenge is greater in that resources are typically organised in distinct administrative domains, where there is no central control of underlying infrastructure layers such as networking and external firewalls. Instead, since resources are under the control of the members that provides them, which typically configures them in a non-public network of their organisation the resources often have to be accessed indirectly. Be it through a public proxy node that relays authenticated and authorized resource oriented messages to the internal node, or via more elaborate architectures such as SSH tunnels or VPNs

What is common beyond this is, that when a job is submitted by a user to the Grid, the Grid middleware, typically is responsible for discovering and selecting an appropriate resource candidate where the job could be executed. In Globus Toolkit for example, the combination of the Gatekeeper service and the Grid Resource Allocation Management (GRAM) protocol provides the backbone functionality for ensuring both authentication and authorization in addition to resource discovery and utilization from a particular client's perspective. The selection of an appropriate resource is based on the set of job requirements supplied with the client's submission. Upon selecting a valid resource, the Gatekeeper is subsequently also responsible for instantiating a Job Manager that then schedules the particular process. How this happens is defined by the particular selection of job scheduler, be it a simple process fork or a submission to a Local Resource Manager (LRM) such as Condor or PBS [71] [47] [46]. A limitation of this design, is that the Globus Toolkit relies on the client submitting the job directly to the resource in which it executes, thereby claiming all the responsibility of interacting with the resource to handle the job lifecycle and the staging of input and output data from the job [20].

An upside down approach of this is employed by the MiG [132] [20], where the resources themselves are responsible for discovering whether the Grid node has any jobs that are available for them to carry out. By flipping this responsibility between the central Grid server and the resources, it ensures a simplified model, where the Grid oriented logic of the resources is provided entirely by the central server and no resource discovery is required. This model limits the amount of cross domain server administration there has to occur for maintaining a Grid resource. The reason being that the provider is responsible for defining and registering their resources through the central server, hereafter these configurations are utilized to configure the resource itself as long as it is accessible from the Grid itself. The result is that the Grid maintainers only are required a minimal involvement in maintaining external providers resources. This difference also has an effect on the client side when it comes to job management, because the client is interacting with the central Grid server instead of a resource directly, the responsibility the client is alleviated from having to manage the life cycle of the job directly as was the case in Globus [46].

ARC [41], However does not include the approach to resource mapping and discovery. The management and discovery of ARC resources are instead provided through two services namely the Local Information Services (LIS), the Index Services (IS) and relies on the OpenLDAP protocol for exchanging and storing any resource related information. In term of data structure the ISs and LISs entries are organised in a topological structure of a multi-level tree where the LISs constitute the bottom leaves of the tree. The LIS are responsible for describing and characterizing resources be they storage, computing, or job oriented. This information is collected on the resource itself by the service and can be subsequently extracted. The ISs are used to maintain a dynamic list of the resources themselves. Entries are added to this by a registrant which can either an LIS or another IS. This basically makes the Index Service a list of contact URLs of ISs and LISs [41]. The end result is a bottom-up approach, where the LIS on a particular resource ensure that it is registered in an IS by sending it its endpoint URL of the LIS service. This process occurs at a periodical interval to update the LDAP entries as according to the current state of the registered resources.

Each of these approaches have certain benefits and drawbacks. In the native Globus Toolkit, the Gatekeeper has to actively find a node on which the users job can be submitted, meaning that the it potentially has to go through N-1 nodes before finding an appropriate resource. This hurts the scalability of the GRAM oriented architecture, in that with the growth of the number of nodes in the Grid, the worst case discovery equally grows linearly in runtime complexity. In contrast the MiG architecture relies on the resources themselves having to query for an appropriate job, flipping the discovery mechanism on its head, however it does mean that the MiG server is central to the successful execution of jobs, since no jobs can be discovered without it, leaving it as a potential single point of failure.

Overall, resource discovery and subsequent registered state is always a set of potentially outdated information due to the distributed nature of Grid and Grid-like systems. This means that whatever the model of the Grid, resource discovery and subsequent state tracking is critical to have a functioning Grid of reliable and reachable resources.

2.2.2 Jobs

As highlighted in the inception of the initial Grids, before a Grid can be useful to the participating community, the users must to be able to utilize the shared computational resources. A popular approach to facilitate this, is to enable the user to define jobs. A job can be defined as a description of how a particular computational task should be executed, in addition to what is required of the underlying system before the execution can take place. This description can be wide ranging in terms of the information that it specifies. Covering attributes such as the type of job, the amount of required hardware capabilities and additional runtime dependencies for a job to successfully execute. The following itemization is an non-exhaustive list of common attributes that are defined in a Grid framework's job description:

- **CPU:** The amount of CPUs (cores) required;
- **Memory:** The amount of Memory required, typically defined in MegaBytes or GigaBytes;
- **Input:** The required data that needs to be staged in the Grid resource before the job can execute successfully;
- **Output:** The generated output from the job, typically in the form of files or streams;
- **RuntimeEnvironment:** Defines an environment that indicates the presence of a certain set of operating system attributes, can include details such as available compiler, file system, dedicated hardware, accelerators and beyond;
- **CPU Time:** The amount of time required to finish the job.

The format of the particular job description depends on the framework used. Globus utilizes the Resource Specification Language (RSL), while ARC uses an extended version of it [41]. Both the basic and the extended version use the following structure to define a particular job:

- Resource requirements are a set of constraints that the underlying resource must provide in order for the job to be successfully execute. They include attributes such as the machine type, number of nodes, amount of memory, max CPU time
- Job configuration on the other hand are the attributes that specifies how the job is to be carried out on the valid resource. They include attributes such as directory, executable, arguments for the target execution, and environment

2.2.3 Job Scheduling

Job scheduling in a Grid setting is the task of matching a particular job description to a matching resource that fulfill the requirement of that job. Upon matching a job to a resource, the subsequent task of the scheduler is to translate the job description into a structure that can be executed on the resource in question. To translate the description, a common abstraction layer is often employed to provide a uniform interface for transforming the description into a format that can be scheduled and processed on a supported system. In terms of the particular scheduling type, Grids have often scheduled jobs in a batch oriented manner.

2.2.3.1 Batch Oriented

Batch oriented scheduling, is the act of executing a job in a non-interactive manner, this means that jobs are queue and executed when a system is ready to process them. A major reason for this is that since a Grid is made up of shared resources amongst its participants, a job's requirements cannot be expected to be fulfilled at the time of job submission. Therefore, upon submission from a user, jobs are put into a queue where they will wait until a matching resource becomes available.

Existing and historical Grid frameworks have often provided batch oriented scheduling by utilizing traditional Local Resource Management System (LRMS). The LRMS in question is then responsible for receiving, handling, queuing, submitting, and retrieving jobs received from the Grid. For instance, Grid frameworks such as Arc or Globus utilize and support many LRMS back-ends [114], including HTCondor [60], SLURM [92], Portable Batch System (PBS) [6].

2.2.4 Job Strategy

Job strategy is deciding in which order queued jobs should be executed. The following itemization lists some common scheduling, strategies used for scheduling user jobs:

- **FirstFit:** The job will be scheduled on the first resource that fits
- **BestFit:** Find the best resource that matches the jobs requirements
- **FairFit:** Like BestFit but with an increasing priority as time passes, this is to avoid the starvation tendency of BestFit
- **First in First Out:** The First queued job is also the first to be scheduled
- **Last In First Out:** The most recent job to be queued is also the first to be scheduled.

2.2.5 Job Dependencies

Job dependencies are the set of requirements that a resource runtime environment have to fulfill in order for the job to be processed successfully. For instance, if a particular job executes an application that is tasked with generating a model that can classify images, how that classification application is developed defines its set of dependencies. Specifically, if the application was developed in Python3.8 with the use of packages such as Tensorflow and Keras those are then part of the application dependencies. For such an application to be successfully executed, the runtime environment would have to provide an operating system that supports Python3.8, in addition to valid versions of Tensorflow, and Keras.

How job dependencies have been provided has evolved over the years. From early on with native compilation and installation from the source code, to the leveraging of package managers to install the compiled binaries and required support libraries over the internet. At the present, the use of package managers is still one of the default ways that dependencies are provided on computational resources. Therefore, a simple approach to provide its dependencies could be to make the user specify a list of the required software that their intended application requires. This list could then be installed as a script before the job is executed. The MiG [19] for instance handles dependencies through the definition of a set of user and Grid provided runtime environments. A particular runtime environment then establishes a kind of contract between the users and the resources. In MiG, both the user and the resource have to uphold the contract before a job can be successfully scheduled. Specifically, users are responsible for creating runtime environments, subsequently it is up to the owner of the resource to advertise the environments that the resource provide.

2.2.6 Resource environments

A problem that naturally follows from the requirement of having to support a wide range of software dependencies across multiple resources is the maintenance of the resources while being a Grid participant. With changing dependency requirements, and an increasing user pool, the resource nodes are bound to be the target of a wide range of applications. This variety increases the likelihood of package, version, and dependency conflicts, which increases the amount of administration that a resource node imposes to keep it operational. To mitigate the possibility of environmental conflicts, there are several existing models that can be utilized. The most popular includes the bundling of job dependencies in virtual environments, VM images, or container images. All have the objective in common that they attempt to segment the job requirements into their own self contained compartments, thereby removing the possibility of conflicting with other dependencies beyond their own compartment.

2.2.6.1 Virtual Environments

A virtual environment in the context of an operating system is a method for isolating a particular language runtime with the required packages in a separated compartment. When such a compartment is loaded, it ensures that requests to a package or library is first searched for within the compartment. If this search did not produce a successful result, the search is widen to check whether the global system provides it instead. This provides a hierarchical search structure, where the lowest virtual environment can be swapped at runtime. Examples of virtual environment managers includes, debootstrap [32] for Debian, virtualenv [149] for Python 2 and 3, Gem for Ruby and Anaconda [11] for multiple language dependencies including Python, R, Go and beyond. However, virtual environments are not part of the basic operating system installation on major Linux distributions such as CentOS or Debian. Instead these have to be selected, installed and configured after a resource has been provisioned.

2.2.6.2 Virtual Machines

Desktop Virtualization is the concept of detaching the physical hardware from the operating system [183]. This provides the ability to have simultaneous operating systems utilizing the same underlying hardware resources, also known as time-sharing, thereby enabling a more efficient usage of the hardware. A beneficial side-effect of being able to run multiple operating systems on a single computational node, is the ability to segment and isolate users into their respective system. This segmentation introduces a security barrier between the running operating systems and their processes [17]. In terms of Grids, VMs are useful in that they allow the dynamic management of resources. This capability is enabled through the hypervisor, which provides on-demand scaling and provisioning of VMs, given there is hardware capabilities to support it. This ability is available in both on premise cloud frameworks such as OpenStack or through every major public cloud provider, including AWS, OCI, or Azure. In addition, what is equally important when thinking about providing complex resource environments, is the hypervisor and cloud provider abilities to create VM templates. A template can be seen as a copy of an existing VM, which includes the installed OS, its applications and the VMs configuration.

The usefulness of VMs to provide dynamic resources in Grid and cloud solutions is well established across multiple projects and providers. Including EGI FedCloud [43], INDIGO-DataCloud [62], OpenNebula [122], AWS [8], OCI, AWS and beyond.

2.2.6.3 Containers

Containers are a form of operating system virtualization that enables the isolation and segmentation of environments within a single operating system [186]. This is in contrast to the classic notion of VMs, that provides isolation and segmentation via virtualization of fully-fledged operating systems. This has the implication that containers usually imposes less resource requirements because they rely on a single operating system, which resources are shared amongst them. Specifically, containers usually utilizes normal operating system call interfaces and do not rely on emulation as virtualized VMs [186]. In Unix-like systems the capability is provided via the underlying operating system kernel, an overview of this can be seen in Figure 2.4. Examples of container implementations for Unix-like systems includes LXC [185], Docker [37], Singularity [162], and Podman [133]. A more extensive overview of implementations can be seen at [186].

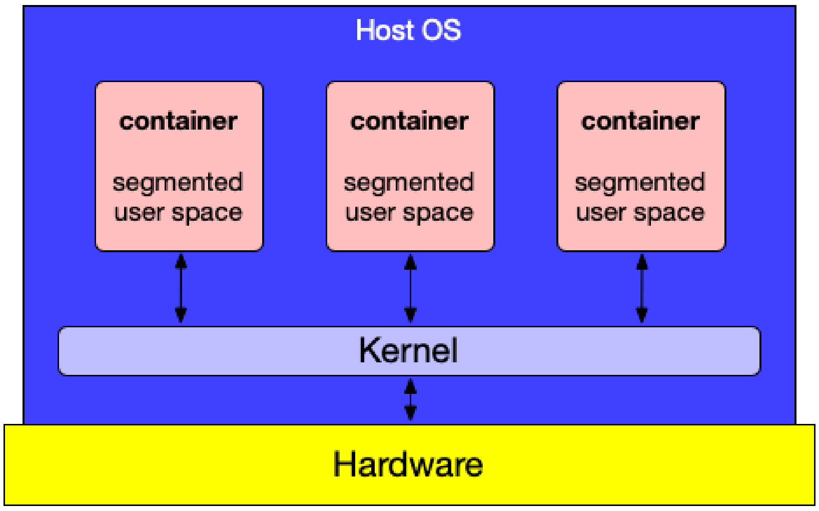


Figure 2.4: Unix-like operating system virtualisation.

In terms of standardisation, the Open Container Initiative (OCI) project [169] is the structure tasked with creating a standard API for containers. The OCI aims to enable cross container implementations to be compatible with each other. To mark whether a container implementation adheres to the OCI standard, the project is developing a certification program that validates the adherence to the standard and thereby ensures compatibility. As of writing, the OCI has seen adopting by projects such as Kubernetes and Apache Mesos [170]. An additional important defacto standard that has been developed in the container world, is how containers are built to provide pre-configured environments. Specifically, the Docker image format [170] is the go-to standard for how a container should be built. This has the implication that other implementations often has the ability to either convert existing Docker images or is able to make them directly. [38] [161]. Aligning with the OCI compatible implementations or future adopters will give the broadest support for supporting container runtime environments across a wide range of heterogeneous resources. For instance, Buildah [108] is a project that specializes in building OCI container compatible images. By utilizing Buildah, the compatibility and maintenance of current specified image builds down the road should be manageable at a minimum cost.

However, containers as a runtime environment, does not come without certain costs and potential pitfalls. Firstly, the runtime environment has to be build before it can be utilized, this build can range from seconds to hours depending on how complex the target environment is. Secondly, managing the complexity of such environment is a compromise between minimizing the libraries, binaries and applications in the build, while fulfilling the wishes and requirement of its prospective users. The result of this, is that the time it takes to build an image can range from seconds to hours. In addition, due to the changing nature of dependencies, the container image has to be maintained if it is to be used on a long term basis. For maintaining images to be feasible at scale, such as with hundreds of images utilized across federated organisations, centralized management of these images is a major task. Therefore, the development and maintenance of such container based runtime environments, is likely to be most scaleable and manageable for federated organisations if the responsibility for managing these is delegated to either the end user themselves, or to as low as level as possible within the organisation.

2.2.7 Workflows

Workflows are used to describe the order in which a set of jobs should be scheduled and executed. Originating in the world of business applications, classic workflows have enabled the automation of tasks that have had a predictable order and requirement. Classic example of this includes workflows such as ordering tickets, generating annual financial records, automating complex administrative computing tasks including automated backup, provisioning of compute resources and many more. Scientific workflow however, are typically of a different nature. These workflows tend to consist of longer running tasks that are not functionality oriented but are typically compute oriented towards discerning information from collected datasets. For instance, a classic scientific workflow could involve the analysis of 3D imaging tomography datasets that have been generated at a scientific instrument. Specifically, the workflow could be tasked with investigating voids in composite materials [76]. What such a workflow example also implies, is that because the task is often exploratory in nature, the time it will take is often not deterministic at the onset.

Numerous frameworks exists to define workflows, including traditional ones such as Apache Airflow [167], Kepler [7], Globus Workflow [47], Tarvena [26], and more recent ones such as Dask [154]. These frameworks typically employ a top down or data flow model to organise the specific tasks to be executed as part of a singular workflow. In terms of how the frameworks structures the tasks to be executed, they typically employ a list or a directed acyclic graph (DAG) of tasks to be executed. A simple example of a scientific workflow can be seen in Figure 2.5.

Preprocess
Analysis
Postprocess

Figure 2.5: Example Workflow.

2.2.7.1 Data Management

A key component in providing a computational infrastructure to users, is to ensure that the users are able to manage their data when utilizing the infrastructure. In scientific computing, there are two extreme scenarios when it comes to handling data in a computational infrastructure. One extreme is the handling of large datasets, that consists of multi gigabyte or terabyte data files, another is the management of millions of kilobyte sized files. This reality does not change when computational resources are spread out across multiple organisations. Either of the two scenarios, requires the underlying ability to both upload and download data to and from the infrastructure, what distinguishes them is how they each are best optimized to gain the best performance. However, this is not something that is going to be covered extensible in this thesis, but the basic aspect of how the data should be provided to the infrastructure in the first place. Specifically in the Grid of Clouds context, the ability to both, stage and extract data to and from the utilized resources from multiple locations is key. There have been multiple approaches in how this can be achieved. One approach is to define the specific datasets via the computational framework itself. MiG [20], Globus Toolkit [53], and ARC [41] for instance allows for this by enabling the user to set the required input during the creation of a job. These inputs are then staged to the executing resource before the job is being executed.

In the MiG, the data files are then required to be staged in the responsible user's filesystem on the central Grid server. This means that the data first has to be uploaded to the data storage of the MiG, before it can be utilized for a job. In addition, once uploaded, the executing resource has to transfer the data to itself, via a transfer protocol such as SSH File Transfer Protocol (SFTP) [187]. Upon a subsequent job submission, the job file then describes where the resource can expect to find the input data relative to its own filesystem environment.

ARC handles the staging of inputs and outputs via the Data Transfer Request (DTR) framework, as described by [113] it is a three-layer architecture. An overview of the architecture can be seen in Figure 2.6. As this figure indicates, the DTR is responsible for the staging the data requirements of jobs, managing transfers, and transfer resources such as bandwidth and cached connections [115]. The DTR is managed by the ARC Compute

Element, which the user can stage files to via the available client tools or as part of a job submission. The location of the data, can therefore either be designated as a local systems path, or as an URL. If locally available, the client will upload the data directly as part of the job submission, on the other hand, if a URL is specified, the resource will download the data directly [116].

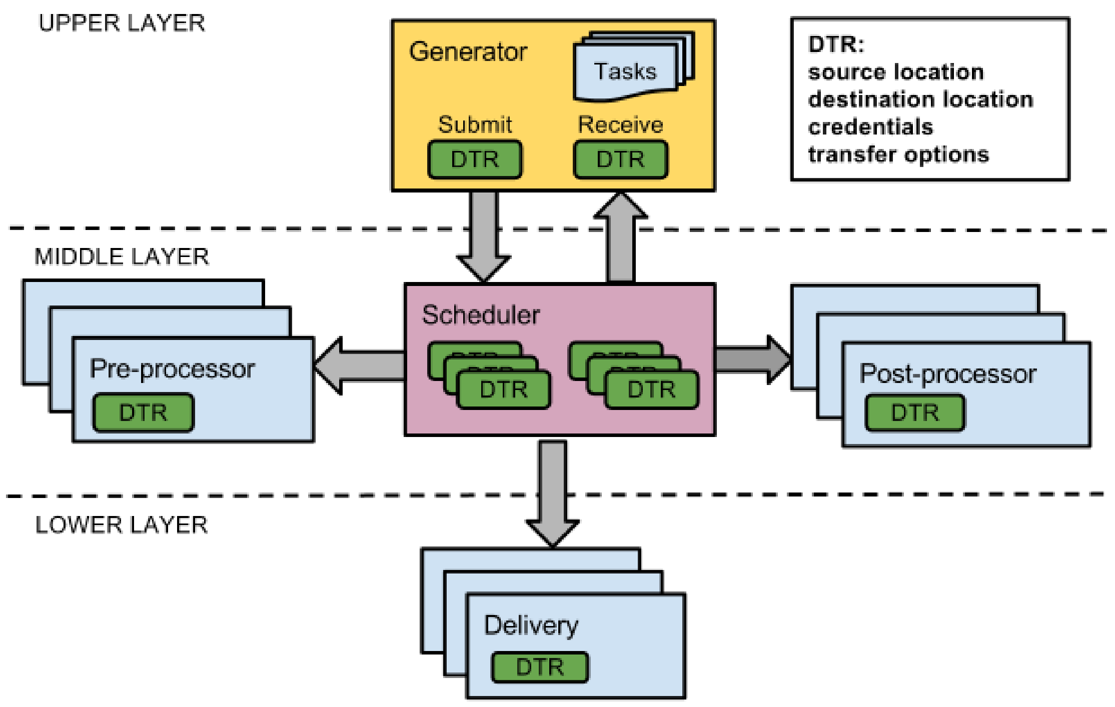


Figure 2.6: ARC's Data Transfer Request framework [113].

A commonality across these middleware frameworks, is that the data staging is described as part of the job definition, meaning that for each job the input data has to be specified at submission. This means that the implementation that constituents the job is being separated from the input it typically requires to be executed successfully. This enables the framework, to execute the same job with differing inputs. However, in terms of transfer ability of the job, that is the ability to execute the same job at another infrastructure or provider, the traditional Grid middleware approach does impose some difficulties. Firstly, because the input data is tied to the job definition, it is required that each infrastructure and provider support the same job definition. Secondly, the job definition format of how data files are staged has to be available between the infrastructures.

An alternative solution to this will be presented in Chapter 3.

2.2.8 Establishing a Grid

An example of how a modern Grid architecture could be defined can be seen in Figure 2.7. The small Grid presented here consists of a network of four organisations that enable Grid users access to a set of shared resources across their institutions. This could for instance include access to classic HPC Centers, Cloud infrastructures and data generated at a scientific instrument such as an X-ray beamline facility. In order to establish such a Grid, the classical approach would be to utilize a Grid middleware framework, like the Globus Toolkit, ARC, DIET or the Minimum intrusion Grid [20], to name a few. Given that an operating system had been pre-installed and configured on the Grid resources, the task would then typical involve the installation and configuration of the Grid middleware framework on each individual resource in accordance with their designated role, whether a Manager or Client agent of the Grid. However, this would also typically involve the convincing of the several organisational stakeholders that this is a worthwhile endeavor to begin with, including both management and the institutions IT infrastructure team that is to carry out this action. In addition, each individual organisation have to

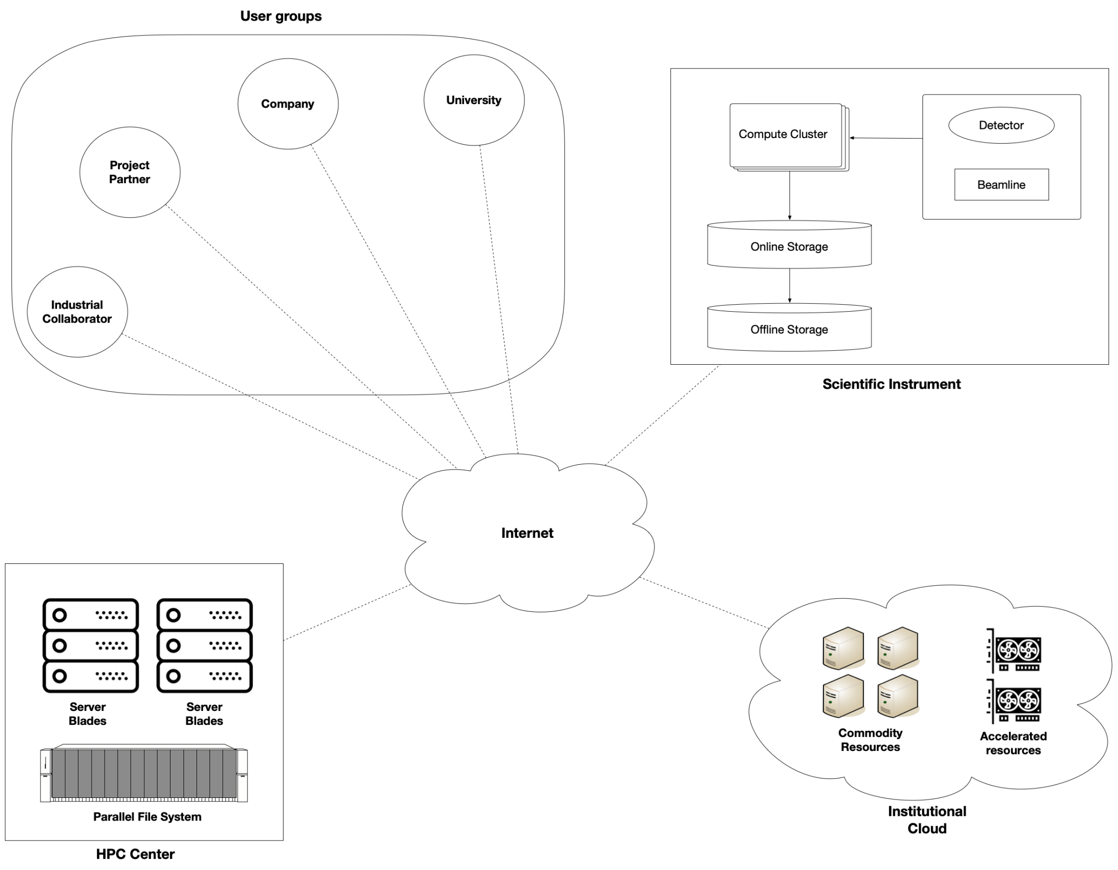


Figure 2.7: A small Grid example.

commit how long they will support such a network of resources, which involves the management, maintenance, support and development of the network with the associated cost. Adding this all together, the establishment of cross organisation Grids can often be a daunting task in terms of reaching the consensus that it actually should be done.

2.2.9 Grid Disadvantages

Part of the reasons why Grids themselves and the associated Grid middleware projects did not develop into sustained commercial and open source projects, is that they tried to be all encompassing. Specifically, they tried to do everything that was required within the Grid project as per the traditional Grid definitions. Furthermore, as previously highlighted, they did not experience enough utilization to be commercially viable. This, in part was also due to the work required to register, manage, and support individual resources before the introduction of technologies such as virtualization. Another common drawback of Grids is that to either join or exploit these resources restrict external institutions and organisations from exposing or sharing internal resources in an ad-hoc manner. Instead the enrollment of a new provider, or adoption of a new user typically involves a substantial bureaucratic process which is established to evaluate the potential impact from giving access to said purpose. Additionally, as a user of the Grid, the target provider of a particular job is often not exposed, limiting the choice of provider to the framework.

2.2.9.1 Modern Grid Alternative

The ability to easily establish minor Grid networks for smaller ephemeral collaborations and projects can be of great benefit to the involved organisations. For instance, at UCPH, as one would expect there are a wide range of ongoing scientific projects typically with the involvement of external scientific, educational and commercial institutions, one of these is a collaboration between the eScience group at NBI and the MAX IV laboratory on how to provide interactive and shared compute capacity in addition to data access at their respective institutions. In such scenarios, it would be of benefit to be able to establish smaller Grids for sharing resources between a select number of institutions for a particular project or collaboration. Furthermore, they should be established in such a way that it does not require a complete architecture redesign or relinquishing the resource to be completely dedicated to only the established Grid.

2.3 Cloud Computing

Cloud computing, according to National Institute of Standards and Technology (NIST) [112], is a model for enabling ubiquitous, convenient, on-demand network access to shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management [87]. Cloud computing is not in itself a new technology, but a combination of several existing technologies into a coherent service [59]. The cloud computing model exposes the shared pool of computing resources, such as networking, servers, storage, applications, or services in a variety of models. These includes Software as a Service (SaaS), Platform as a Service (PaaS), Infrastructure as a Service (IaaS) in the original definition [87]. These models describe what level of the cloud computing software stack the service is operating. An overview of these service models and how they apply to the cloud computing stack can be seen in Figure 2.8. Furthermore, Figure 2.8 also highlights examples of cloud services that each layer enables. For instance, AWS EC2 is an infrastructure service because it provides the user with the ability to provision and manage virtual machine resources on-demand. In addition to providing different functionality, the different service models also imply a separation of responsibilities of the computational stack. An overview of this can be seen in Figure 2.9. From Figure 2.9 we can also verify that the capabilities provided to us by the AWS EC2 service is made possible because the networking, storage, servers and virtualization components are provided by the IaaS. This implies that the responsibility of providing and managing the different components of the computational stack diminishes the further we move up the hierarchy in Figure 2.8.

Table 2.1: Cloud Computing Deployment Models per NIST definition [87]

Type	Description
Private Cloud	The cloud is provisioned exclusively by a single organisation comprising multiple users. Does not have to be operated by the organisation itself and can exist both on or off premise.
Community Cloud	The cloud is provisioned for exclusive usage by a specific community. Like a Private cloud it may exist both on or off premise.
Public Cloud	The cloud is provisioned to be used by the general public. The cloud exists on premise of the cloud provider.
Hybrid Cloud	The cloud infrastructure is a combination of two or more distinct cloud infrastructure such as a private and public cloud. Each of these are independent infrastructures but support the exchange of data and applications portability.

These service models are distinct for how the services are actually deployed. That is described by the Deployment Models concept [87]. The Deployment Model describes how the cloud is deployed, whether it is a Private, Community, Public, or Hybrid cloud. A detailed overview of the traditional Deployment Models can be seen in Table 2.1.

In addition to the NIST deployment model definition, there exist additional models in the general literature for how clouds can be deployed. Related to the concept of a Grid of Clouds is the Multi-Cloud and the Cloud Federation deployment models [175]. The definition of these is still up for debate, but in regards to this thesis, the

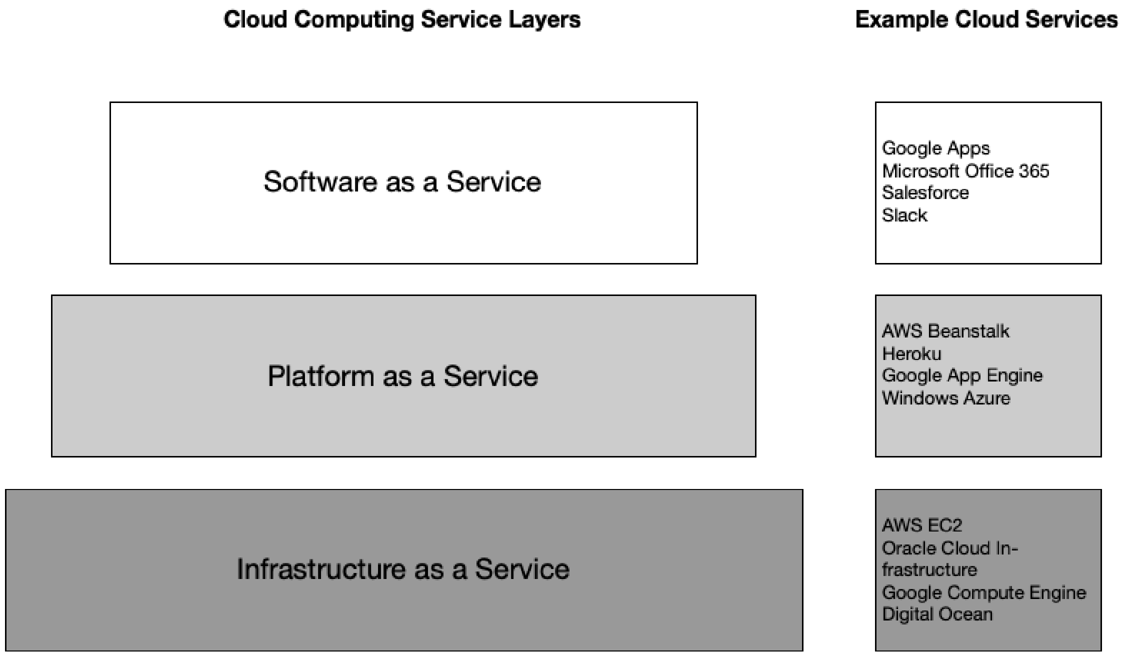


Figure 2.8: Updated Cloud Computing Service Stack with inspiration from [23].

definitions provided by [175] [172] [59] will suffice. Namely, that Multi-Cloud is defined as being a system that uses more than one Cloud Service Provider (CSP) to provide services from multiple providers. In addition, that in Multi-Cloud represents a user-centric solution [175], where the user is aware that the system is using multiple clouds and has to make an active choice about which ones to chose. In contrast, Cloud Federation is defined by [175] as being "an agreement for the cooperation among medium-sized Cloud providers, enabling them to share computing, storage and networking resources.". That is, the Cloud Federation enables the exchange of virtual resources such as VMs or PaaS services amongst the participating clouds.

2.3.1 Multi-Cloud

Multi-Cloud is a common research problem that has seen much attention in the cloud community [131]. The attraction of Multi-Cloud is that it potentially empowers the users with greater capabilities when utilizing cloud resources. This includes the increased access to cloud resources in general, for example compute, storage, networking etc. This, in addition to the ability to switch between multiple cloud providers, thereby increasing the flexibility of the CSP user, allowing them to utilize other CSPs to provide the same or additional services. In addition, it mitigates the so called lock-in effect. Lock-in is commonly understood as being the phenomenon when the cost of moving from one type of software, service, solution etc to another is so great, that it makes it very unattractive to do. In the case of Cloud Computing, it should be understood as the CSP having locked the user into their ecosystem, thereby limiting or making it infeasible for the user to switch to another CSP that provides the same or additional functionalities and services. A lock-in is typically caused by the CSP's usage of proprietary APIs and services that are not compatible with other CSPs or cloud infrastructures in general. For instance, the difference between the APIs for creating an OCI VM in Table 2.2 and an AWS EC2 VM as can be seen in Table 2.3. From these two tables, it is clear that a request to create an instance at OCI is not compatible with the EC2 API. It should be noted though, that some attributes for the two APIs have been left out in order for the tables to fit on a single page. This however does not detract from the point of them being inherently incompatible.

There have been many attempts at establishing common API standards in the area of cloud computing to establish cloud provider interoperability. Examples of attempted standards includes OCCI [121], CIMI [190],

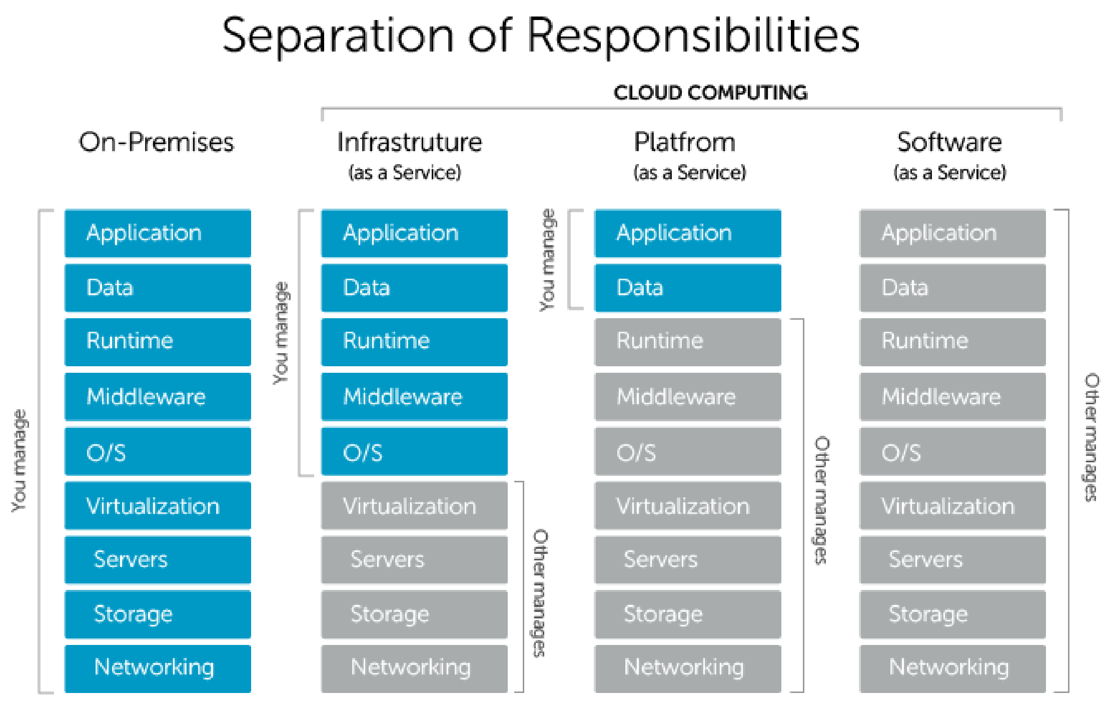


Figure 2.9: Cloud Computing Service Responsibilities [27].

CDMI [159], and TOSCA [120]. However, these have not seen widespread adoption or a lot of attention by the commercial CSPs, nor in the realm of academic and scientific cloud installations [172]. The most successful, as far as I have been able to determine is TOSCA, which have been adopted for a number of Multi-Cloud projects such as SeaClouds, Cloudify, and INDIGO-Datacloud [172]. Even with a number of adoptions, TOSCA is still limited to non-public Cloud Computing infrastructures. One of the reasons is that it is both a complex and complicated specification, in that it aims at describing the entire infrastructure via a single common standard. Other approaches includes the adoption of a common broker, that is then tasked with establishing semantic interoperability amongst the designated CSPs by establishing common abstractions. Similarly, a more practical approach would be to rely on the common abstraction via libraries such as Apache libcloud or Apache Jclouds [168], that allows for local Multi-Cloud usage without having to interact with an external broker.

In terms of orchestrating resources across multiple CSPs, existing frameworks includes Cloudiator, Roboconf, INDIGO-DataCloud, SeaClouds and beyond [172]. All of which supports some form of Multi-Cloud, that is, they are able to create resources at multiple CSPs. Each of these foremost enables the user in some form to orchestrate resources at a designated cloud platform. However, none of these frameworks enable the user to establish a federation of clouds. Furthermore, they are developed to be hosted as their individual services and are not suitable for ad-hoc installation and execution on, for instance, a scientist's laptop. Instead they focus on being persistent services with capabilities such as cross-cloud orchestration.

Multi-Cloud addresses several issues according to [131] and highlighted in [59]. Specifically, they list the following 10 issues that it addresses:

1. Dealing with peaks in service/resource requests using external ones on demand;
2. Optimising costs or improving quality of services;
3. Reacting to changes of the offers by the providers;
4. Following constraints such as new locations or laws;

Table 2.2: VM Instance API OCI Subset for LaunchInstanceDetails [124]

Attribute	Required	Type	Minlength	MaxLength
availabilityDomain	Yes	string	1	255
compartmentId	Yes	string	1	255
createVnicDetails	No	CreateVnicDetails	NA	NA
dedicatedVMHostId	No	string	1	255
definedTags	No	object	NA	NA
displayName	No	string	1	255
extendedMetadata	No	object	NA	NA
faultDomain	No	string	1	255
freeformTags	No	object	NA	NA
hostnameLabel	No	string	1	63
imageId	No	string	1	255
instanceOptions	No	InstanceOptions	NA	NA
ipxeScript	No	string	1	10240
isPvEncryptionInTransitEnabled	No	boolean	NA	NA
launchMode	No	string	NA	NA
launchOptions	No	LaunchOptions	NA	NA
metadata	No	object	NA	NA
shape	Yes	string	1	255
shapeConfig	No	LaunchInstanceShapeConfigDetails	NA	NA
sourceDetails	No	InstanceSourceDetails	NA	NA
subnetId	No	string	1	255

5. Ensuring the high availability of resources and services;
6. Avoiding the dependence on only one external provider;
7. Ensuring backups to deal with disasters or scheduled inactivity;
8. Acting as intermediary;
9. Enhance own Cloud service/resources offers, based on agreements with others; and
10. Consuming different services for their particularities not provided elsewhere;

A sub-category of Multi-Cloud is the area of Cross-Cloud. There is not an exact distinction between Multi-Cloud and Cross-Cloud. In general the most commonly stated difference is that Multi-Cloud uses multiple CSPs to provide a service or application, whereas Cross-Cloud is designed for transferring data and applications across different clouds more streamlined and cohesive [59]. This means that Cross-Cloud introduces additional enhancements to Multi-Cloud by enabling a single deployment of multiple instances for a particular application across different CSPs, thereby potentially unlocking more sophisticated best-fitting selection amongst the supported CSPs [172]. Furthermore, it can potentially also reduced the impact of a single CSP failure in terms of application availability.

2.4 Cloud Federation

As according to [175] not much research in decentralized Cloud Federations has been conducted. Existing results points out that the usage of a per-Cloud broker as a viable solution to interconnect cloud. By loosely coupling the clouds, each provider that can dynamically discover, select, and schedule their respective resources independently of each other. In Chapter 5 this thesis will present its contribution to providing a Grid of Clouds via

Table 2.3: Instance API AWS EC2 Subset for RunInstances [9]

Attribute	Required	Type	Constraints
AdditionalInfo	No	string	NA
BlockDeviceMapping.N	No	Array of BlockDeviceMapping	NA
CapacityReservationSpecification	No	CapacityReservationSpecification	NA
ClientToken	No	string	Max 64 ASCII characters
CpuOptions	No	CpuOptionsRequest	NA
CreditSpecification	No	CreditSpecificationRequest	NA
DisableApiTerminiation	No	Boolean	NA
DryRun	No	Boolean	NA
ElasticGpuSpecification.N	No	Array of ElasticGpuSpecifcation	NA
ElasticInferenceAccelerator.N	No	Array of ElasticInferenceAccelerator	NA
EnclaveOptions	No	EnclaveOptionsRequest	NA
HibernationOptions	No	HibernationOptionsRequest	NA
IamInstanceProfile	No	IamInstanceProfileSpecification	NA
ImageId	No	string	NA
InstanceInitiatedShutdownBehavior	No	string	(stop — terminate)
InstanceMarketOptions	No	InstanceMarketOptionsRequest	NA
Ipv6Address.N	No	Array of Ipv6Address	NA
Ipv6AddressCount	No	Integer	NA
KernelId	No	string	NA
KeyName	No	string	NA
LaunchTemplate	No	LaunchTemplateSpecification	NA
Monitoring	No	LaunchTemplatesMonitoringRequest	NA
NetworkInterfaces	No	Array of LaunchTemplateInstance...	NA
Placement	No	LaunchTemplatePlacementRequest	NA
RamDiskId	No	String	NA
SecurityGroupIds	No	Array of strings	NA
UserData	No	String	NA

the establishment of a decentralized broker that is responsible for discovering and selecting a resource provider to create a Grid of Clouds. An alternative to a decentralized broker, is the centralized approach, where a sole arbiter is the decision making entity, which every participant has to interact with to discovery, select and schedule resources.

2.5 Summary

In this Chapter, I presented the related and background work on how distributed architecture models like Grids and Clouds have developed. This includes the different approaches applied for providing computational resources to users. Furthermore, the Chapter covered some of the important components in establishing a Grid, including Authentication and Authorization, Resource and Service registration and discovery, Scheduling of computational jobs, and how data is managed in a distributed architecture. Additionally, other aspects of providing computational resources were covered, including Orchestration, Configuration, and how environments can be isolated in a shared settings, such as a provision resource. Furthermore, the Chapter covered how the Cloud notion developed from the Grid context and how it has been able to allow dynamic provisioning of resources. Finally, some of the most recent developments in this area was introduced, specifically how techniques such as Multi-Cloud, Cross-Cloud and Cloud Federations are being pursued in academia.

Chapter 3

Ubiquitous Data Access

In this Chapter, the research and results in developing the MiG Utils [95] data access library will be presented, that in addition to the presentation of the HISS model and what impact it potentially could have.

The idea and motivation for sharing data in an ubiquitous manner, was inspired from the scenario in the MultiScale Multimodal and Multidimensional for Engineering (MUMMERING) project [2]. This project was established with the aim of creating a research tool that could empower scientists with access to the wealth of 3D imaging modalities for applications in materials engineering, in addition to training 15 Early Stage Researcher (ESR) students [2]. This tool and applications are to be used to analyse the terabytes and petabytes worth of collected 3D imaging tomography data from scanned materials at various European scientific instruments. As part of this, a fundamental issue is how the data is to be managed as explained in Section 2.2.7.1. Specifically, where should the generated data be stored and how should it be made accessible for analysis. In [106] I introduced the notion of utilising the existing Imaging Data Management System (IDMS) [181] at UCPH as the designated data management platform. The reason for choosing this system was that it provided us with the necessary foundations to both access, manage, and process the generated data. The combination of these functionalities and a web based user interface made it applicable to our project. Furthermore, because the computational infrastructure knowledge of the participants in the project is wide ranging, it was important that the data management platform be usable across this range. For example, at one end of the range there were physicist that were mostly application and analysis oriented, in contrast to computer scientists that develop computational infrastructures and therefore required additional access to the underlying systems. Meeting such needs, could be split into two types of users, namely limited but with high emphasis on usability and interactive platforms, compare to low-level system access with commandline interfaces.

This scenario is not just applicable to the MUMMERING project, but is the case for many collaborations, be they in the world of academic or industrial. In addition, another typical aspect, is that in collaborations where the participants are a collection of cross institutional entities, a classic problem, is that both the data and analysis are kept within each organisation. A common data management platform, like IDMC facilitates the sharing of data, but it does not alleviate the fact that all datasets then will be located at the IDMS provider. This however might not be feasible, especially in collaborations with organisations that produce data in the tera and petabyte range like scientific instruments, such as ESRF, EuXFEL, and MAX IV. The issues here includes, the amount of data to be stored, and the amount that would have to be transferred, in addition to how access rights, including ownership, read, and write rights are to be given and upheld. Also, in a collaboration, such as an EU research project is by its nature limited in time. This means that at some point, the collaboration will have to be dismantled. The institutions involved would likely want to have continued access to the generated data beyond the scope of the project, likely feeding into other research projects. Having to rely on a central data repository controlled by a single organisation is therefore not suitable or adequate. Instead, a more realistic and manageable architecture, would be that each organisation is responsible for their own data management platform which is the primary repository for their data. Sharing within the collaboration is then based upon given access on a use-case basis where it is deemed necessary. An example of how I imagine this would work can be seen in Figure 3.1 and as I originally presented in [106]. Here I imagine that a user has some analysis that they would like to execute on a piece of data that is being generated by one of MAX IV's scientific instruments. In this scenario, its imagined that the instrument is able to store the data directly at the UCPH's data storage, namely, D1. Meanwhile, the user is able to implement their analysis on their own computational resource, such as their own laptop. Furthermore, if the UCPH D1 storage provides a method for accessing the data no matter the location of the data nor the analysis, that is enable the ability to define a unique access method, the user would be able to develop their analysis, such that it would be executable across any compute resource that can reach the D1 storage. In addition, this setup does not limit MAX IV from developing and scheduling their own analysis, which would subsequently be scheduled at their own Compute resource without interfering with their collaborators work.

By defining that the analysis states from where the data should be accessible, is inverse to how Grid frameworks have traditionally provided data input to a particular analysis. As explained in Section 2.2.7.1, these frameworks have commonly been responsible for both hosting the data and subsequently staging it on computational resources as part of the job execution with a name that matches the expectations of the analysis.

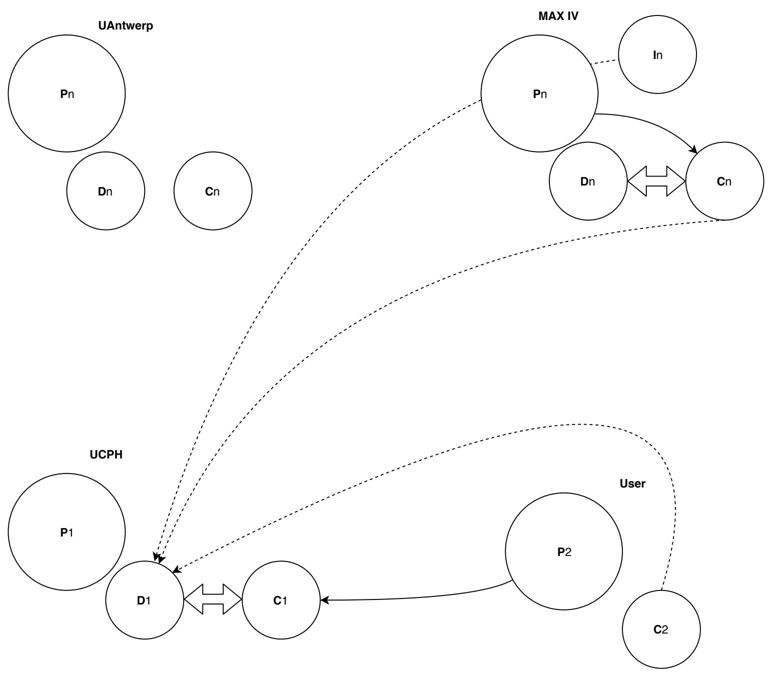


Figure 3.1: MUMMERING Organizational Overview. P = Partner/Organization, D = Datastore, C = Compute resource, I = Instrument, — = Submit execution on remote compute resource, - - - = load or store data from/on data store, $\Longleftrightarrow$ High speed 1 Gbit/s bandwidth. [106].

To provide this capability, MiG Utils leverages the fact that the IDMS system has an existing feature called Share Links for sharing datasets with external collaborators. This was used as the primary method for enabling Ubiquitous Data Access in [106].

3.1 Share Links

With the preexisting IDMS Share Links functionality, a user is able to share a specific directory or file path inside their personal user directory on the IDMS platform with external users that are not registered on the platform. At its core, this feature is similar to the Dropbox [39] sharing feature, or a Google Drive sharing [55]. While enabling additional security and control measure in how the data can be accessed, in addition to being accessible via multiple protocols. The supported protocols includes SFTP, WebDAV [188], File Transfer Protocol Secure (FTPS) [184], and SSH Filesystem (SSHFS) [163].

3.1.1 Security

The IDMS Share Link is a 10 character string of randomly selected ASCII characters. Given that there are ˜62 possible characters to chose from, we are able to generate 62^{10} distinct Share Links. This space size was chosen as to mitigate the potential for possible Share Link collisions and in turn mitigate the potential for brute force attacks.

At the creation of a Share Link, the sharer is able to select the access control rights that an individual with the link has. The selection here includes one of the following options: namely ReadOnly, ReadWrite, or WriteOnly. A big part of the Share Links additional security, is that the link is kept secret, and is not shared with anyone who is not intended to access the data. This model is the so called *security through obscurity*, which on its own, is not a very secure model. However it is not the only protection, the underlying IDMS system also imposes security mechanism to limit the abuse of or potential breach of Share Links. This includes rate limits and IP bans for any violators of a preset amount of attempts over a given interval.

In addition, it is planned that the Share Links will have an adjustable levels of security, which could include the requirement for passwords or private/public authentication keys. Furthermore, it has been contemplated that the users could be allowed to define the entropy of the Share Link, or be able to specify an expiry date. For instance, this could be achieved in a similar manner to how S3 implements the expiration. S3 does this by encrypting an expiration timestamp with a private key. The server then decrypts with the public key and checks the validity of the timestamp. This allows the link creator to set an expire time at the time of creation, with no possibility of manipulation. The value is simply passed as a query parameter.

3.2 MiG Utils

The MiG Utils [95] library was developed as an initial prototype to provide ubiquitous data access to the IDMS. As presented in [106] it is a Python 3 library that provides a simple interface for accessing, managing and interacting with a pre-existing Share Link. The library establishes an API that is similar to what one would expect for using the regular Python operating system interface [150], including functionalities such as open, close, remove, mkdir, rmdir, read, write, list, exists, seek, tell, and more. Currently the MiG Utils support two protocols to interact with a designated datastorage, namely, SFTP and SSHFS.

The overall architecture of the MiG Utils library can be seen in Figure 3.2. As indicated by the interface layer, the library makes an emphasis on usability for the users by establishing a set of interface helper classes. These classes implement the boilerplate code that is necessary before the library can establish a connection to the designated datastorage. The result is that typically the user only has to provide the designated Share Link, in addition to the datastorage in question.

A basic example of MiG Utils usage can be seen in Listing 1. In this listing, the IDMC system is defined as the designated datastorage, afterwards the current content of the share is listed. Afterwards a basic file called *example_write* is created and subsequently read from the share. As a simple validation of its correctness, the content of the created and read file's equality is verified before listing the content of the root location of the share. The result of executing this simple example can be seen in Listing 2 where as expected the *example_write* file is created and the content of that file is *Hello World*.

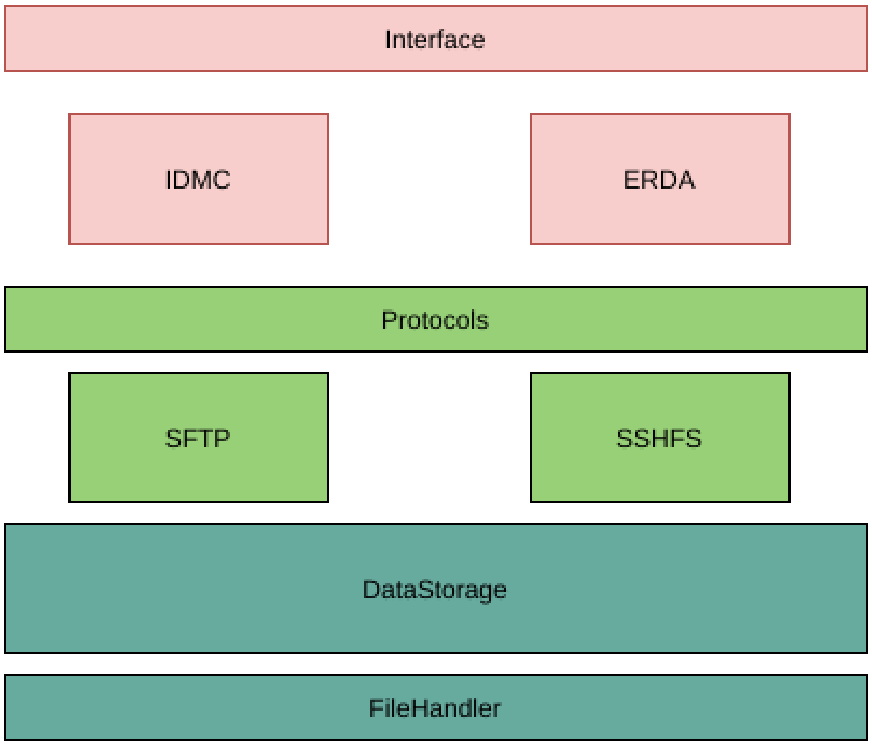


Figure 3.2: The Minimum Intrusion Grid Utilities Library.

```python
from mig.io import IDMCShare

if __name__ == "__main__":
    datastorage = IDMCShare("SHARELINK")
    data = b"Hello World"
    # List files/dirs in share
    print(datastorage.list())

    # Write binary string
    with datastorage.open("example_write", "wb") as _file:
        _file.write(data)

    read_data = None
    # Read the binary string
    with datastorage.open("example_write", "rb") as _file:
        read_data = _file.read()

    assert data == read_data
    print(datastorage.list())
    print(read_data)
```

Listing 1: mig_utils_hello_world.py.

```
(venv) rasmus@debian:~/repos/mig_utils$ python3 example.py
[]
['example_write']
b'Hello World'
(venv) rasmus@debian:~/repos/mig_utils$
```

Listing 2: mig_utils_run_example.py.

3.2.1 Benchmarks

To evaluate the effectiveness of the MiG Utils library, a number of benchmarks will be presented in this section. This includes how the library can be used to submit the same analysis across multiple systems without code adaption. In addition, it will be shown how MiG Utils can be utilized across different computational infrastructures to decrease the time it would have taken to collect the same number of results.

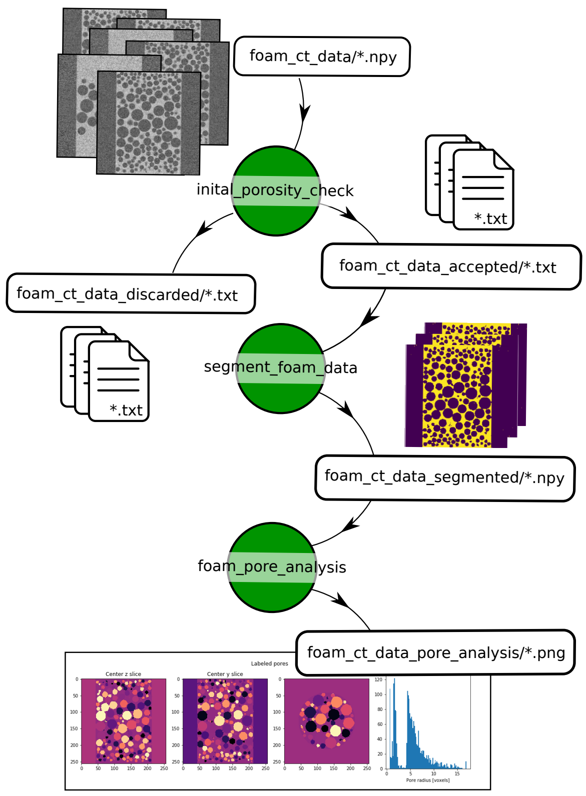
The first benchmark involves the analysis of a 3D X-ray computer generated computed tomography (CT) dataset of 100 samples. These samples are artificial generated to represent aluminum foam [158]. This is the same benchmark that was used in [81] to test the dynamic capabilities of a novel workflows framework that will be introduced in Chapter 4. The example is tasked with analysing the pore radius distribution in every sample. Here some samples only have a very few pores and need to be discarded. The aim therefore is to evaluate the CT datasets into which samples have a sufficient number of pores to be analysed. An overview of this process can be seen in Figure 3.3. As can be seen here, the benchmark is a three step workflow. Each of these steps requires both an input dataset and in turn generates a file, either for subsequent analysis or to indicate that the foam has to few pores. Additional details about what each step is doing can be seen in the following enumeration adopted from [81]:

1. Step 1 ('initial_porosity_check'): A two-component Gaussian Mix- ture Model is fitted to a small sample (around 1 %) of the intensity data, providing a rough idea of the air-to- aluminium;
2. Step 2 ('segment_foam_data'): In the first step of the segmentation process, noise is reduced using a Median filter. The filter kernel size is defined as a variable whose value is set in the Pattern. Thereafter, the image is segmented using Otsu thresholding [127]. Finally, a morphological closing operation is performed to remove possible remaining single-voxel noise;
3. Step 3 ('foam_pore_analysis'): To investigate the pore size distribution, the individual pores are identified using the watershed algorithm [35] with local peaks in a distance transform of the segmented data as seeds;

The workflow presented in [81] was adjusted so that it could utilize the MiG Utils library for staging inputs and outputs instead of relying on the MiG [20] to accomplish this. To evaluate and test that the library performed as expected, the adjusted workflow was executed in four different environments. Namely a normal desktop PC, a shared compute server, an external cloud resource, and a dedicated SLURM cluster. The specifications for these systems can be found in Appendix D.1. Since this benchmark is dependent on collecting and storing data to a remote storage unit, the bandwidth from the specific compute environment to the storage is going to have an impact on the performance. Therefore a set of bandwidth benchmarks was made from the benchmark environments to the IDMC system, an overview of these results can be seen in Figures 3.4a through 3.4b in Section 3.2.1.1.

3.2.1.1 Test environments

As is shown in the bandwidth Figures from 3.4a to 3.4f, the bandwidth from the different environments ranges from a peak read rate from MODI at 200 MBps, to the lowest write rate at 3 MBps from the internal OpenStack



foam_ct_data/*.npy
inital_porosity_check
*.txt
foam_ct_data_accepted/*.txt
foam_ct_data_discarded/*.txt
segment_foam_data
*.txt
foam_ct_data_segmented/*.npy
foam_pore_analysis
foam_ct_data_pore_analysis/*.png

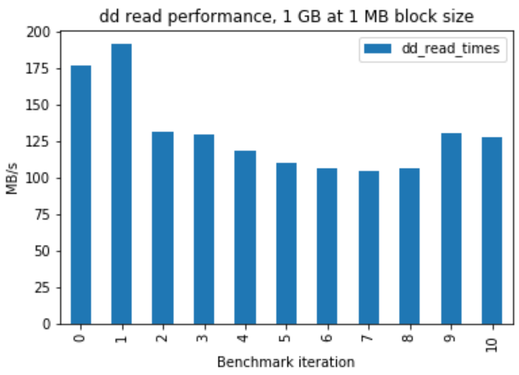
Figure 3.3: The foam analysis workflow with additional file images to make the data state clearer through the different stages. Taken from [81].

VM. The significance of these rates on any benchmark, of course depends on the size of the data that is being retrieve and stored during the execution. In the case of the MiG Utils benchmarks, the staging ranged from 67 MB as input for the initial_porosity_check_bench, to storing 136 KB in the case of the foam_pore_analysis_bench benchmark. Furthermore, the system specifications for the different benchmark systems can be found in Appendix D.1.

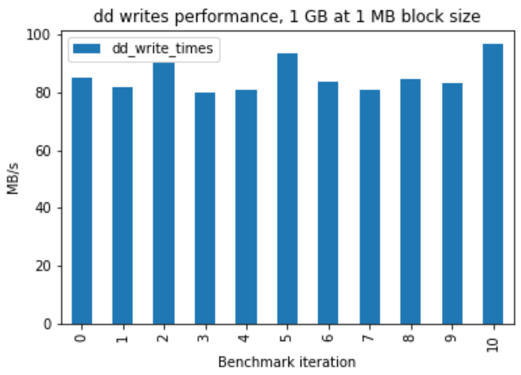
3.2.1.2 Benchmarking Mig_Utils

The results of the Mig Utils library can be seen in Figure 3.5a through 3.5c. The figures presents the results of running the ‘mig_utils_benchmarks‘ implementations, which can be found in Appendix E.1. In addition, the original implementations of which can be found in [80]. However, it should be noted in this instance, there were additional changes for benchmarking the pore analysis steps compared to the original implementation [80] as can be seen in Appendix 13, 14, 15. These changes were made to facility the benchmarking of the individual steps, and the ability to automatically schedule the 100 jobs per benchmark via the runner script as can be seen in Appendix 16. Furthermore, an additional SLURM based runner was implemented to enable the submission of the foam pore analysis steps to the MODI cluster.

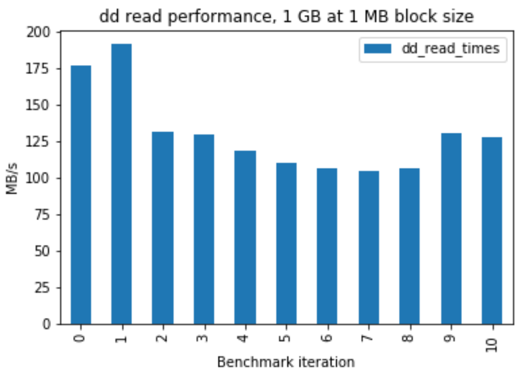
As shown in Figure 3.5a and Figure 3.5b, the DAG and MODI systems, exhibit a comparable performance on the ‘foam_pore_analysis_bench‘. This is in contrast to the Internal OpenStack VM consistently performs an estimated 20 to 40 seconds faster. The reason likely being that DAG and MODI are shared systems, whereas the Internal OpenStack VM is a machine with dedicated resources. Looking at the first two steps of the workflow, there is less deviation across the benchmarks, with DAG and the Internal OpenStack VM consistently executing the ‘initial_porosity_check_bench‘ step faster than the ‘segment_foam_data_bench‘ step. This is in contrast to the MODI benchmarks which in comparison have a fluctuating performance on the same two steps. However, it should be noted that on the MODI system, each individual three step benchmark were submitted in parallel, meaning that they as far as the system had capability for it, executed the benchmarks in parallel. This may have



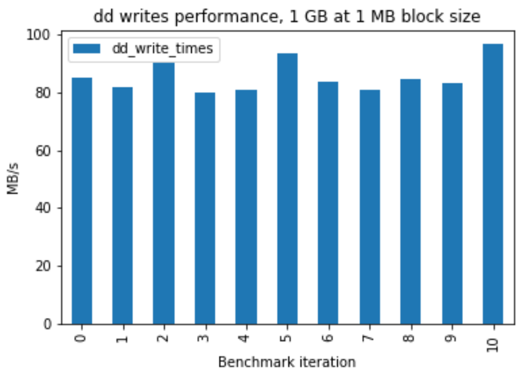
(a) Sequential Read from MODI SLURM to IDMC.



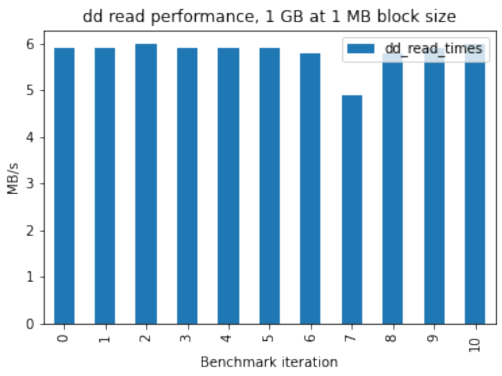
(b) Sequential Write from MODI SLURM to IDMC.



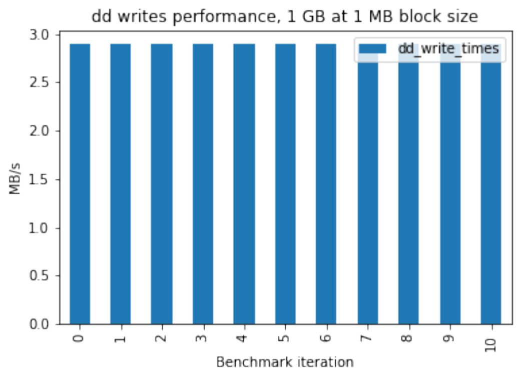
(c) Sequential Read from DAG to IDMC.



(d) Sequential Write from DAG to IDMC.



(e) Sequential Read from an Internal OpenStack VM.



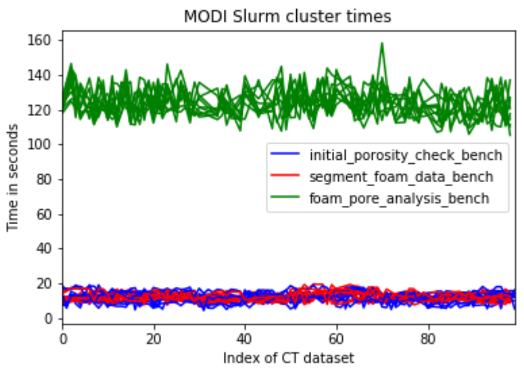
(f) Sequential Write from an Internal OpenStack VM.

Figure 3.4: Bandwidth measures from mig_utils benchmark environments, see Appendix D.1.

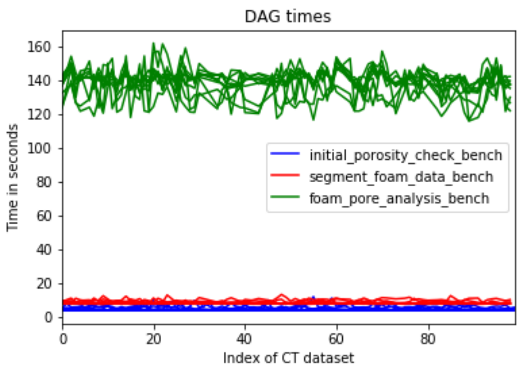
been the cause of the both increase and fluctuating runtime as shown in Figure 3.5a.

As these benchmark shows, utilizing the MiG Utils library enables the same implementation to be executed across multiple systems. This is possible through minor changes only to how the specific implementation loads and saves its inputs and outputs.

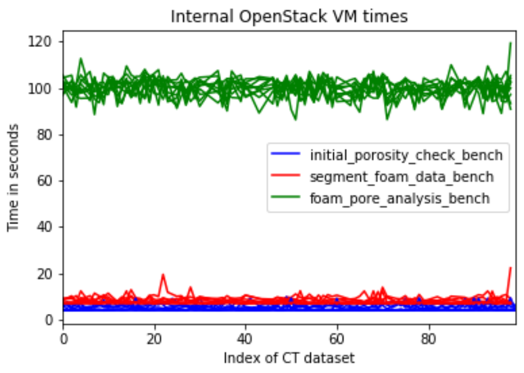
As the results also indicate, the performance of the individual steps is impacted by how fast the inputs and outputs can be served. However, whether to utilize the MiG Utils library and a central or several data storage repositories to facility the data management of a particular analysis in this seamless manner, has to be judged on a case by case basis. What has to be considered here, is the amount of data that has to be loaded and stored, for each job that has to be executed and whether the time it requires to complete this staging, makes it worthwhile converting existing implementations to the MiG Utils structure. In addition, what also has to be taken into consideration, is the scenarios in which the data has to be used. That is, does the owner and users



(a) MODI Compute times.



(b) DAG compute times.



(c) An Internal Openstack VM times.

Figure 3.5: Benchmarks for the workflow shown in Figure 3.3.

foresee a scenario as in the MUMMERING project as highlighted in Figure 3.1 where multiple collaborators would benefit from having cross organisational sharing of data. In such an environment, a tool like the MiG Utils library could be a useful complement to traditional sharing. Specifically, enabling the sharing of algorithms via their implementations without the direct sharing of the underlying datasets and results.

3.3 Data on the Grid

By abstracting the placement of the underlying datasets, it enables users to design their implementations to be independent of where they are executed. Instead, the responsibility falls on the organisation and the people responsible for providing data management and data access gateways. This includes the establishment of a sharing functionality that enables the tools like the MiG Utils library to both retrieve and store datasets. Inspiration to this can be drawn from both the presented IDMS Share Links [106], and commercial solutions such as Google Drive [55] and Dropbox [39]. Extending from these implementations, a three tier sharing stack could be imagined. Namely the ability to share at a Basic, Advanced, and Complex level.

3.3.1 Basic Sharing

Basic sharing would involve the basic functionalities provided by the current implementations of IDMC Share Links, Google Drive, and Dropbox. The basic part involves providing users with the ability to share the datasets anonymously at a unique endpoint, such as an URL. Firstly, the users should be able to define basic access restrictions on said endpoint, thereby covering controls such as determining whether requests are able to perform read and write operations. Secondly, the users should also be able to define whether anonymous users are able to access the endpoint at all.

3.3.2 Advanced Sharing

Advanced sharing functionalities builds on the foundations set by Basic Sharing and enhances the users ability to control how their datasets can be accessed. What is imagined at the advanced level it that a specific user is able to define the security level or policy of the endpoint. What is meant by the security level in this context, is which preset policies for each level that should be enforced on the Shared resource. One could imagine a three tier model of Low, Medium, and a High security level. Low security would be the default sharing as is the case with the current IDMC Share Links, that is that they are only protected by the generated unique id that the user is responsible for only sharing with whom they wish to share. Medium security would increase the required amount of authentication to require a valid and allowed set of credentials before a request is accepted, this could be either a valid password or an allowed public key authentication. High security could allow the user to define which IP addresses would be allowed to access the shared resource, High security could also require that two factor authentication is validated before a connection can be establish to the shared link.

3.3.3 Complex Sharing

Complex sharing would allow the same possibilities as Advanced sharing, but instead of having a set of predefined levels, the user would instead be able to enable or disable the specific policies manually.

3.4 The HIgh Throughput Model

As I presented in [107], data production and processing is a substantial challenge at scientific instruments such as MAX IV, where when fully operational, will have 16 X-ray beamlines that can potentially produce upwards of 18 terabyte (TB) per hour of imaging data. With the current approaches, a classic way to manage this data, would be to construct an in house infrastructure consisting of high speed parallel files and an associated batch oriented computer cluster. An example of this can be seen in Figure 3.6 which displays the MAX IV infrastructure architecture for handling beamline data flows and processing. The subsequent processing workflow typically involves three steps, namely, pre-processing, analysis, and postprocessing. The pre-processing covers the task for preparing the data before it can be analysed; this could include tasks such as transforming or transposing the produced datasets. Pre-processing could also cover tasks such as filtering out incorrect or unwanted data points.

A traditional approach to perform such workflow steps, would be to utilize a Big Data framework such as Hadoop or Spark to complete the workflow. An example of how this typically would be accomplished can be seen in Figure 3.7. When performing pre-processing tasks, such as filtering out noisy or invalid data points, a classic architectural approach also typically implies a substantial amount of data movement. An example of this can be seen in Figure 3.8. In this Figure, the beamline produces rays of X-ray that are directed towards the object at a given set of angles. Then the rays that interact with the object, will experience electron absorption from the object, the rate at which is determined by the material of the object. After the interaction, the remaining electrons are absorbed by the detector. It is from the detector absorption, and the measured electron levels from each angle, that the internal structure of the object can be imaged. The detector itself outputs the datasets for each angle as image slices that are stored at a designated storage location. This step is also known as acquisition in the a tomography workflow as illustrated in Figure 1.2. Subsequently, the object slices can then be used for reconstructing a 3D image, which then can segmented into areas of interest and eventually modelled for material characteristics.

Acquisition describes the phase of generating the raw datasets as described in the previous section. After the raw datasets have been written to the storage, typically in scenarios such as 3D imagining, a number of pre-processing tasks have to performed before the data can be successfully reconstructed. For instance, the generated data could have noisy signals that would have to be discarded before an adequate reconstruction could be performed. Discarding such noisy signals, requires the complete traversal of the collected datasets, which could involve processing TB of data. At MAX IV for example an 8 hour beamtime experiment can produce up to 144 TB. This implies that potentially 144 TB could subsequently have to be pre-processed, by having the

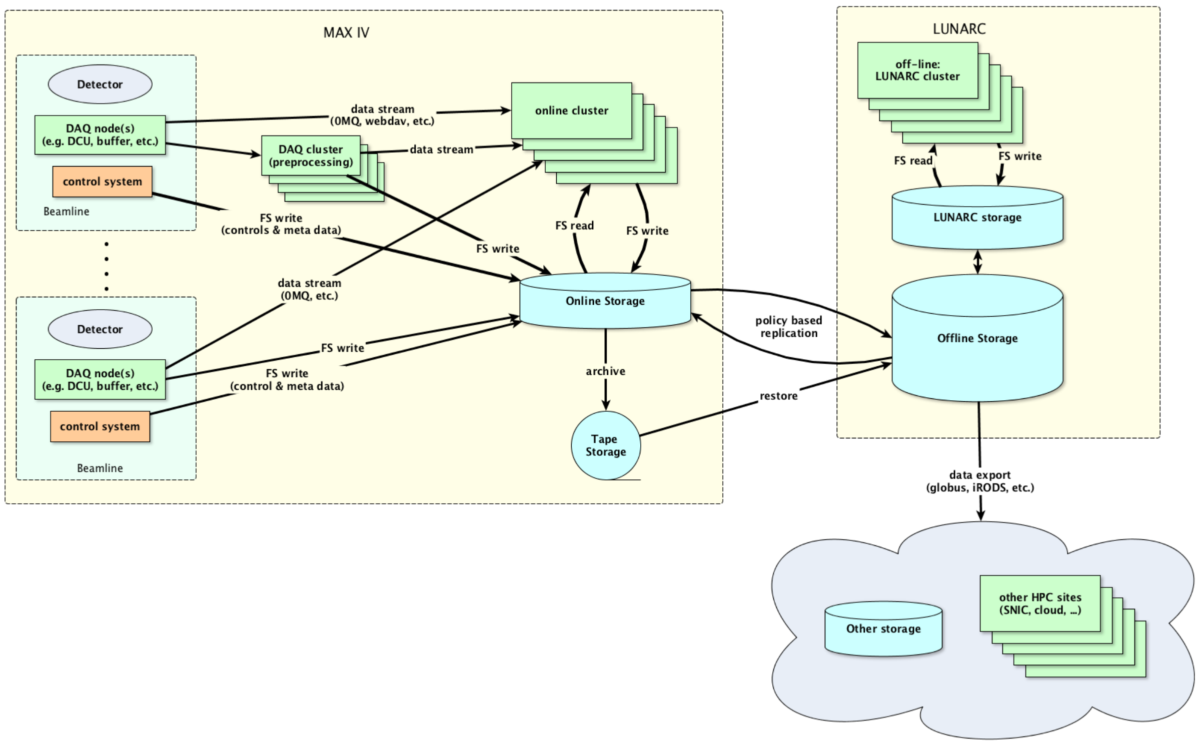


Figure 3.6: MAX IV Architecture at the end of 2019 [82].

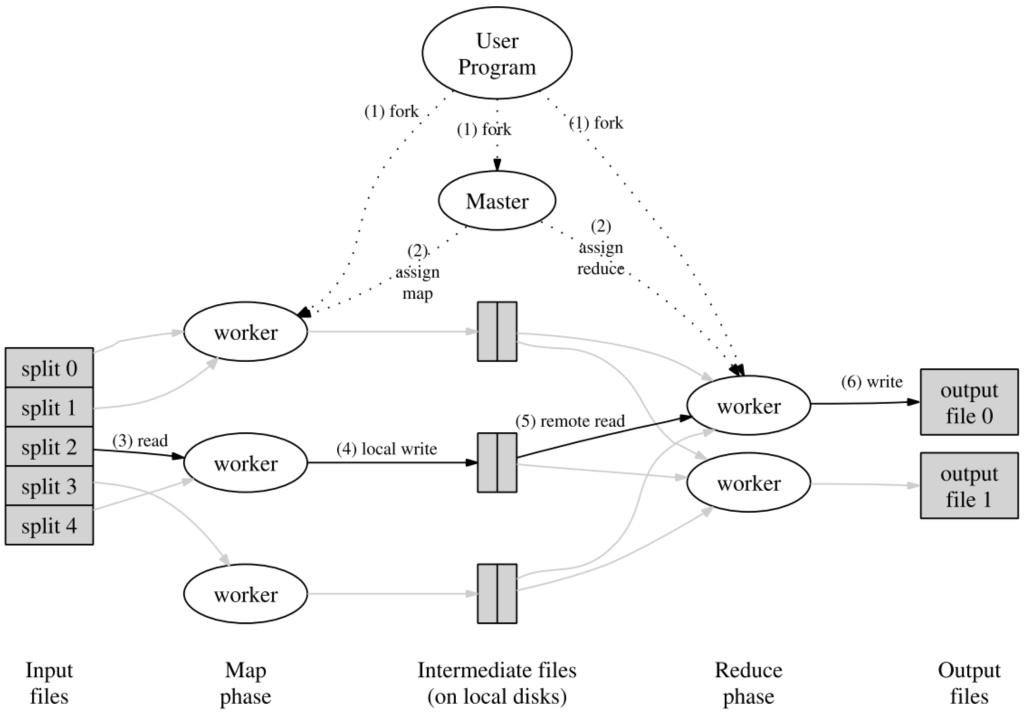


Figure 3.7: Hadoop HDFS Workflow [31].

individual nodes in the compute cluster load a certain subset of the data from the parallel file system before storing the filtered data back to the storage.

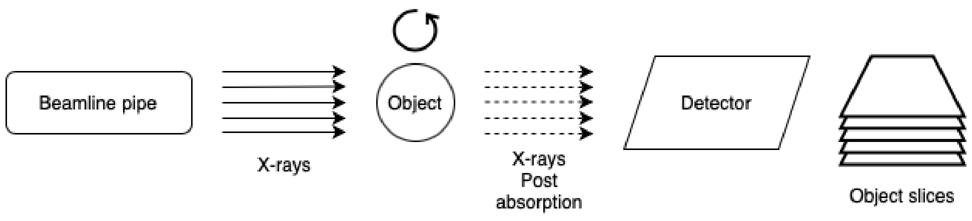
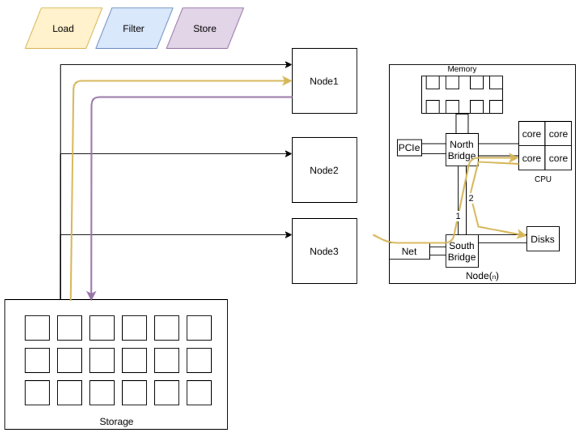
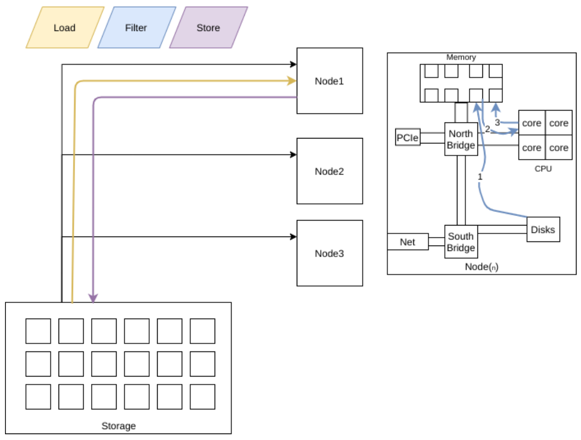


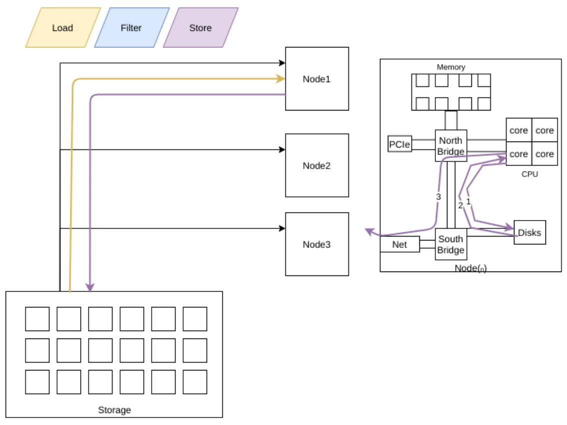
Figure 3.8: A Beamline scenario.



(a) 1. The data is loaded into CPU via the network interface, and the North Bridge. 2. It is then stored on the disk to be ready for subsequent processing.



(b) 1. The data is then loaded into memory where it is now ready to be processed by the CPU. 2. The CPU will then copy the data into its registers, perform some operations before storing it back into memory.



(c) 1. Filtered data is written back to the disk. 2. When ready, the filtered data is then transferred to the external storage where it is now ready for subsequent processing.

Figure 3.9: An example of data movement in a traditional compute-oriented infrastructure.

As is shown in Figures 3.9a to 3.9c, when conducting a pre-processing task, substantial amounts of data transfers can occur in a classic computational cluster setup, such as the one displayed from MAX IV. This movement includes both loading the data into a particular node, but also when transferring back the filtered data. Similarly, as was also illustrated by Figure 3.7, Big Data frameworks such as Hadoop can exhibit the same substantial amount of data movement, which is not unexpected since their task is to process data. It should be noted though, that how the data is being stored and handled within the node is only an example of a classic computer architecture, and will vary both depending on the vendor, design of the logic board, and the chosen set of components that a particular node have been configured with. For instance, specialized hardware like Remote Direct Memory Access (RDMA) network cards will change and optimize the data flow within a particular node, by being able to access data in the memory directly. Furthermore, frameworks like Hadoop, also have variants that allow for the data to be processed in memory without having to be stored on the disk itself. However, this does not detract from the fact, that any savings in data movement beyond this, still still wanted and beneficial.

The result of these additional data movement, is increased compute time consumed before the task is completed. Any reduction in terms of required data movement will have a beneficial impact on performance. Specifically, it is expected to result in lower execution times. The HISS model aims at achieving such reduction, by performing in-situ computations as it is either being stored or read. This model was developed in collaboration with MAX IV, as it was inspired by their need for in-situ computation during the acquisition phase of a beamline experiments. To perform such computations, we created the HISS design for establishing a front-end storage buffer system that would be able to perform computation on incoming and outgoing data streams, an overview of can be seen in Figure 3.10.

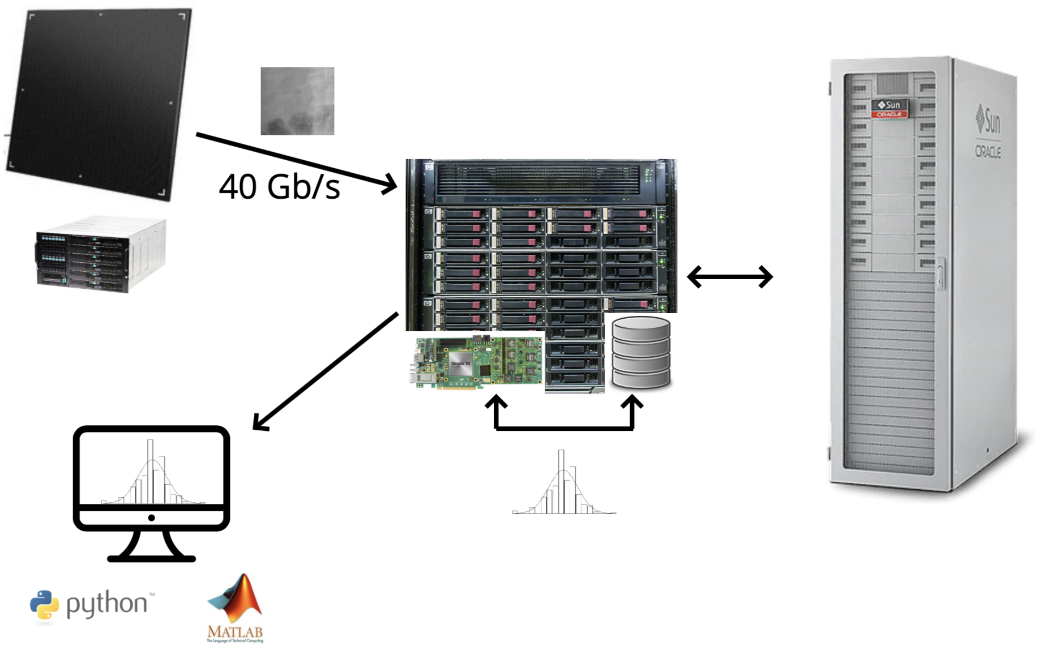
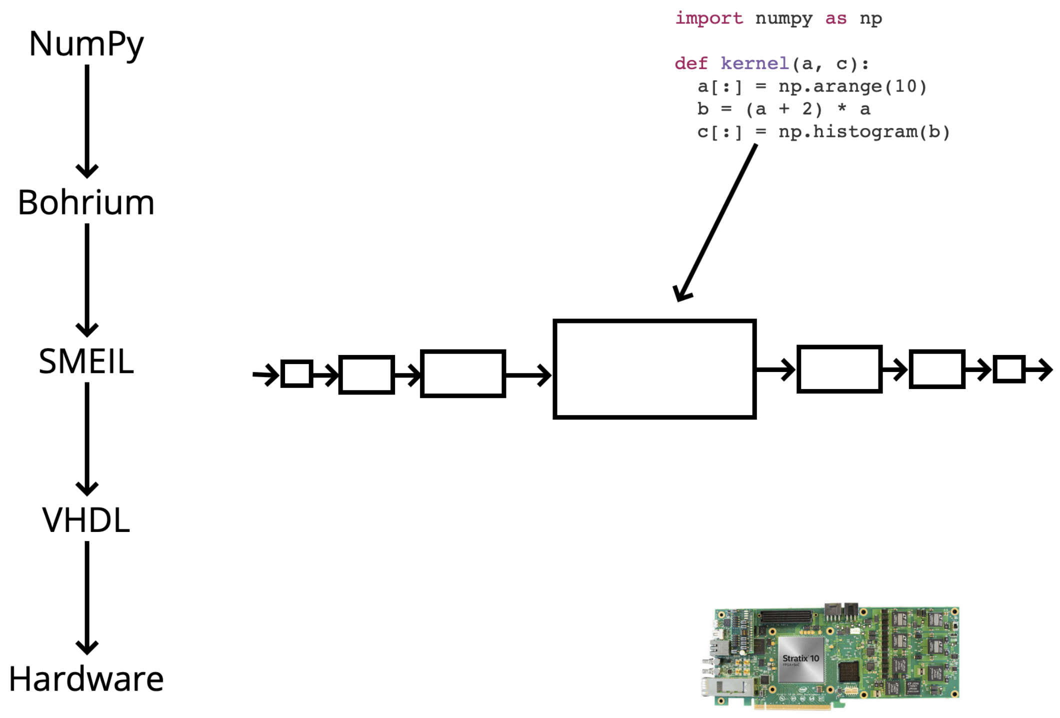


Figure 3.10: An example of a HISS use case [107].

The HISS system was designed to perform these in-situ computation by applying computational kernels to data streams as they passed through the system. The computation was intended to be performed by translating non-loop Python code to VHDL that can be synthesised to Field-Programmable-Gate-Array via the software stack shown in Figure 3.11. Furthermore, HISS was designed to act as a front buffer systems that is located in front of regular persistent storage systems such as a parallel file system, a collection of bare bone storage blades, or a tape archive. This implies that HISS is not designed to be a permanent storage location, but as an ephemeral location where data is fed through before it is being stored or delivered at its intended endpoint. In this regard of being a temporary storage location, the aim of HISS is similar to storage systems such as GekkoFS [180], with the extension of applying computational kernels to the data streams as they flow through the system. HISS was designed to be a distributed storage system with no central master server.

The result of this work, was a partial Go prototype implementation of an object storage file system [99].



```
import numpy as np

def kernel(a, c):
  a[:] = np.arange(10)
  b = (a + 2) * a
  c[:] = np.histogram(b)
```

Figure 3.11: From NumPy to hardware Synthesis in HISS [107].

This initial file system introduce a three tier hierarchy data storage structure. An overview of this hierarchy can be seen in Figure 3.12. In HISS, an individual Block is the lowest denomination of some form of data that can be interacted with, a Block is defined by a three attributes, that is Data, an ID, and, a Length. The ID is a unique identifier that specifies where a particular block is located on the storage device. A Stream, is defined by two attributes, that is an ID and an array of blocks. At the top of the hierarchy is the Collection, a Collection contains a number of Streams. The aim of this structure, is to enable the user to read and write spatially and temporally coherent data structures to and from the designated storage device. By doing so the design seeks to enable high bandwidth read and write operations from and to HISS.

HISS was designed with the three tier hierarchy to reflect the reality of scientific applications. Scientific applications have been shown to exhibit both read and write intensive I/O requests on both extremely large and extremely small sizes on sequential and irregular access patterns. [29]. In the initial design, HISS focused on enabling the computational kernels being applied to sequential and high bandwidth I/O patterns. An example of how Blocks, Streams, and Collections are utilized in the HISS prototype can be seen in Listing 3. Here a single hello world block is uploaded and subsequently downloaded to the client PC.

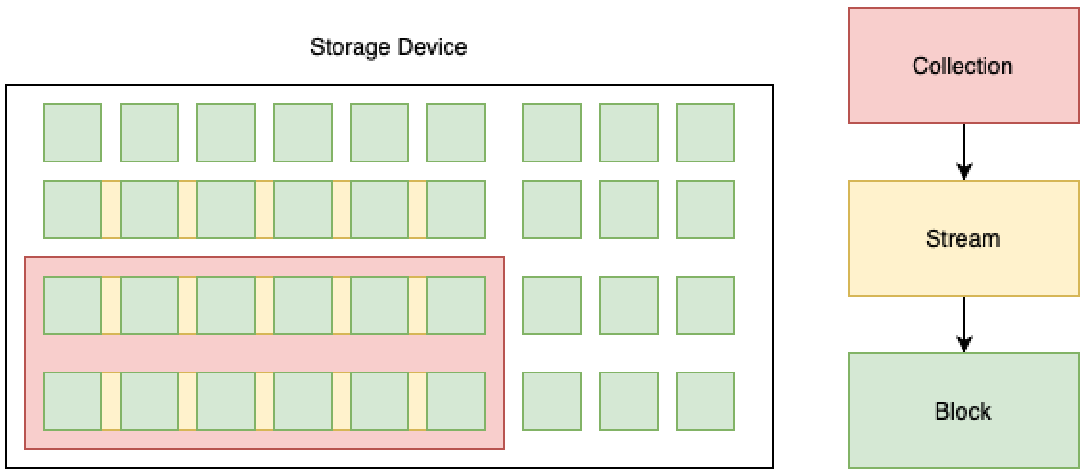


Figure 3.12: HISS Storage Object Hierarchy.

```go
import (
"github.com/rasmunk/hiss_prototype/pkg/client"
)

client := client.NewClient()
// Connect to the HISS endpoint (Specify public service IP)
client.Connect("127.0.0.1", "5001")
// Disconnect from HISS
client.Disconnect()

// Define a block
buffer := []byte("Hello World")
block := io.NewBlockV2(io.Data(buffer))

// Define a BlockStream
blocks := []io.BlockV2{*block}
blockStream := io.NewBlockStream(io.Blocks(blocks))

// Define a BlockCollection
streams := []io.BlockStream{*blockStream}
blockCollection := io.NewBlockCollection(io.BlockStreams(streams))

// Upload blocks in a user controlled size
client.Execute(block2.Upload(block))
client.Execute(stream.Upload(blockStream))
client.Execute(collection.Upload(blockCollection))
```

Listing 3: HISS hello_world.go.

The work on HISS did not progress beyond the initial design and prototype of achieving basic I/O of Blocks, Streams, and Collections. Multiple set of circumstances were the reason for this. The biggest of these was that, the HISS systems primary contribution would be the introduction of enabling the users to define and apply computational kernels in-situ to the I/O data streams. To accomplish this, the plan was to acquire the necessary hardware to conduct such tests and benchmarks. A number of viable hardware platforms were identified. The most promising of which was a variant of the Stratix 10 board and the prototype Samsung SmartSSD, which can be seen in the enumeration below. However these were subsequently deemed to be inaccessible either due to price or access and therefore made an actual implementation of the full HISS design infeasible.

1. Samsung SmartSSD [195]
2. DE10-Pro-GH2E2-165 - DE10-Pro GX 1650KLE Development Kit, 4GBx4 [165]
3. DE10-Pro-GH2E2-280 - DE10-Pro GX 2800KLE Development Kit, 4GBx4 [164]

3.5 Summary

In this Chapter, I presented my work in providing ubiquitous access to datasets across organisational boundaries. The MiG Utils library was presented, including how it can be used to make a foam analysis implementation location independent, by allowing it to both stage input data and store results at a predetermined data repository. The Chapter showed the impact of running a three step tomography analysis, including initial benchmarks of how the various stepped performed across five different systems. Furthermore, the idea and design for the HISS storage was presented, including how such a system could benefit large data producers such as scientific instruments. Specifically, that alleviating the classic compute cluster of pre-processing tasks, such as filtering to a front buffer storage system could potentially save subsequent processing time. In addition, the basic design for the underlying object storage engine was presented, including how collection of data could be stored in order for it to subsequently apply computational kernels to a particular data Block, Stream or Collection.

Chapter 4

Interactive Data Analysis

In this Chapter I will present the development and integration of interactive data analysis platforms that were developed throughout the thesis. This includes the establishment of two computational services at UCPH, namely the DAG and MODI. What will be covered, includes the design of these services, the developed architecture, and their subsequent use and contributions to the area of interactive compute platforms.

4.1 Data Analysis Gateway

Providing large computational resources to users has long been a challenge in the scientific community. Historically, the approach has been to establish large Unix-like compute cluster platforms that researchers, teachers, and maybe students could utilize at a defined consumption rate. However, to use such resources, the traditional approach has been to rely on command line access with accounts managed either via regular local user synchronisation, or via an integrated user database distributed across the computational nodes. This approach requires that the user has certain knowledge and skills in utilizing a shell-based system which the younger the user, the less likely they are to have encountered before. In today's world, non computer scientists are typically only familiar with GUI based interfaces such as web based platforms. The traditional skills of utilizing command-line skills can of course be acquired through training and usage, but such a time investment is in contrast to what the typical scientists, teacher or student is interested in. They want to utilize the computational resources to discover answers to their research questions.

4.1.1 Existing interactive services

In terms of existing solutions, several projects have dedicated their mission to provide web based interactive programming and data processing [5] [52] [91] [145]. Furthermore, specialized platforms such as RStudio [155], Google Colab [3], Kaggle [68] are also an area that has seen much development. However, as presented in [105], these platforms have certain limitations, such as the available hardware platforms, the given lifespans of a session, and the available development environments, an overview of which can be seen in Table 4.1 and 4.2. The most significant of which, is that the limitations and the overall usage policy is defined by the external provider. A highlight of which is that the most generous in terms of session time before being stopped was CoCalc with 24 hours. In comparison to this, internally provided services allow the institution to define a policy that suits their needs. Because of this, the DAG service was developed and introduced at UCPH. The platform delivers a JupyterLab [137] based web experience for scientists to perform interactive programming in various languages. Currently this includes Python, R, C#, and Q# with the possibility of extensions.

4.1.2 Jupyter

Jupyter [138] is the overall project that develops and delivers tools for interactive data analysis tools and scientific computing. At the inception, the foundation of this project was the IPython Notebook [130] format (.ipynb).

As presented in [103]. “It is based on interpreting special segments of a JSON document as source code, which can be executed by a custom programming language runtime environment (also known as a *kernel*).“ The Jupyter Notebook is then the subsequent developed web interface, to allow an interactive document experience. This in turn was replaced by the JupyterLab [137] interface, which aims at providing a Interactive Development Environment (IDE) in a browser setting. Both of these web interfaces are based on providing a single web-based user experience. Therefore, JupyterHub [136] is not a data analysis interface per say, but the de-facto standard to enable multiple users to utilize the same compute resources for individual Jupyter Notebook/Lab sessions. It does this through its own web interface gateway and backend database to segment and register individual users before allowing them to start/spawn a Jupyter session. In addition, it allows for the extension of both custom Spawners and Authenticators to allow for site specific implementations on how the users should be allowed to spawn an instance and how either the Jupyter Notebook or JupyterLab instances should be spawned [103]. An overview of its architecture can be seen in Figure 4.1.

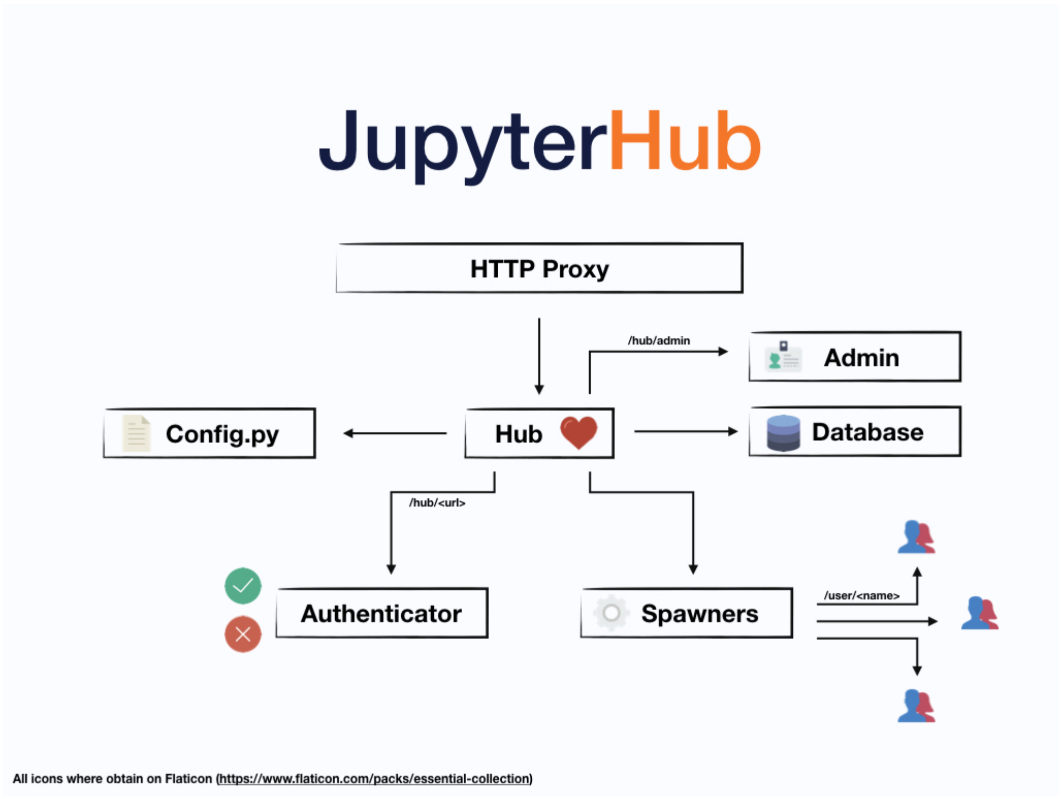


Figure 4.1: JupyterHub Architecture [142].

4.1.3 DAG in detail

The service leverages several known Jupyter based technologies, including the multi-user version of JupyterLab named JupyterHub [136] to allow multiple different users to initiate a personal environment. Furthermore, a common obstacle when utilizing either public or private interactive data processing environments is how data management will be provided. This is typically established via a public or private data platform. The mentioned public data processing services have a variety of limitations the most common being the maximum allowed space. Google Colab for instance defaults to using Google Drive which as of writing is limited to 15 GB of free storage [105]. An overview over the different interactive data processing providers and their limitations can be seen in Table 4.1 and 4.2.

In terms of providing the computational environment, the DAG service leverages a container based software stack to provide both application dependencies and process isolation. This was chosen over utilizing bare metal

Table 4.1: Subset of Jupyter Cloud Platforms Features, taken from [105]

Provider	Native Persistence	Languages	Collaborate	MaxTime (inactive,max)
Binder[1]	None	User specified [1]	Git	10m, 12h[2]
Kaggle [70]	Kaggle Datasets	Python3,R	Yes	60m, 9h
Google Colab [54]	GDrive, GCloud Storage	Python3,R	Yes	60m,12h* [3]
Azure Notebooks [90] [89]	Azure Libraries	Python{2,3},R,F#	NA	60m,8h* [4]
CoCalc [28]	CoCalc Project	Python{2,3},R,Julia,etc	Yes*	30m, 24h
Datalore [63]	Per Workbook	Python3	Yes	60m, 120h [5]
DAG [96]	ERDA	Python2,3,R,C++,etc	Yes	2h, unlimited [6]

Table 4.2: Hardware available on Jupyter Cloud Platforms, taken from [105]

Provider	CPU	Memory (GB)	Disk Size (GB)	Accelerators
Binder	NA	1 Min, 2 MAX	No specified limit*	None
Kaggle1	4 cores	17	5	None
Kaggle2	2 cores	14	5	GPU [7] or TPU [8] [69]
Google Colab Free	NA	NA	GDrive 15	GPU or TPU (thresholded access)
Azure Notebooks (per project)	NA	4	1	GPU (Pay)
Cocalc (per project)	1 shared core	1 shared	3	None
Datalore	2 cores	4	10	None
DAG	8 cores	8	unlimited [9]	None

or visualized environments; the reason for this was to benefit from the capabilities of containers compared to the other options as highlighted in Section 2.2.6. This includes the increased isolation compared to bare metal, the reduced load from operating system processes compared to virtual machines, and the increased efficiency when hosting multiple users compared both to bare metal and virtual machines. To provide the orchestration of the containers, DAG at the moment utilizes the Docker Swarm cluster container orchestrator [36], which is responsible for keeping track of each participating node, their health, and their scheduled services. It schedules services as underlying containers across the available nodes in a load aware manner. The reason being, that since every participating node in the prototype was internally hosted, Docker Swarm allowed for an instant an easy to manage infrastructure that could schedule JuptyerLab sessions via an adapted version of the Jupyter SwarmSpawner [94]. To provide data persistence, Docker Swarm uses the notion of volumes. A volume is a special created directory on the host within the Docker application's state. When a container or Swarm service for that matter is created, a volume can be associated with it. When this happens, the external volume directory is mapped into a specified path within the spawned container. Thereby it ensures, that any data written to that container path, is kept persistent by the host after the container has been terminated. However, this is not the case for any other data created inside a container, these cease to exists due to the ephemeral nature of the container's default overlay filesystem. An overview of the current DAG architecture can be seen in Figure 4.2.

At UCPH, the Electronic Research Data Archive (ERDA) provide several services. The most prominent of which is as the name indicates, data management. In addition, it also provides other services including archiving, sharing, and project collaboration [176]. ERDA like the IDMC is a specialized deployed version of the MiG. Because both ERDA and IDMC are the go to platforms for scientific datasets at UCPH, they were deemed as reasonable candidates for integrating with the DAG architecture to empower them with data processing capabilities. To enable access to the users individual datasets at either ERDA or IDMC, the underlying MiG project was extended with the ability to both act as a front proxy to a data processing platform like DAG. In addition, the MiG was also adjusted to allow the generation, management, and forwarding of time-limited user credentials to the data processing system. The choice fell on utilizing public key authentication as introduced in Section 2.2.1.1. These credentials were then subsequently used by DAG to mount the users individual home directory via SSHFS. To enable the DAG Docker Swarm service to receive and utilize the received credentials, the specialized docker volume plugin *docker-volume-sshfs* [93], was used. With this plugin, the DAG service

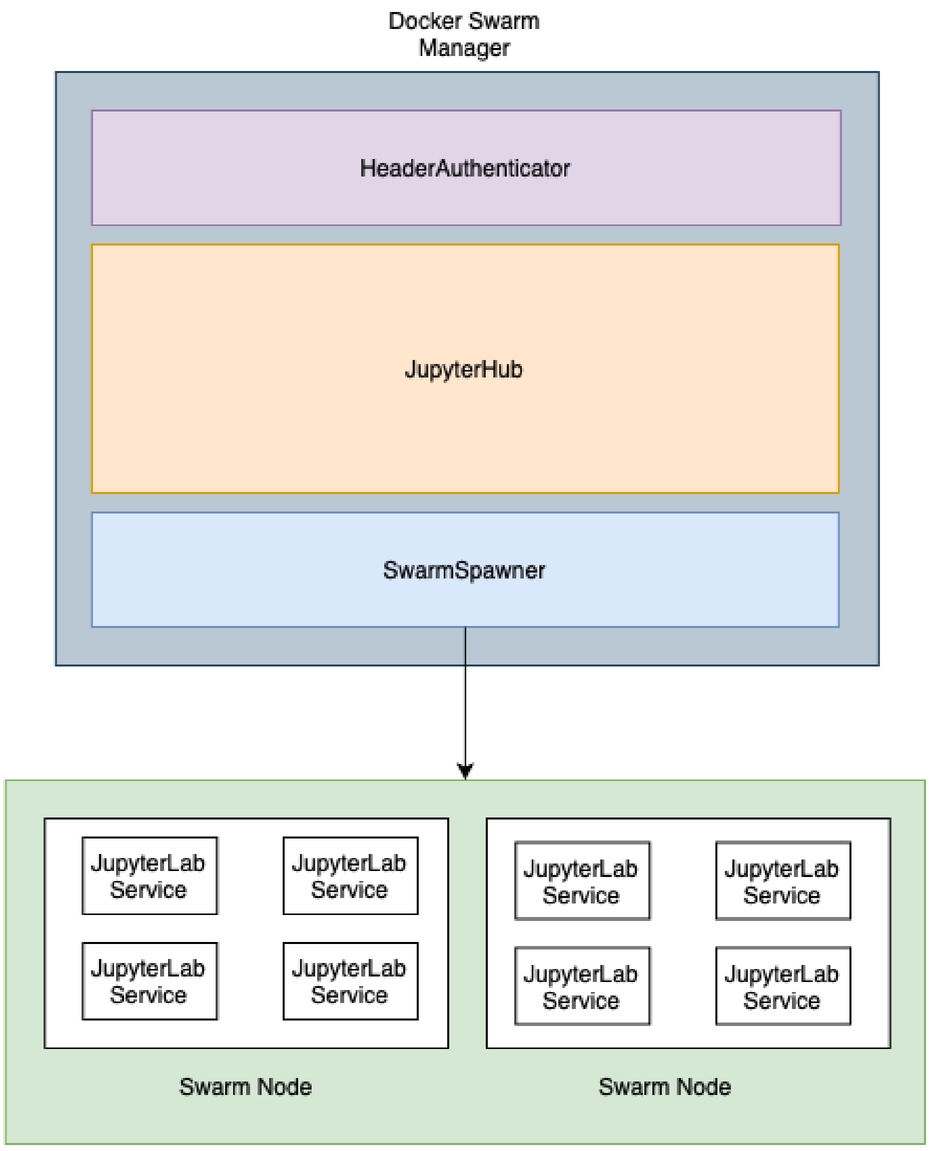


Figure 4.2: DAG Architecture.

was able to create and maintain the SSHFS connection in a host specific directory, prior to it being mounted into a spawned user Jupyter service. An overview of the integrated system can be seen in Figure 4.3.

As Figure 4.3 indicates, the integration was made possible by forwarding ERDA/IDMC user and mounting credentials to the backend DAG service. Since native JupyterHub does not support accepting such information, a supported extenstion instead had to be defined to make such integration possible. JupyterHub by default support three types of third party extensions. Specifically, the Authenticator, Spawner, and Services. When dealing with the authentication and storing of user associated information, the Authenticator is the defined part of the architecture that could support such functionality. In this scenario, it was deemed sufficient to pass the information by leveraging the data management platform's ability to forward user associated Headers since it would be acting as a front proxy to the DAG service. In terms of available options, none of the existing authenticators [66] had the capability to allow both for authentication and subsequent storage of forwarded headers. The closest existing authenticator that provide part of this functionality is the REMOTE_USER authenticator [30]. Although this does enable an Apache front proxy to forward the user credentials as part of that particular header, it does not allow for any customisation of any additional headers or which URL endpoint the headers should be posted

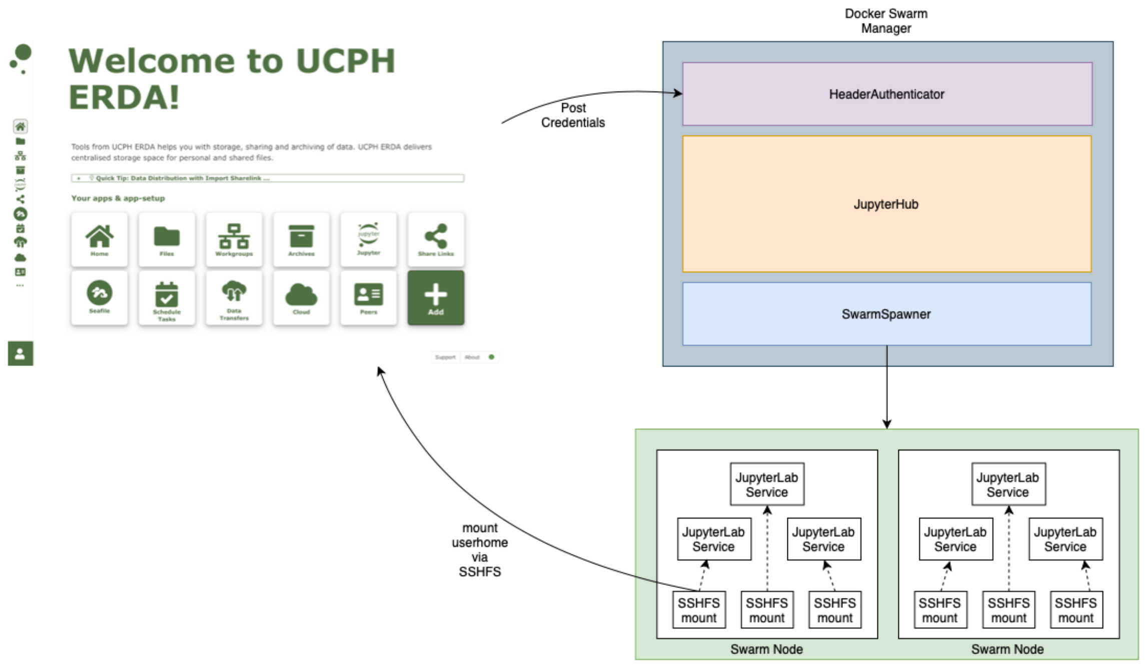


Figure 4.3: DAG integration with MiG.

to.

4.1.4 HeaderAuthenticator

In response to the limitations of the REMOTE_USER, I introduced a new Authenticator called HeaderAuthenticator, which alters the REMOTE_USER authenticator's functionality, such that the JupyterHub administrator can define a custom set of endpoints that can receive header posts, in addition to defining how they should be parsed. An example of how the HeaderAuthenticator can be activated in JupyterHub can be seen in Listing 4. The reason for the *enable_auth_state* boolean is set, is because the HeaderAuthenticator stores the transmitted data in the JupyterHub's user authentication state [67] dictionary, which JupyterHub automatically encrypts before it stores it in its database. Beyond defining the specific header names, and the endpoint on which they should be expected, the HeaderAuthenticator also allows the administrator to define how the particular headers should be parsed. It allows for this custom parsing via the *header_parser_classes* attribute as shown in example one in Listing 5. The second example in Listing 5 shows how a regular expression parser could be used too extract usernames from the header **MyAuthHeader**.

```
c = get_config()

c.JupyterHub.authenticator_class = 'jhubauthenticators.HeaderAuthenticator'

c.HeaderAuthenticator.enable_auth_state = True
c.HeaderAuthenticator.allowed_headers = {'auth': 'MyAuthHeader',
                                         'auth_data': 'MyCustomHeader'}
```

Listing 4: JupyterHub HeaderAuthenticator .

```python
from jhubauthenticators import JSONParser, RegexUsernameParser
c = get_config()

c.JupyterHub.authenticator_class = 'jhubauthenticators.HeaderAuthenticator'
c.HeaderAuthenticator.enable_auth_state = True

### Example 1 ###
# JSON Parser
c.HeaderAuthenticator.header_parser_classes = {'auth_data': JSONParser}


### Example 2 ###
# RegexUsernameParser
c.HeaderAuthenticator.header_parser_classes = {'auth': RegexUsernameParser}

# Email regex extractor
RegexUsernameParser.username_extract_regex = \
    '([a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+)'

# Replace every '@' and '.' char in the extracted username with '_'
RegexUsernameParser.replace_extract_chars = {'@': '_', '.': '_'}
```

Listing 5: JupyterHub HeaderAuthenticator with custom parsers .

4.1.5 Runtime Environments

To provide the users a set of pre-built environments a set of base container images were established at [104]. These are specialized extensions of the base image called *jupyter/base-notebook* provided by the Jupyter project [141]. Using this base image, has the benefit that the underlying Jupyter environment is both maintained and updated by the Jupyter developers, which alleviates the task of performing administrative burdens in keeping Jupyter itself up to date. The result of this was a stack of docker images [104], that covered some of the most requested packages bundled in different images. An example of this, is the *datascience-notebook* which contains what in our experience was the most common Python packages for performing data analysis tasks, including Numpy [118], SciPy [157], and Pandas [129], to name a few. The full stack is hosted at the DockerHub public repository, which makes it free to use by anyone which wishes to use it. Meaning that it is accessible to be used by any host that has Docker installed and external access to the Dockerhub repository.

4.1.6 Result

The result of this, was that UCPH was able to provide an integrated data management and data processing platform in an interactive environment for single user tasks. The university was able to accomplish this through the deployment of DAG and the integration of MiG. Since its deployment, the service has been a beneficial tool at UCPH, this includes serving teachers in delivering courses. In particular courses that covers areas such as data analytic, programming, or modelling has reported great benefit from the DAG service. A couple of course highlights include Introduction to Computing for Physicists [178], Applied Statistics [177] and Physcial Oceanography [65]. Beyond the installation at UCPH, a DAG-like service has also been deployed in a prototype environment at MAX IV. This was established from an ongoing collaboration between the eScience group at NBI and the Controls & IT [84] department at MAX IV. This has enabled continued refinement and development of the service. Since the DAG service is designed to allow each user to spawn an isolated environment with access only to the particular hosting node's hardware, it is not suited for applications that require access to multiple hardware nodes. Examples of this includes applications such as ocean or climate simulations that because of their scale, typically require access to multi-node resources to perform long term simulations in feasible a time scale. Instead, such needs could be served by an environment like a classic batch oriented cluster that allows cross node coordination. In its current inception, DAG does not enable the user to utilize such platforms. Because of this limitation, a complementary service was subsequently developed at UCPH, namely the MPI Oriented Development and Investigation (MODI) service.

4.2 MPI Oriented Development and Investigation

The initial motivation for MODI, as presented in Appendix B.1, was to enable the High Performance Parallel Computing course and researchers at UCPH access to a small scale high performance computing cluster with an interactive portal like the DAG service. The aim therefore was to establish a small sandbox cluster, that could both act as a teaching platform for how a classic batch oriented compute cluster is used and as a general multi-node compute resource. The target audience for this included both the students, teachers, and researchers at UCPH and similar institutions that utilize such systems. Therefore, the MODI system would act as a training facility towards using greater and more complex computational platforms such as a regular HPC Center [179], or a EuroHPC facility such as the Barcelona Supercomputing Center [15]. To provide such an experience, several features and tools had to be available on the platform. Foremost, it meant that all complexity should not be hidden away or handled for the user. The reason being that, in a regular HPC environment, the user is typically required to stage their datasets to a shared scratch space which each compute node has access to. Furthermore, these systems in contrast to DAG, still rely on a command-line interface, one of the most prominent reasons for this, is the legacy of the tools used to provide a multi-node batch-oriented compute cluster. For instance, in SLURM as presented in Section 2.2.3.1, the default interface is still the usage of command-line programs such as *sinfo*, *squeue*, and *sbatch*. Because of this, it was important that the MODI service delivered a regular command-line interface. However, this did not mean that MODI could just be a regular Unix-like compute cluster with a standard primary login node, which every user had to authenticate against to get access to the service. This would require the manual creation and management of both user creation, authentication, staging, and maintenance. Instead, the user management should be automatically handled, leaving only the task of data staging and job management to the user.

4.2.1 Difference to DAG

As in DAG, other solutions have focused on integrating a web portal environment with a compute environment like a batch oriented cluster [153], [145], [52]. MODI differs to these as presented in Appendix B.1 that it delivers all of its functionality through the common JupyterLab and JupyterHub interfaces by integrating it with an existing data management platform, namely, the ERDA/IDMC services at UCPH. Thereby it establishes an easy to access web powered computational platform that automatically integrates the users datasets and their user credentials with a classic SLURM cluster.

4.2.2 MODI Design

To provide the presented MODI service, several features and capabilities had to be established before such a service could be initiated. The features included the integration with ERDA/IDMC, the authentication and authorization of the user, the integration with a regular Unix-like batch oriented cluster, and the handling of user information so that it is properly translated between the involved systems. An overview of the MODI Architecture image can be seen in Figure 4.4. As indicated by Figure 4.4, the implementation of MODI, relies on a backend Lightweight Directory Access Protocol (LDAP) database to store user information. The reason for this was that in the Unix-like world, it is one of the most standardised and proven methods for providing distributed and uniform user accounts. [191]. A detailed explanation of how LDAP was utilized can be found in Appendix B.1. The HeaderAuthenticator was utilized across both the DAG and MODI service to enable the custom forwarding of user and mounting credentials from the ERDA/IDMC front proxies.

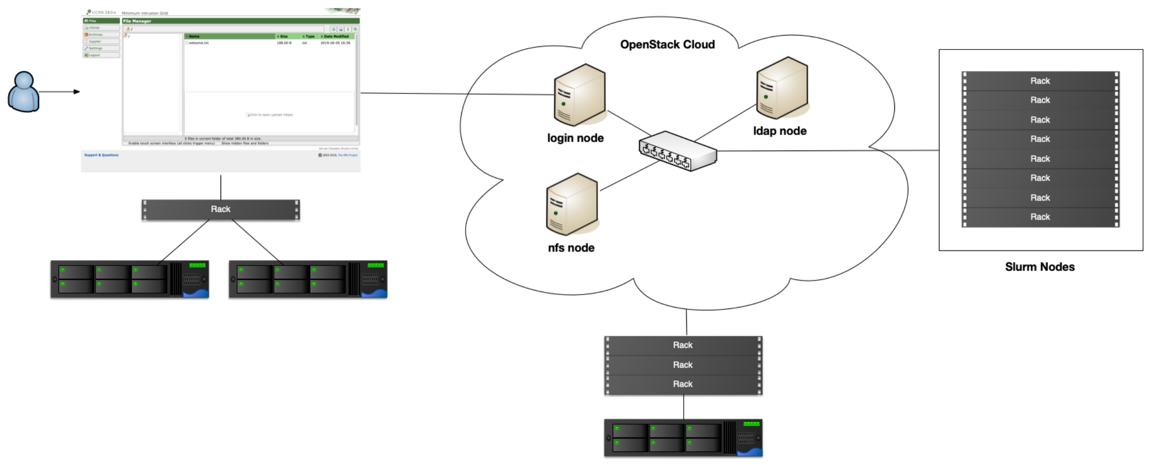


Figure 4.4: MODI Architecture.

4.2.3 LDAP Hooks

Because JupyterHub manages its own user accounts, their information had to be integrated into the backend LDAP user database to ensure consistency between the two systems. Although the existing LDAPAuthenticator [134] would allow JupyterHub to authenticate via an LDAP based database. Nevertheless, the LDAPAuthenticator is not able to fully automate the integration of external users into the subsequent LDAP provided database. The reason being that it does not allow for the management of user accounts beyond verifying that the requested user exists and that it can be correctly authenticated [134]. Because of this, a different approach had to be employed to deliver the MODI system as described. For instance, JupyterHub's API for Spawners does allow for the execution of external functions before a particular Notebook is spawned. Any function that should be executed at this state, must be assigned to the *pre_spawn_hook*. This functionality could be leveraged by the JupyterHub service to ensure the translation of its internal user information to be LDAP compatible before a user's JupyterLab session is spawned. Therefore this was deemed a good place for inserting logic that ensures that any external user is properly integrated into the LDAP database. I created the *ldap_hooks* [97] library to exploit the hooks functionality in the JupyterHub API, and to enable the automatic integration of external users into a designated LDAP based database. To integrate the library with an existing JupyterHub service, the service configuration has to define which library hook should be executed before a particular Notebook is spawned. An example of such a configuration can be seen in Listing 6. In this example, the *hello_hook* is a simple function that tells the Spawner to log a "Hello World" string before the Notebook is spawned. In contrast to this, the example in Listing 7 shows a realistic usage of the *ldap_hooks* library. With the configuration in Listing 7, the JupyterHub service is able to create a user in the LDAP based database before the Notebook is initiated. Achieving this was made possible with the *ldap_hooks* library *setup_ldap_entry_hook* functions, that validates the required information is provided by the specified *submit_spawner_attribute* and *submit_spawner_attribute_keys*. Hereafter, the *ldap_hooks* library utilizes the connection options to ensure that the *LDAP.url* is reachable and that the provided authentication details can establish a valid connection. Since certain pieces of information might not be externally provided, the library enables attribute values to be dynamically discovered via a set of predefined methods. The reason for this, is that before any user information can be successfully submitted to the LDAP database, it has to be correctly formatted as per the specified *objectclasses* that is expected by the underlying Directory Information Tree (DIT). To account for this translation and eventual missing required pieces of information, the *ldap_hooks* library implements a set of dynamic search methods. The set of available search constants includes the three options (*LDAP_FIRST_SEARCH_ATTRIBUTE_QUERY*, *LDAP_SEARCH_ATTRIBUTE_QUERY*, and *SPAWNER_SUBMIT_DATA*) in Listing 7 that are assigned to the *LDAP.dynamic_attributes* variable. In order, they try to extract the key value of the dictionary by utilizing the method described by the value constants. *LDAP_FIRST_SEARCH_ATTRIBUTE_QUERY* searches the LDAP database for the first matching object that

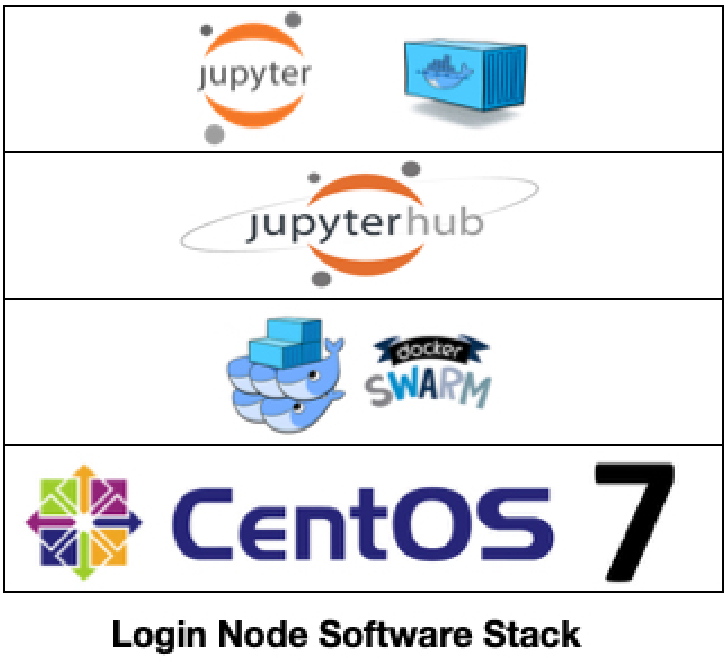


Figure 4.5: MODI Login Stack.

has the specified key attribute. *LDAP_SEARCH_ATTRIBUTE_QUERY* has the same behaviour, but potentially returns multiple matches of the specific attribute. The library conducts the searches by constructing search query based on the defined *LDAP.object_classes*, *LDAP.unique_object_attributes*, and *LDAP.search_attribute_queries*.

In addition, the *LDAP.search_result_operations* allows for the library to modify the result of searched attribute before it is associated with a new user. For instance, as shown in the example in Listing 7, the *uidNumber* attribute is incremented by one before it is associated with the new user and submitted to the LDAP DIT. Beyond the used methods in the example, *ldap_hooks* implements a number of additional search methods that can be found in the library itself. Additional details on how this is accomplished can be found in [97].

```python
# Example config
from ldap_hooks import hello_hook

c = get_config()
# Make the Spawner log "Hello World"
c.Spawner.pre_spawn_hook = hello_hook
```

Listing 6: Basic JupyterHub LDAP Hooks configuration.

```
# /// imports
c = get_config()
# General jupyterhub config definitions here
c.DockerSpawner.pre_spawn_hook = setup_ldap_entry_hook
# Authenticator setup
c.JupyterHub.authenticator_class = "jhubauthenticators.DummyAuthenticator"
# Define LDAP connection options
LDAP.url = "openldap"
LDAP.user = "cn=admin,dc=migrid,dc=org"
LDAP.password = "dummyldap_password"
LDAP.base_dn = "dc=migrid,dc=org"
# LDAP get dn to submit to the DIT
LDAP.submit_spawner_attribute = "user.data"
LDAP.submit_spawner_attribute_keys = ("User", "CERT")
# Prepare LDAP object
LDAP.replace_object_with = {"/": "+"}
# Dynamic attributes and where to find the value
LDAP.dynamic_attributes = {
    "uid": LDAP_FIRST_SEARCH_ATTRIBUTE_QUERY,
    "emailAddress": SPAWNER_SUBMIT_DATA,
    "uidNumber": LDAP_SEARCH_ATTRIBUTE_QUERY,
}

LDAP.set_spawner_attributes = {
    "environment": {"NB_USER": "{uid}", "NB_UID": "{uidNumber}"},
}
# Attributes used to check whether the ldap data
# of type object_classes already exists
LDAP.unique_object_attributes = ["uid"]
LDAP.search_attribute_queries = [
    {
        "search_base": LDAP.base_dn,
        "search_filter": "(objectclass=X-nextUserIdentifier)",
        "attributes": ["uidNumber"],
    }
]
modify_dn = "cn=uidNext" + "," + LDAP.base_dn
LDAP.search_result_operations = {
    "uidNumber": {"action": INCREMENT_ATTRIBUTE, "modify_dn": modify_dn}
}
# Submit object settings
LDAP.object_classes = ["X-certsDistinguishedName", "PosixAccount"]
LDAP.object_attributes = {
    "uid": "{uid}",
    "uidNumber": "{uidNumber}",
    "gidNumber": "100",
    "homeDirectory": "/home/{uid}",
}
```

Listing 7: Complex JupyterHub LDAP Hooks configuration.

4.3 Dynamic scheduling of tasks

When providing computational platforms, as highlighted in Sections 2.2.2 through 2.2.5, an important aspect is how scheduling is to occur. Traditionally, scientific workflows have been scheduled in a static manner. Static meaning that the set of tasks that have to be completed is predetermined when the workflow is scheduled. The static workflow itself has typically been parsed by various libraries and frameworks as a Directed Acyclic Graph (DAG). This has been sufficient in scenarios where the amount of tasks that have to be completed is known at the start. However, in scientific computing, the static approach have been identified by some [83] to not always fulfill the requirements of scientific workflows. At UCPH, this has been addressed by designing a dynamic scheduling solution for modern scientific workflows [81]. The design consists of splitting workflow tasks into their constituent parts by defining a combination of Patterns and Recipes which disjoints any meaningful inter-dependencies. The definitions for Patterns and Recipes are presented in the two following itemization, which are excerpts from [81]. It should be noted though, that traditional static systems can indeed change their execution branch as they progress through a workflow, but with the introduction of the Patterns and Recipes a workflow is inherently dynamic and does not require complex or costly patches to achieve it. A Recipe is defined by the following properties:

- **Name:** This is the unique identifier of the Recipe. It is used by Patterns to identify the linked Recipe, and by the implementation to keep track of changes to an already registered Recipe;
- **Instructions:** User defined code. For instance, a user's analysis algorithm. It may rely on input data or variables, provided by a Pattern.

A Pattern is defined by the following properties:

- **Name:** A unique identifier;
- **Triggering Event:** This describes how the systems should match a given event with the execution of a particular Recipe;
- **Recipe:** The name of a Recipe, used to define the processing taking place in a job;
- **Variables:** A set of variables to be passed to the Recipe by this Pattern at job creation. These could be any data structure understood by the Recipe and may include additional input files or possible output locations.

Via these two constructs, the user is able to define how processing should be scheduled by the system. This is accomplished without specifically defining how one task might lead to another, thereby only implicitly constructing how different processing tasks could lead to additional processing. Instead, the workflows becomes an emergent property of the system. The user is responsible for organising the conditions for when processing should occur via the construction of Patterns and Recipes. Specifically, the Patterns are constructed to specify what system events should trigger a certain Recipe that describes the instructions that should be executed.

The overall construct of Patterns and Recipes was named MEOW. To act as a prototype implementation, a Python library named mig_meow [79] was developed by David Marchant to allow programmatic management of MEOW. MiG was chosen as a good candidate for implementing MEOW, the main reason for this, was that it recently introduced an event-driven trigger system [16], that enables the scheduling of jobs based on a particular event being triggered. This trigger system could therefore be leveraged in MEOW Patterns to schedule a particular Recipe as a job to the MiG job grid infrastructure. To integrate MEOW into the MiG, the initial implementation relies on the user submitting their Recipes as .ipynb Jupyter Notebooks. By utilizing Notebooks, the workflows implementation would be able to support multiple programming languages as part of the Recipe Instructions field. Currently the MiG workflows implementation only supports Python, but this is not an architectural limitation. Additional languages could be supported if how the Notebooks inputs and outputs are recognized would be updated to work with other languages. Currently, both Patterns and the Recipe Notebooks are registered via the *'mig_meow'* library.

4.4 Jupyter Notebook Parameterizer

When an .ipynb Notebook is registered in the MiG, the MiG prepares a combination of executable statements that utilizes the *notebook_parameterizer* [102] and *papermill* [117] to parameterize and execute the registered Notebook. Papermill is a general tool to parameterize and execute .ipynb Notebooks, that expected the variables to be parameterized and defined in a single cell that is then tagged with the keyword *parameters* as can be seen at [117]. Although this is sufficient for simple scenarios, where the variable definitions to be parameterized can be separated into an individual cell, complex scenarios with variable definitions throughout multiple cells call for another solution. In such complex parameterization scenarios, the *notebook_parameterizer* allows for multiple cell parameterization without the tag requirement. I developed the *notebook_parameterizer* to achieve this by iterating through every code cell of a specified Notebook to conduct parameterization via variable value replacements.

In addition, the *notebook_parameterizer* is implemented as an command line tool, that requires two positional arguments and two optional once. An example of its usage can be seen in Listing 8. The first positional argument (NOTEBOOK_PATH) is the path to the Notebook that should be parameterized, the second positional argument (PARAMETERS_PATH) is the path to the parameters file that is used to parameterize the specified Notebook. The parameters file, is expected to be a YAML formatted file, containing key value pairs of the variable name of the variables the *notebook_parameterizer* should look for in the Notebook, and the value that it should assign to that variable. An example of this format can be seen in Listing 9. As part of the variable definitions in the YAML parameters file, the *notebook_parameterizer* also supports using system environment variables for parameterization, this is achieved by prefixing the parameter key value with the string "ENV_" followed by the name of the environment variable. These can then be expanded by the tool, if so specified via the *-e* flag. After applying a parameters file to an input Notebook the *notebook_parameterizer* outputs the parameterized Notebook at the path specified with the *-o* flag. The example in Listing 9 for instance could be used to parameterize the benchmark example in Chapter 3. The example currently hardcodes the two parameters *extra* and *threshold* while dynamically loading the *infile*, *outfile_insufficient*, and *outfile_thresholded* values. After generating a parameterized Notebook, the MiG implementation of the MEOW then subsequently uses Papermill to execute the resulting Notebook.

```
rasmus@debian:~/repos/notebook_parameterizer$ notebook_parameterizer -h
Usage: notebook_parameterizer [OPTIONS] NOTEBOOK_PATH PARAMETERS_PATH

Options:
  -o, --output_notebook_path TEXT
                                  Path to the parameterized output notebook
  -e, --expand_env_values         Should ENV_ prefixed parameter values be
                                  expanded to their matching OS environment
                                  variable value

  -h, --help                      Show this message and exit.
```

Listing 8: notebook_parameterizer.py usage example.

```
extra: 1
infile: ENV_WORKFLOW_INPUT_PATH
outfile_insuffcent: ENV_outfile_insuffcent
outfile_thresholded: ENV_outfile_thresholded
threshold: 10000
```

Listing 9: notebook_parameterizer.py example parameters file.

4.5 A Dynamic Workflow

Scientific applications that may benefit from dynamic workflows have certain key characteristics. Firstly, they exhibit a dynamic behaviour, where the set of expected execution steps cannot be predetermined, but are discovered during their runtime. Secondly, they rely on a set of iterations in which a similar task is executed, such as in a reduction scenario. Furthermore, other use cases include settings, such as environment monitoring of a certain environment where the set of available monitors that supply data to the workflow are fluctuating. In such as scenario, a static workflow might not be suitable, since it is not easily adaptable to include new monitors or remove existing ones, without redefining the entire workflow. Since this work, is not the central work of this thesis, only a simple example of a dynamic workflow will be provided, instead more advanced examples are available in [81]. A simple example can be seen in Figure 4.6. The workflow here is an illustration of how a dynamic scientific workflow could be constituted. In this instance, an experiment E, which could be an X-ray detector, would produce hundreds of data files that needs to be stored and processed. To begin with, E stores the data files in D1, each of these data files will then trigger a MEOW event by a registered Pattern. This event produced in D1 will then subsequently schedule a job (purple), which determines whether the data is relevant for keeping or if it should be discarded. The data that is deemed relevant, is written to D2, while the discarded ones might trigger additional experiments in E. When writing to D2, an additional MEOW events are triggered. Since two Patterns are registered at D2, the first one schedules a job for further processing (green) which produces D3. This first job, could an analysis or segmentation of the data received in D2. The second job (blue) is also scheduled but requires some human interaction before it can complete successfully, this could be that some threshold has to be set, or a region of interest has to be identified. Any identified data is then written to D4 which then triggers another layer of processing, such as a final analysis of the selected region which is the output to D5. MEOW makes such a scenario easy because of its inherent dynamic design, where workflows can emerge without being predetermined.

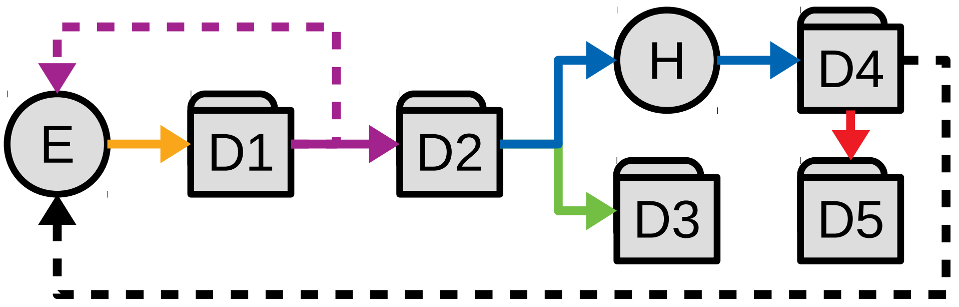


Figure 4.6: An example workflow. Data directories and external actors are shown as nodes. Solid arrows are MEOW workflow jobs. Dotted arrows show calls for an experiment run [81].

Management of MEOW workflows is possible via the *mig_meow* Python library that enables the user to submit MEOW related requests to a designated endpoint that supports MEOW workflows. Currently, MiG is the only known supported grid middleware framework that supports MEOW workflows. It enables this by establishing a JSON API endpoint that the *mig_meow* library can communicate with. The MiG exposes this endpoint at the */cgi-sid/jsoninterface.py?output_format=json* path, which supports the MEOW API. With the integration of MEOW into MiG, the user is able to construct Patterns with associated triggers within the MiG architecture. Because MiG is based on providing file based event job scheduling via shared MiG Workgroups directories and inotify events [16], the MEOW workflows were implemented to leverage the underlying event-driven architecture as Pattern Triggering Events. In addition, any submitted Recipe, is parsed and prepared as a job description by the MiG to be subsequently scheduled once the designated event is triggered in the filesystem.

4.6 Summary

In this Chapter, the work surrounding interactive data analysis was presented. This included the two developed services DAG and MODI at UCPH that enabled the university to provide interactive teaching and programming environments. Underlying both systems was the utilization of several technologies from the Jupyter project, including JupyterHub for multi-user management and JupyterLab for providing a web based interactive user environment. JupyterLab was the basis for providing several web based functionalities, namely interactive programming, shell based access, and general file management. The result of this work was a basic service definition, that enables institutions like UCPH to deploy similar DAG and MODI services. DAG was designed to provide the individual user with an isolated single machine environment by utilizing Docker and the underlying container technology. MODI on the other hand, was designed and implemented to provide users with an isolated development environment that integrates with a small scale classic batch-oriented cluster environment that allows for scheduling of multi-node applications. This was made possible by extending the existing DAG architecture with additional components. Specifically, the MODI cluster integration with JupyterHub was made possible with the introduction of the novel *ldap_hooks* library, which ensures that external user information are integrated with a designated LDAP based database DIT. Data management and access was established with the integration of the external UCPH ERDA and IDMC data repositories. This integration was made possible with the introduction of the HeaderAuthenticator JupyterHub authenticator, which enables the JupyterHub service to receive both user and additional data via Header forwards from a front proxy. Both of these services, are not restricted to be only applicable to UCPH, both DAG and MODI are generic software architectures that can be deployed with specialized configuration in different infrastructure environments. For instance, DAG was deployed as a prototype platform at MAX IV, which I have continuously collaborated with. Furthermore, the work in providing dynamic workflows was presented, including how it was integrated into the MiG and enabled the creation and execution of inherent dynamic workflows at UCPH as presented in [81].

Chapter 5

Proposing a Grid of Clouds

In this Chapter, the work related to integrating different cloud environments in an overall architecture is presented. This includes the initial work in introducing the underlying Cloud Orchestrator corc, and how it supports the future establishment and interconnecting of disassociated resources in separate cloud environments. Relating this to the existing literature, the work presented here provides a model for how a Cloud Federation or Grid of Clouds can be established with a decentralized broker architecture. In such as model, each individual entity in the grid, hosts an independent cloud broker that is responsible for coordinating and delegating requests for resources. The result of such a model, is that each organisation in the Grid of Clouds would be able to run their infrastructure, without having to request permission or coordinate with a central control unit, which would be the case if a centralized broker design was employed.

5.1 Cloud Orchestrator

When establishing a Grid of Clouds, the interconnection between cloud providers is typically established with the aim of sharing resources across the entities within the collaborating organisations. Before such sharing can happen, fundamentals, such as orchestration of said resources is required. Orchestration as defined in Section 2.2.1.3 is about providing a method for configuring, managing and coordinating computer systems such as establishing compute resources through a bundled and easy workflow [151]. Many projects have been devoted to the task of providing an over composing tool or framework to enable generic management and orchestration across multiple cloud providers. As highlighted in Section 2.2.1.3 and 2.3.1, this includes tools such as TerraForm, Cloudiator, SeaClouds, Cloudify, and INDIGO-Datacloud. Each of these are able to orchestrate resources at cloud providers, in addition to some of them that are able to provide Cross-Cloud functionality. However, these projects are developed as full stack platforms, that centralize the responsibility of orchestrating and managing the organisations cloud computing infrastructure. Although, this is not possible without the introduction of substantial complexity, which requires a team of people to operate and maintain. Corc takes another approach to these projects. It does not try to be an all-encompassing framework for orchestrating and managing a running infrastructure at an organisational level. Instead, it has three aims in mind. Foremost enabling a single user, such as an individual scientist, with the ability to orchestrate and manage Multi-Cloud resources without the involvement of cloud experts. Secondly, in extension to this, it aims at empowering them with the capability to schedule their applications or scientific analysis on such orchestrated resources. Thirdly, it aims at providing the foundations necessary, to interconnect the orchestrated resources in a Grid of Clouds, primarily by using its orchestration capabilities and management features to subsequently interconnect various computing infrastructures via a decentralized broker. In terms of implementation, corc was developed as a Python3 package. This was done because of the growing popularity and usage of Python in the scientific community. For instance, it is estimated by IEEE as being the top programming language in 2019 [61]. Because of this, it was deemed as a reasonable expectation, that the subsequent contributors and users corc would already be avid users of Python or at least familiar with if it they wished to contribute to the project. Also, since I had most experience with Python compared to other languages, it was deemed the best usage of time resources. In terms of usage, corc exposes two different interfaces, that is either via the provided CLI, or by using it as a framework within an existing application or service. No matter the choice in interface, by using the exposed interface classes, both approaches will utilize corc's underlying configuration to discover its infrastructure settings. In essence, what corc is contributing, is a combined toolbox and framework for orchestrating and conducting scientific experiments on a range of cloud providers that is expandable due to the extendability of the underlying framework structure. The tool-part of the package, aims to enable scientist to conduct computational experiment on large scale infrastructure without having to ask for resources at the organisation's IT department. Furthermore, corc itself provides Multi-Cloud functionality to the users, in that it enables users to switch between cloud providers by changing the designated target of the orchestration request; thereby the benefits of Multi-Cloud as described in 2.3.1.

5.1.1 Framework

Corc defines a set of abstract interfaces, that exposes APIs to conduct a certain task or functionality. For orchestration, corc introduces the *Orchestrator* abstract API, which can be seen in Appendix 18. As shown here, the interface provides a set of methods, which a specific provider specification can choose to implement. The point of each method, is described by the doc-string associated with each method. Particular highlights include the *setup* and *tear_down* methods that are responsible for controlling the lifetime of resources. The others are complementary to these functionalities to determine whether a resource has been orchestrated or not. Beyond Orchestration, the framework also defines APIs for how a Scheduler should operate and how Storage should be provided. These two API definitions can be found in Appendix 19 and 20.

To implement a provider into the underlying framework, corc currently adopts two approaches. The most straightforward approach is to utilize a well established abstraction library that already implements the required functionality for orchestrating resources at said provider. As was highlighted in Section 2.3.1, the Apache libcloud library is exactly such a project, abstracting as of writing more than 30 providers [168]. Therefore, it was deemed a reasonable candidate as an initial cloud provider, which enabled the integration of the AWS EC2 into corc, with the possible of additional libcloud providers with little effort because of the introduction the general Apache Orchestrators. Specialized cloud providers that are not supported by such general libraries often defines their own set of development frameworks and usage libraries. For instance, OCI is currently not supported by any known Pythonic abstraction library. Therefore, it was integrated into corc by utilizing the OCI provided oci-python-sdk [126] with a set of complementary OCI Orchestrators.

5.1.2 Architecture

Corc is designed such that the different responsibilities you would expect in orchestration is split into their own compartments. An overview of this can be seen in Figure 5.1. The Orchestrator itself is devised from a set of underlying sub-components as shown in Figure 5.2. It exposes the combined functionality of these sub-components via the common API, which can be seen in Appendix 18. The sub-components are provider specific, meaning that each provider has to define an implementation for how for instance orchestration is to occur. For instance, both the OCI and AWS EC2 provider implementations each have an instance Orchestrator implementation which utilizes the shared sub-components to provide the functionality defined by the Orchestrator API.

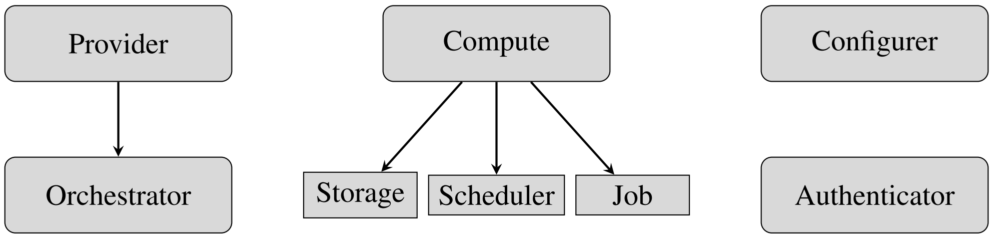


Figure 5.1: Cloud Orchestrator Framework Architectural Overview [105].

Similarly to the Orchestrator, corc also defines a division of responsibility for providing a Scheduler; an overview of its component framework can be seen in Figure 5.3. The Scheduler also exposes a common API, which can be seen in Appendix 19, where also each scheduling backend, be it Kubernetes, or a simple process submission, is defined as a particular Scheduler extension. In terms of configuring orchestrated resources, corc currently only implements the AnsibleConfigurer, which as highlighted in 2.2.1.4 is but one of many possible configuration management tools. The reason for using Ansible in this instance, was that it is a minimal configuration tool, that does not require a daemon running on either the platform that is responsible for carrying out the configuration, nor on the resource that is to be configured. The simplicity of Ansible's architecture allows for minimal intrusion, imposed requirements on the overall architecture, and potential failures of having to ensure the state of daemon applications. However, a limitation is that it only applies the configuration, it does not ensure that the configured resources are consistent to that configuration over time. Since the intended primary task of corc is to orchestrate ephemeral resources that have a limited lifespan to for instance carry out an experiment, this limitation was deemed acceptable. Furthermore, in use cases such as hosting Jupyter Notebook applications on orchestrated resources, this also allows the user to configure the resources to their needs after it has been deployed and configured by corc. Making it malleable by each user needs beyond or as a complement to the standard corc configuration.

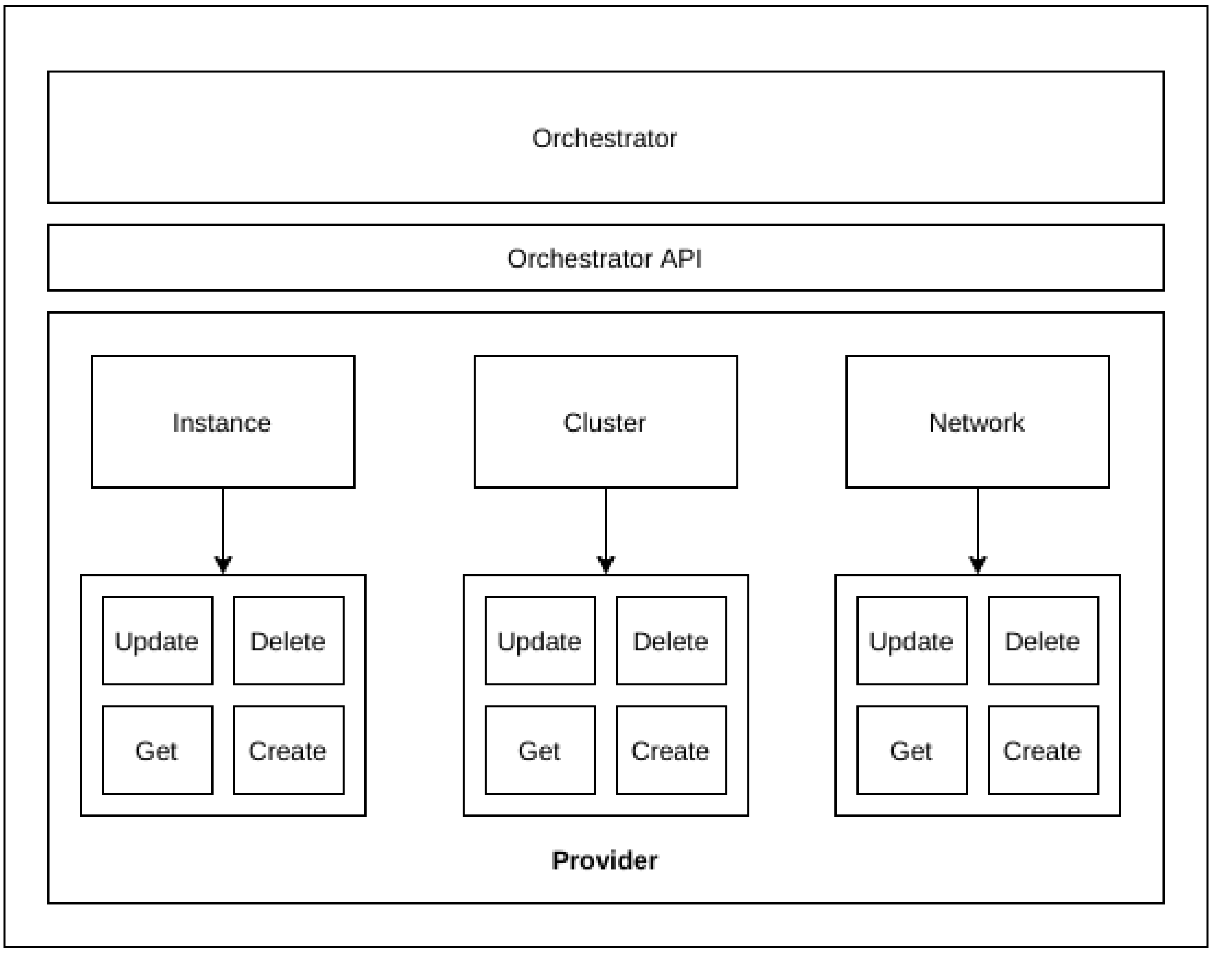


Figure 5.2: Corc Orchestrator Components.

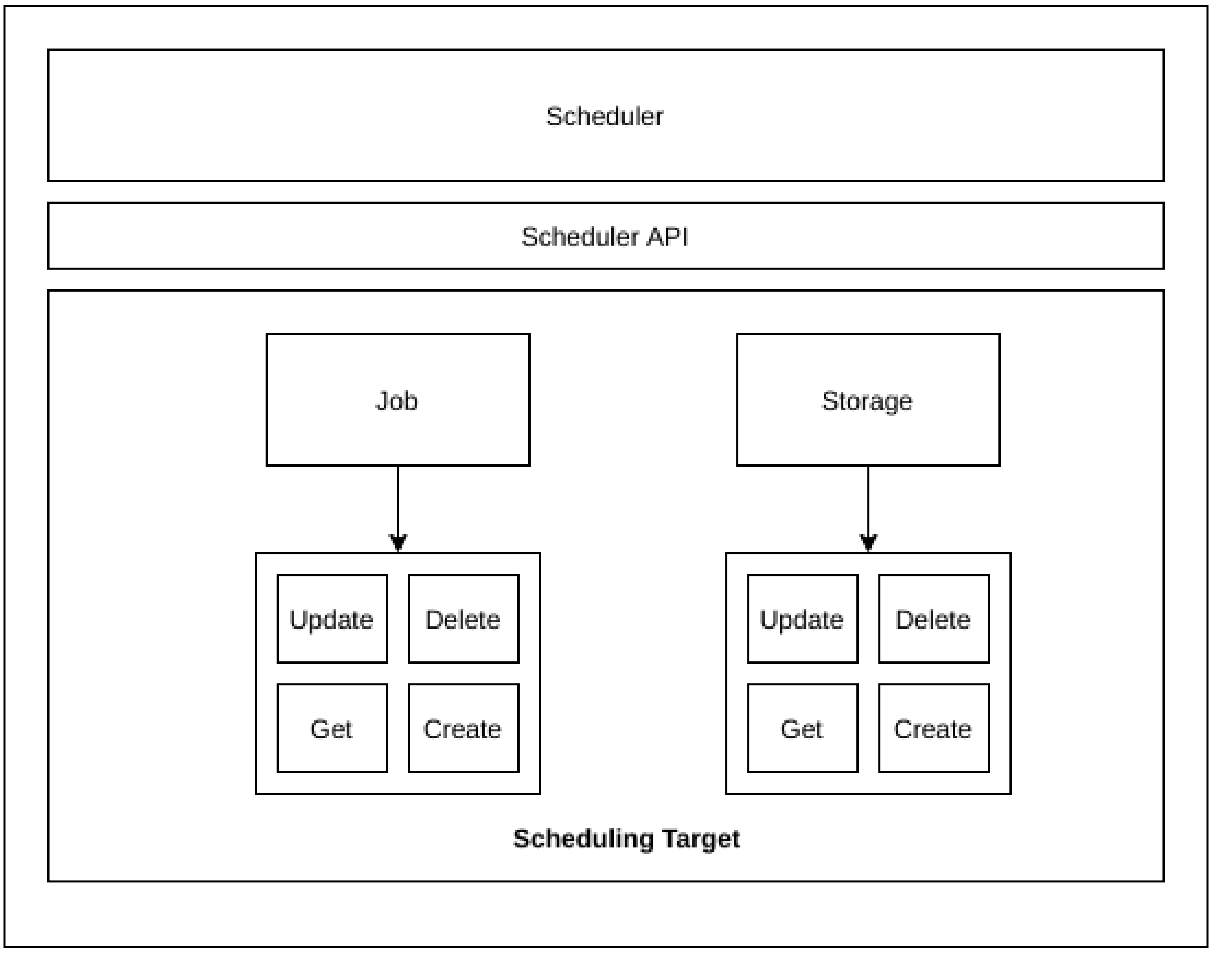


Figure 5.3: Corc Scheduler Components.

5.1.2.1 Configuration

Corc's own configuration is defined as a YAML file that defines the underlying settings of corc. The expected format of this file can be seen in Listing 10. As indicated by the structure of Listing 10, corc defines a set of high level configuration attributes, that each defines attributes for each specific component of corc. Currently, the set of attributes includes, *configuers*, *job*, *providers*, and *storage*. Currently, the *configurers* and *providers* attributes, accept a list of definitions, each for their particular *configurer* or *provider* that corc supports. Likewise with *storage*, corc expects definitions for which storage provider it should use when staging job inputs and outputs. Finally, there is the *job* definition itself, which expects to be told which application it should execute via the supported scheduler. An example of a full configuration file which defines an OCI VM instance infrastructure can be seen in Appendix 21.

```
corc:
  configurers:
  job:
  providers:
  storage:
```

Listing 10: Corc Configuration Structure.

5.1.2.2 Providers

As previously highlighted, corc currently supports two providers for orchestration, namely the OCI and AWS EC2 platforms. In addition, due to the nature of utilizing the Apache libcloud library and the generalized implementation of the Apache Orchestrators within corc, 30 some providers that libcloud supports can subsequently be integrated into corc without much effort. In terms of provider authentication and authorization, corc does not attempt to re-implement these. Instead it expects that the user, defines their personal credentials as expected by the specific provider. Upon usage, corc will instead dynamically load those credentials as per the authentication and authorization conventions of the specific provider. To ensure that an introduced provider, not only is supported in the underlying framework, via the component definitions, such as a provider Orchestrator implementation. The provider should also be integrated into the CLI of corc to provide the full functionality to the user. This is achieved by integrating the introduced provider into the CLI component of the corc framework, here the developer is given the option of specifying which features of corc the provider support, be it the orchestration types, job submission, or any other particular provider features or semantics. Examples of corc CLI usage can be found in [98].

5.2 Cloud enabling applications and computation

When the pool of available resources is increased in an organisation, either through cloud bursting, machine purchase, or virtual expansion, the next question that is posed is often how these additional resources should be employed. Depending on the existing infrastructure, the associated application stack, and the needs of the associated stakeholders, multiple different priorities could be employed when designating which area should benefit from the availability of additional resources. One possibility is to enable existing services that currently are tied in a traditional fashion to the existing resources to exploit the additional capability. However, how this can be achieved depends on many factors, including the design and implementation of the existing services. With corc, one such options is to enable the service to use cloud resources either via a shared Grid of Clouds or at a singular cloud provider. Cloud enabling is made possible in corc via two approaches, either integrating corc into the service/application itself, or utilize the CLI tool to schedule a computational resource with an associated service inside of it. Both of these two approaches were employed in two different scenarios for cloud enabling an existing service and application. The first as shown in Section 5.2.2, was utilized to enable the DAG service presented in 4.1 to spawn interactive Jupyter sessions at an external cloud provider with the integration of corc into a novel JupyterHub Spawner. The second example, as shown in Section 5.2.3, allowed the expansion of an existing neutron scattering simulator and called McStas and its X-ray extension McXtrace.

5.2.1 MultipleSpawner

As explained in Section 4.1, the DAG service at UCPH provides a JupyterHub powered service that allows for scheduling and running JupyterLab user sessions in a container based environment. Although this proved sufficient in providing a small scale teaching and research environment at UCPH, it however, as most installed infrastructures are, limited in both the amount of compute power and hardware it can deliver and of which kind. As shown in Appendix C.1, the DAG service is composed of eight compute nodes, each with a 2 GHz AMD EPYC 7501 processor and 256 GB of memory. Since none of the existing internal nodes have any accelerator capacity, the usefulness of DAG has been limited to quick initial exploration and CPU intensive jobs that does not require massive amounts of parallization to be efficient. The reason being that any violations of the resource restrictions as highlighted in Section 4.1 will result in a termination of the JupyterLab user session. A way to both loosen this restriction and expand the set of available hardware platforms on which the DAG JupyterLab user sessions can be hosted, is by expanding its resource pool to include dynamically allocated cloud resources. The standard method for doing this, would be to discover a JupyterHub Spawner that supports the dynamic scheduling of JupyterLab sessions across both different cloud providers as well as on a local infrastructure. However, at the time of writing, to our knowledge no Spawner with such capability currently exists. There are however, many existing Spawners [139] that allow for spawning session at a dedicated cloud provider. Examples of this includes the Amazon ECS FargateSpawner [10], the Google Dataproc [86] cloud Spawners, followed by a wide variety of local Spawners, including DockerSpawner [140] for Docker containers, SwarmSpawner [140] [94] for Docker Swarm services, KubeSpawner [143] for a local or remote Kubernetes cluster, and YarnSpawner [144] to spawn instances on an Apache Hadoop/YARN cluster.

The WrapSpawner [135] has the closest match for providing dynamic selection of cloud providers to host the JupyterLab user session. With its capability, it is possible to wrap multiple different Spawners to allow the selection of different Spawners, and in extension of this, different cloud providers to host the session from each spawn. Nevertheless, it still falls short of the wanted behaviour. Dynamic management of the supported Spawners for instance is not possible, meaning that once the JupyterHub service has been launched with a preset WrapSpawner configuration, that list of supported Spawners cannot be changed without re-configuring the JupyterHub service. This has the unfortunate consequence that the service requires a restart to activate such as change, forcing the existing users to reconnect to the JupyterHub service. This can be avoid by separating the HTTP Proxy as depicted in Figure 4.1 from the JupyterHub service itself. This will ensure the established connections of the existing users, but does not enable the dynamic change of Spawner without the JupyterHub application being restarted. Furthermore, many of the existing Spawners does not support the orchestration of resources, but rely on them already existing at the designated provider. The FargateSpawner for instance requires that an existing ECS cluster has been created before a JupyterLab instance can be spawned. Equally, the same applies to every local Spawner I know of, including the previously highlighted ones.

I therefore introduced the MultipleSpawner [105] to allow for both dynamic updates and selection of Spawners. Furthermore, the MultipleSpawner provides the added capability of orchestrating resources to host the JupyterLab sessions. With corc, the MultipleSpawner allow for the orchestration at multiple cloud providers, which currently includes the OCI and the AWS' EC2 cloud environments. The overall framework layout of the MultipleSpawner can be seen in Figure 5.4, as shown here, the most important component, is the Spawner, which implements both the *multiple* and *scheduler* sub components. The *multiple* component defines the actual Spawner with support from the additional modules, while the *scheduler* is responsible for interacting with the sub Spawner that schedules the actual JupyterLab session on the provided resource. Controlling the lifetime of the scheduled JupyterLab session, is made possible by delegating this responsibility to the sub Spawner that schedules it. An overview of how this is accomplished can be seen in Figure 5.5, specifically, the MultipleSpawner each of the JupyterHub defined control methods with calling the matching sub Spawner method via the defined *scheduler*. It is therefore expected that the chosen Spawner implements these methods in accordance with the defined JupyterHub API.

In terms of supported resource types, the MultipleSpawner is designed to allow the administrator to define three different types of resources. Namely, Bare-metal, Container, and Virtual Machine. To define how each of these resource types are deployed at the cloud target, the MultipleSpawner expects a Spawner Deployment Configuration file to provide such definitions. An example of the Deployment Configuration can be seen in Listing 12 where the three types are defined with a single configuration. As Listing 12 also indicates, each resource type can define a list of configurations that are available to that resource type. The defined deployment configuration can then be used by the Spawner to specify arguments that should be passed on to the Spawner for deployment.

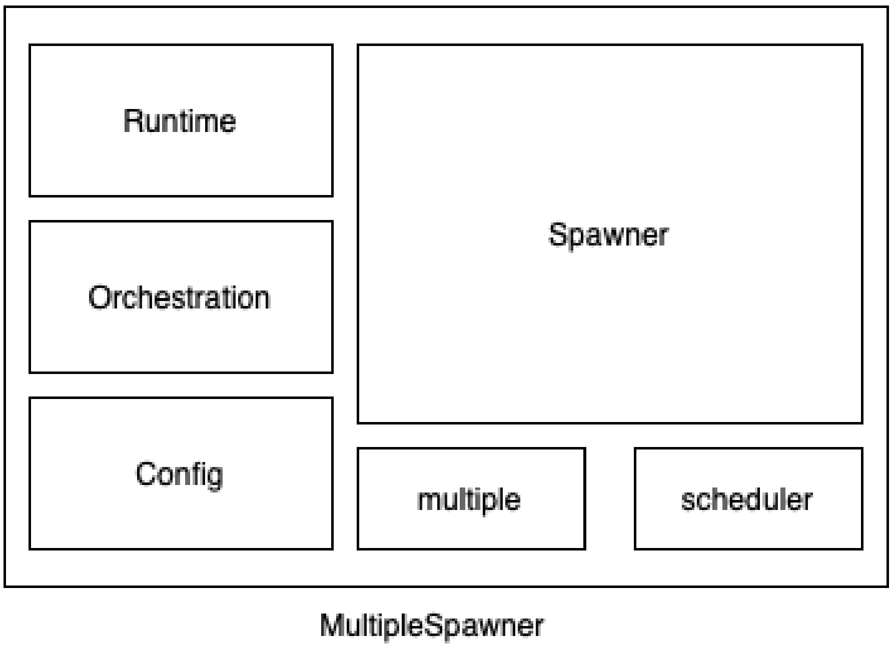


Figure 5.4: MultipleSpawner Framework.

```
{
    "container": [
        {
            "name": "python_notebook",
            "image": "nielsbohr/python-notebook"
        }
    ],
    "virtual_machine": [
        {
            "name": "oracle_linux_7_8",
            "provider": "oci",
            "image": "Oracle Linux 7.8"
        }
    ],
    "bare_metal": [
        {
            "name": "local_machine",
            "provider": "local"
        }
    ]
}
```

Listing 11: Spawner Deployment Configuration.

In addition to the Spawner Deployment Configuration, the MultipleSpawner also expects that the administrator defines a Spawner Template Configuration file. An example of such a file can be seen in Listing 12. The Template Configuration acts as a baseline for how each provided Spawner is configured. Because the Multi-

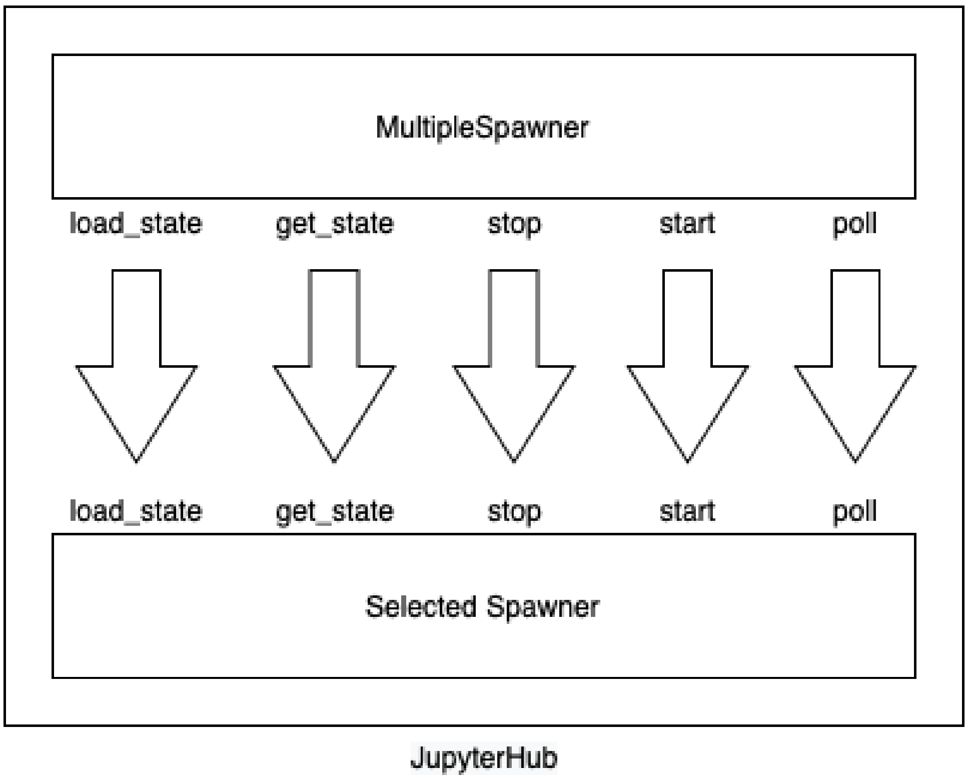


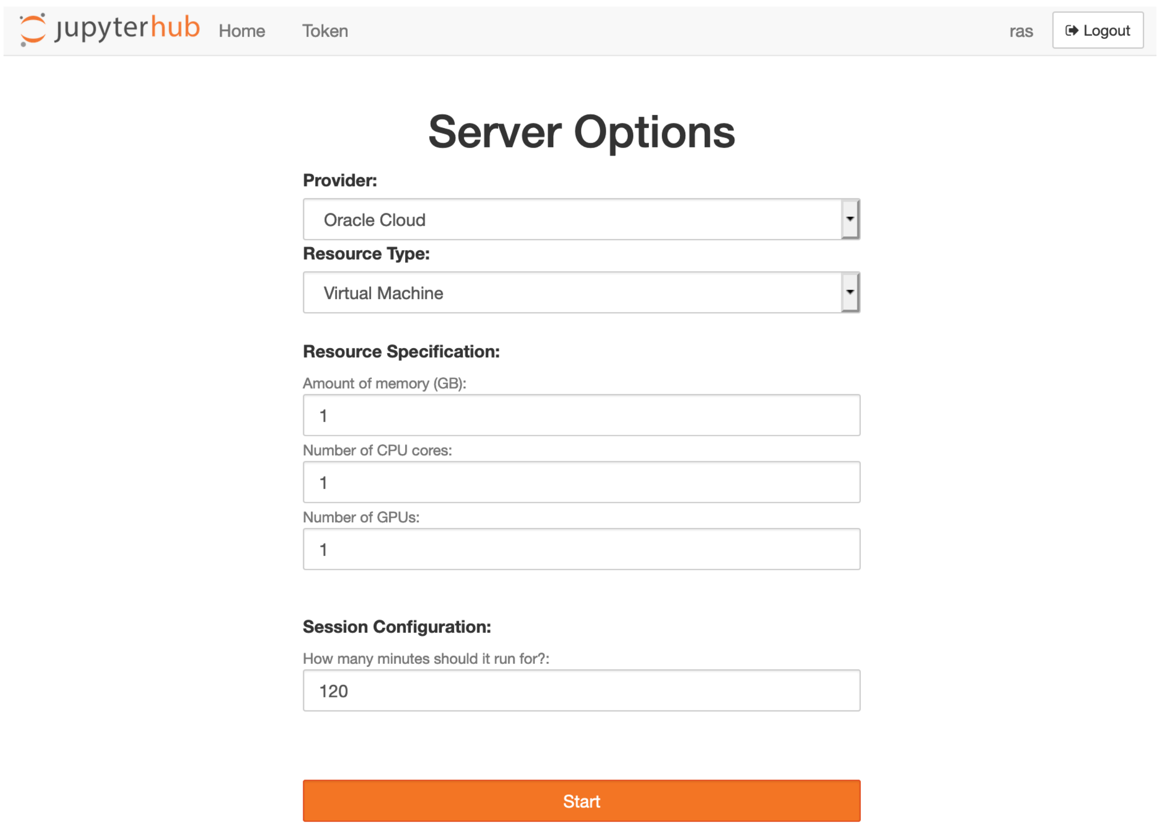
Figure 5.5: MultipleSpawner Wrap Sub Spawner Methods.

pleSpawner supports orchestrating underlying resources via corc, the Template Configuration expects that it be told how that resource should be configured if so required. Furthermore, before the resource can be accessed for either configuring or spawning the JupyterLab user session, the MultipleSpawner also has to be told how it should authenticate against the resource. The configuration and authentication is specified in the Template Configuration via the *configurer* and *authenticator* keys as shown in Listing 12. The expected format for these is a Python class path that is known to the environment in which the JupyterHub is executed. In terms of supported *configurer*, the administrator is free to specify whatever class they wish, but in order to ensure that the specified class is compliant with the API expectations of the MultipleSpawner, it is recommended that the provided corc *configurers* and *authenticators* be used. Currently, corc implements the *corc.configurer.AnsibleConfigurer* and the *corc.authenticator.SSHAuthenticator*. An example of their usage can also be seen in Listing 12.

```
[
    {
        "name": "VirtualMachine Spawner",
        "resource_type": "virtual_machine",
        "providers": ["oci"],
        "spawner": {
            "class": "cloudsshspawner.cloudsshspawner.CloudSSHSpawner",
            "kwargs": {
                # CloudSSHSpawner Kwargs
            }
        },
        "configurer": {
            "class": "corc.configurer.AnsibleConfigurer",
            "options": {
                "host_variables": {
                    "ansible_user": "opc",
                    "ansible_become": "yes",
                    "ansible_become_method": "sudo",
                    "users": [{
                        "name": "{JUPYTERHUB_USER}",
                        "auth_key": "{auth_key}",
                        "sudoer": "yes"
                    }],
                    "jupyterhub": {
                        "server_public_ip": "{server_public_ip}",
                        "server_host_key": "{server_host_key}"
                    }
                },
                "host_settings": {
                    "group": "compute",
                    "port": "22"
                },
                "apply_kwargs": {
                    "playbook_paths": [
                        "~/.multiplespawner/playbooks/prep_environment.yml",
                        "~/.multiplespawner/playbooks/change_user.yml"
                    ]
                }
            }
        },
        "authenticator": {
            "class": "corc.authenticator.SSHAuthenticator",
            "kwargs": {
                "create_certificate": "True",
                "key_name": "{JUPYTERHUB_USER}_id_rsa",
                "load_existing": "True"}
        }
    }
]
```

Listing 12: Spawner Template Configuration.

In terms of the MultipleSpawner's architecture, it can be viewed as sitting on top of the capabilities provided by corc. Figure 5.6 depicts how the MultipleSpawner stack and the underlying corc library are connected.



Server Options

Provider:
Oracle Cloud

Resource Type:
Virtual Machine

Resource Specification:

Amount of memory (GB):
1

Number of CPU cores:
1

Number of GPUs:
1

Session Configuration:

How many minutes should it run for?:
120

Figure 5.6: MultipleSpawner Spawn Page.

The MultipleSpawner has been deployed and tested with the JupyterHub provided LocalProcessSpawner in a local environment at UPCH and with the CloudSSHSpawner [101] at OCI. Therefore it would benefit from additional testing from an incremental roll-out fashion. For instance, deploying it in a private cluster environment inside an organisation like UCPH or MAX IV where the user base could be expanded over time. Furthermore, because the MultipleSpawner relies on a specific sub Spawner to scheduled the requested Notebook, additional testing and benchmarking has to be conducted for each individual Spawner.

5.2.2 Secure communication with resources

Before a JupyterLab session can be spawned, there has to be a method for establishing communication between where the session is scheduled, and where the JupyterHub service is hosted. Usually, this prerequisite has been fulfilled by the underlying infrastructure, where the Spawner simply leverages it. For instance, the KubeSpawner expects that the JupyterHub service is hosted on a node that has access to an existing Kubernetes Cluster, the Kubernetes service is then responsible providing communication between JupyterHub service and the resource on which the JupyterLab session is scheduled by Kubernetes. The same applies to configurations that do not impose a distributed architecture. For instance, the LocalProcessSpawner also is able to successfully manage the JupyterLab session, because it is scheduled directly on the resource that hosts the JupyterHub service. It therefore can be reached directly via operating system services.

The typical scenario implies that either the underlying infrastructure or the selected Spawner provides an adequate method for establishing a connection to the designated resource. However, there exists scenarios where this is not so, especially with the MultipleSpawner, where the designated resources can be adhoc external resources that might not fit the internal architecture stack of the hosting organisation. For instance, when the MultipleSpawner orchestrates a VM at OCI, the resource has to be both reachable and manageable in a secure manner. One approach to accomplish this, is to leverage the existing automatically provided SSH access to the resource, that also allows for the configuration of said resource via the presented *corc.configurer.AnsibleConfigurer*. In terms of feasible Spawners, the SSHSpawner provided by the National Energy Research Scientific Computing Center (NERSC) [109], at first glance seems like a viable candidate. Although at a closer look, it is only a reference implementation, and instead can be a source of inspiration. It is clear from investigating the NERSC SSHSpawner, that it relies on the designated resource, to be hosted within a trusted network, such as inside an HPC Center. This realisation comes from the fact, that subsequent to the JupyterHub SSHSpawner authenticating against said resource with SSH, it relies on the JupyterLab session being instantiated to listen on the external interface of the node.

Furthermore, it expects that the JupyterLab resource can reach the JupyterHub service directly over the network, to establish connections to both the API and activity endpoints. This may be adequate in a closed off environment, such as between nodes in an HPC Center, though one can argue that not even an internal network should be considered secure or trusted. In contrast though, when scheduling resources at a public cloud provider, the communication will always (given you are not the provider itself) happen over an external network that can never be trusted. In such a scenario, the communication between the JupyterHub service and the JupyterLab session has to be protected. One approach to accomplish this, would be to ensure that all traffic between the JupyterHub service and the designated JupyterLab resource travels over an SSH connection. However, since the SSHSpawner was not designed to accomplish this, it was instead used as a foundation for the extended CloudSSHSpawner [101]. The CloudSSHSpawner adds two improvements to the original SSHSpawner, first it creates a remote forward SSH connection [160] to the designated resource before the JupyterLab session is scheduled. This ensure that every request the JupyterLab resource makes against the JupyterHub service travels over the encrypted remote forward connection. Secondly, the CloudSSHSpawner also creates a SSH forward connection from the JupyterHub resource to the JupyterLab resource, thus ensuring that the JupyterHub service is likewise able to communicate with the JupyterLab session over an encrypted connection. An overview of how this is established during the scheduling of a JupyterLab session can be seen in Figure 5.7.

5.2.2.1 Benchmarks

In this section, a simple scenario will be presented and carried out with the MultipleSpawner. As presented in [105], it includes the spawning of a resource type at a designated cloud provider. Specifically a VM with an Nvidia P100 GPU, orchestrated at the OCI. Subsequently to the orchestration, a JupyterLab instance was spawned on said resource. Once spawned, a simple Notebook that utilizes Tensorflow and Keras [119] to build a simple network to classify images was executed and timed before the instance was stopped again via the MultipleSpawner. To compare this performance with DAG, a similar scenario was carried out on said service. The expectation being that since DAG is currently only based on CPU nodes, there would be a significant performance difference. The result these two benchmarks can be seen in Figure 5.8.

As is shown, the MultipleSpawner successfully orchestrated and edspawn a JupyterLab user session at the designated resource type and cloud provider. As expected, Figure 5.8 also shows, there is a substantial difference in the execution time of the two Notebooks. With the OCI GPU version having on average a speedup of 2.8 compared to the DAG CPU version. Furthermore, it is expected, when expanded to additional resource types beyond the orchestration of VMs at the designated cloud provider. It is also expected that there will be a substantial difference in spawn time between the different resource types. The reason for the variance is due to complexity when orchestrating containers vs. VMs vs. bare-metal machines. For instance, when a container cluster has already been orchestrated and configured, the time it takes to schedule a new container is comparable to that of spawning a new process. This however, is of course also true for existing VMs and bare-metal machines, meaning that the orchestration part of the spawning is the most significant part of the time delay from spawning to being redirected to the resource.

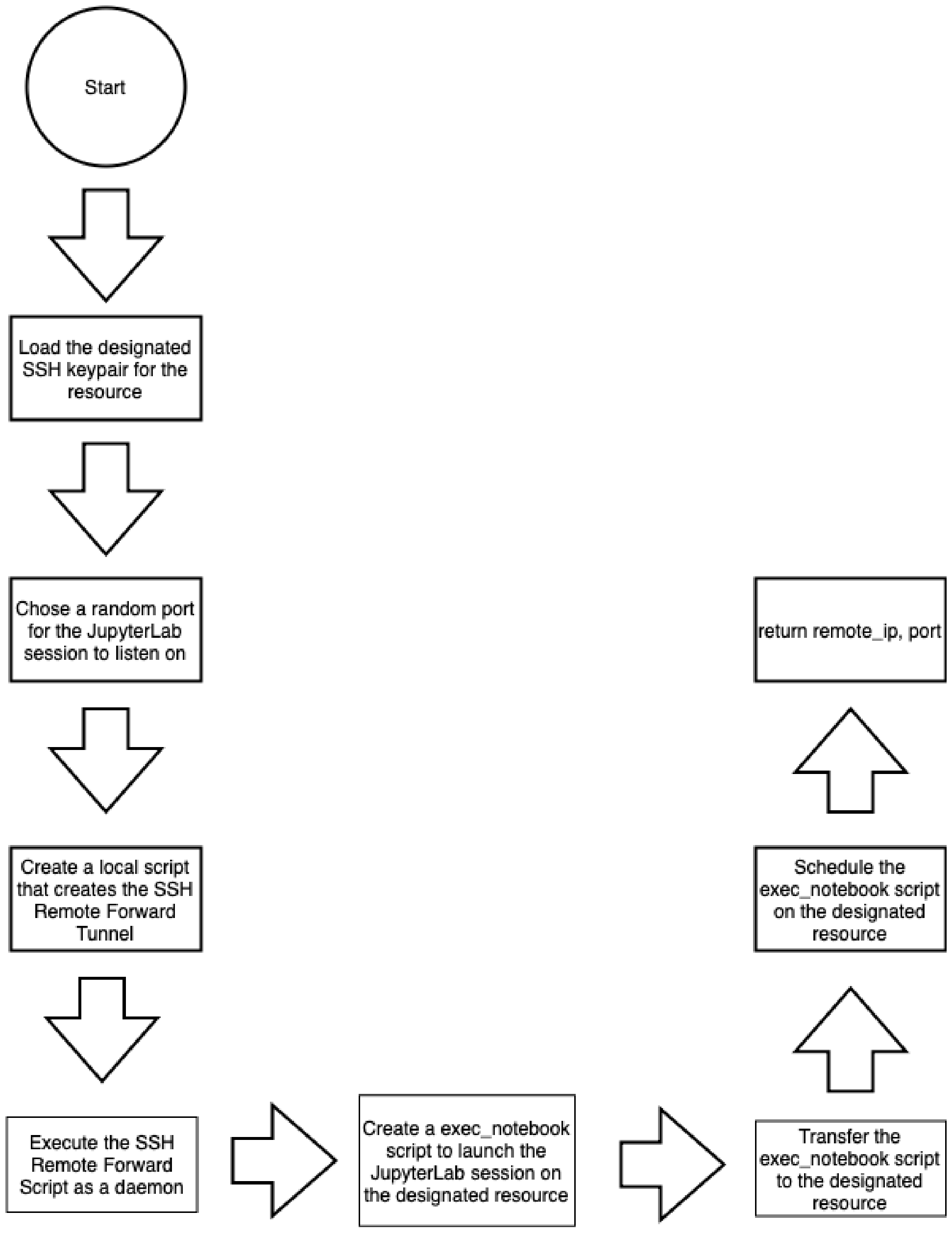

Start
Load the designated SSH keypair for the resource
Chose a random port for the JupyterLab session to listen on
Create a local script that creates the SSH Remote Forward Tunnel
Execute the SSH Remote Forward Script as a daemon
Create a exec_notebook script to launch the JupyterLab session on the designated resource
Transfer the exec_notebook script to the designated resource
Schedule the exec_notebook script on the designated resource
return remote_ip, port

Figure 5.7: CloudSSHSpawner [101] Start Flowchart.

5.2.3 McStas and McXtrace

Another possibility when enabling shared resources across organisational boundaries, is enabling existing applications to utilize cloud resources, to for instance gain access to additional compute capabilities, or simply offloading from existing resources. During this PhD, such a scenario occurred, when a commercial collaborator (Xnovo Technology ApS) of the MUMMERING project wanted to enable their neutron and X-ray ray-trace simulation applications to a commercial cloud provider. The application in question, was McStas [194] and its McXtrace [18] extension. With corc, we were able to accomplish this, but in contrast to the MultipleSpawner, this instead was made possible by utilizing the CLI that corc exposes. Furthermore, in addition to the two simulation tools, the team behind it is also developing a web based platform called McWeb [193] to schedule and monitor simulations using either simulation tool. By utilizing corc in the job submission step of McWeb, it able to allow the scheduling of said job on pre-orchestrated cloud resources. By using the *job* functionality of the corc CLI, the platform was able to schedule simulations to a supported corc scheduler. Because corc only currently

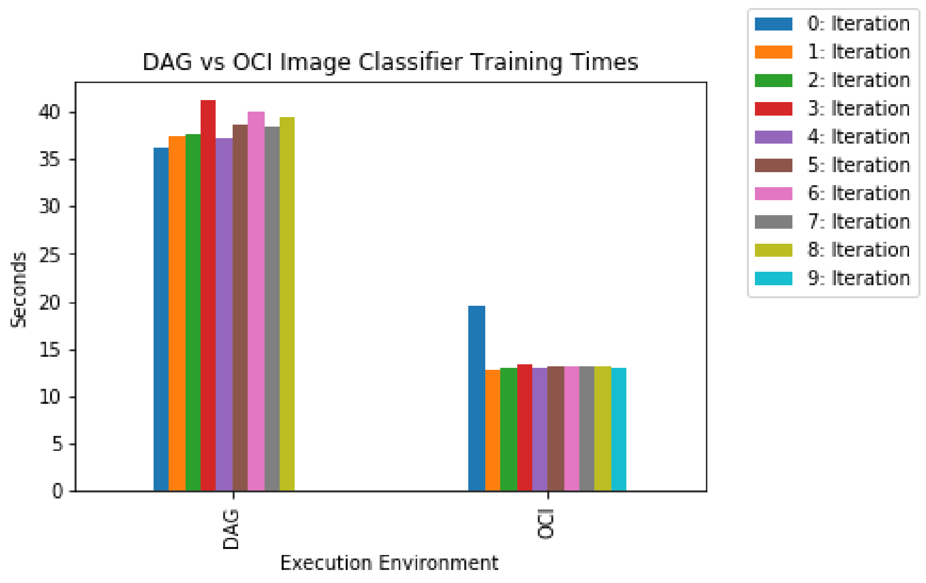


Figure 5.8: Timing data from OCI vs DAG Tensorflow neural network training [105].

supports Kubernetes for job scheduling, the foremost benchmarks were executed as Kubernetes pods. Also, since the job component of corc relies on executing the specific job submission within a container when a containerized scheduler is used. A container image also had to be specified in corc, since this image had to support the execution of either McStas or McXtrace, a specialized container called *nielsbohr/mccode-job-runner:latest* was developed [100], which included both the default corc jobio control package and the mentioned simulation tools.

Furthermore, since McWeb only utilizes corc to schedule the simulations and stage the subsequent inputs and outputs, the Kubernetes cluster itself had to be pre-orchestrated before McWeb can successfully submit the simulations. In addition, as highlighted in Section 5.1, the job part of the corc framework allows for staging of both input and output datasets via storage endpoints. Therefore, at the scheduling of each simulation, the required input data for the McStas or McXtrace simulation, was staged to an S3 endpoint, before being mounted into the subsequent Kubernetes container pod. Similarly, any corc designated output directory, would have its content uploaded to the same S3 endpoint upon simulation completion. To evaluate the effectiveness of empowering the McWeb platform with the ability to submit and execute simulations on a remote cloud resource, a series of benchmarks were conducted.

5.2.3.1 Benchmarks

McStas and McXtrace are Monto Carlo ray tracing simulation tools [18] that aims at modelling the behaviour of neutron and X-ray instrumentation. For instance, a simulation could be defined to model the instrumentation of a potential beamline station at an X-ray facility such as MAX IV. By utilizing modelling, the feasibility of a particular design can be evaluated before construction with the fidelity that the simulation provides [18].

A beamline consists of a long series of devices, from the source of the beam to the detector. Each device along this space, modifies the beam which causes cascading changes for each subsequent device change, which introduces a complexity that, but in a few cases, makes analytical approaches impossible. Instead, the effect of a particular device is modelled with the Monte Carlo ray tracing method. This method provides a viable solution for modelling complex instrumentation chains in simulations with tools such as McStas [18]. In McStas, a large number of numerical photons are traced from the source to the detector, where they are either detected or absorbed [18]. As a particular photon travels through the beamline devices, its parameters are independently changed, as per the interaction between itself and the devices it encounters along the beamline. The independence of the photons becomes a significant factor when computational performance comes into the picture. This is because every photon can be calculated independently; therefore, we can apply a parallel architecture to this task to achieve computational speedup. Thus, to present the potential speedup of utilizing cloud resources, a series of McStas benchmarks were executed on both a scientist laptop and an OCI Kubernetes cluster. The specifications for each of these systems can be seen in Appendix D.2. Also, since both the McStas and McXtrace simulations inherently benefit from parallization, the developers has already enabled the usage of multiple cores on a particular system with the integration of Message Passing Interface (MPI) [171]. Therefore, the scalability, and subsequent speedup of the simulations when deployed to the cloud was verified via an increasing number of applied MPI processes.

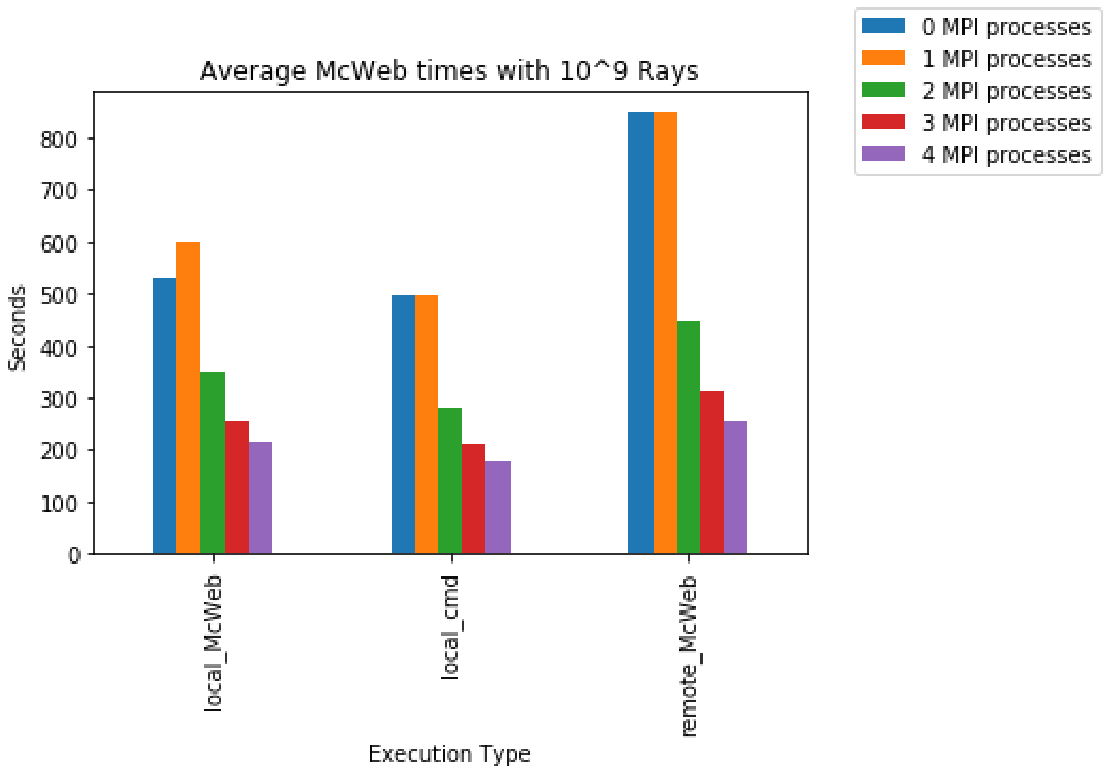


Figure 5.9: McStas Benchmarks, the penalty of local vs remote execution.

The following benchmarks were configured as follows: first two set of times would be captured, namely how long time it took to execute the job, and how long time the simulation itself took. In addition, the McWeb standard test instrument, template-Small-Angle-Neutron-Scattering (SANS) [74] was used for benchmarking how a particular simulation would benefit from the introduced remote execution feature. Secondly, the number of MPI processes applied to the simulation would be scaled from zero to four, as per the available cores in the available systems as shown in Appendix D.2 (PC and OCI Cluster 1). The McStas simulation was executed with 10^9 number of rays, to display the effect of an increasing number of processes on the execution time. The results of these benchmarks can be seen in Figure 5.9.

As shown here, and expected when executing the simulation on a local environment vs. a remote environment with similar specs, there is little difference between them. Furthermore, the local environment is a little bit faster than the remote execution, because it does not have the runtime penalty of having to stage both the inputs and outputs before and after the actual simulation. However, as Figure 5.10 shows, when a remote environment with larger specs (See Appendix D.2 OCI Cluster 2) is the reduction in runtime, from an average of 250 seconds for six MPI processes, to an average of 80 seconds for 24 MPI processes. Thereby highlighting that the execution time for a single simulation is not only faster. Furthermore, by cloud enabling McStas with corc, the

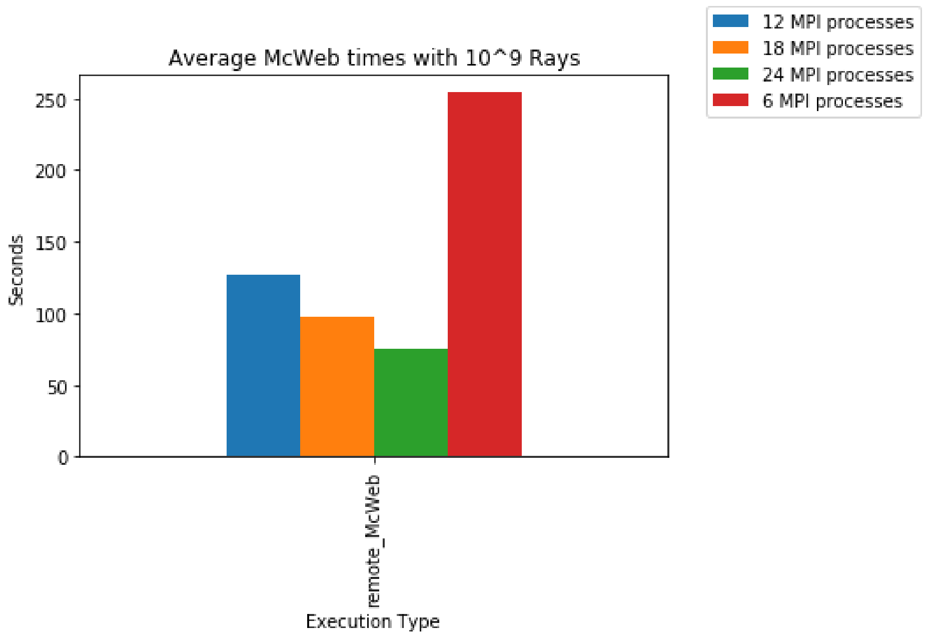


Figure 5.10: McStas Benchmarks, the benefits of cloud execution.

overall system is also able to process more simulations in a given time frame, due to the increase in available compute resources.

5.3 GridCloud

With cloud and resources orchestration in place, the next step in establishing a grid was to design the Grid of Clouds model. This was achieved in tandem while developing the grid_cloud extension, which aims at introducing a decentralized layer on-top of corc. It will achieve this by implementing a light-weight prototype broker that is designed to be in complete control over the domain in which it operates. For now, it is a model based solution, but with further development, it should be a promising model for how a set of individual organisations can interconnect and share resources amongst them. Specifically, by introducing a decentralized broker at each provider, each participating member can commit resources without having to rely on a 3rd party for discovery and selection. Instead, an individual broker will be able to request information from its peers to determine whether the requested resources should be provided internally or externally. A simple model of how I imagine these brokers will be connected can be seen in Figure 5.11. Here the simple network is shown of three participating members, A, B, and C. Each of these have a running grid_cloud broker. Each broker is then responsible for discovering which particular member, be it A, B, or C has available resources fit to serve a particular request.

5.3.1 Design

The grid_cloud broker extension design is such that it will create a P2P layer on top of the regular orchestration. An overview of how of the developing broker is structured can be seen in Figure 5.12. Here the broker cloud expose a common API from which requests for resources can be oriented. For now, the task is limited to providing a CLI extension to corc, that allows for the request of resources from the P2P overlay network. However, since IPv8 defines its own API for enrollment into the network, this could prove as a useful foundation for establishing a common grid_cloud API, for both enrollment and resource management [174].

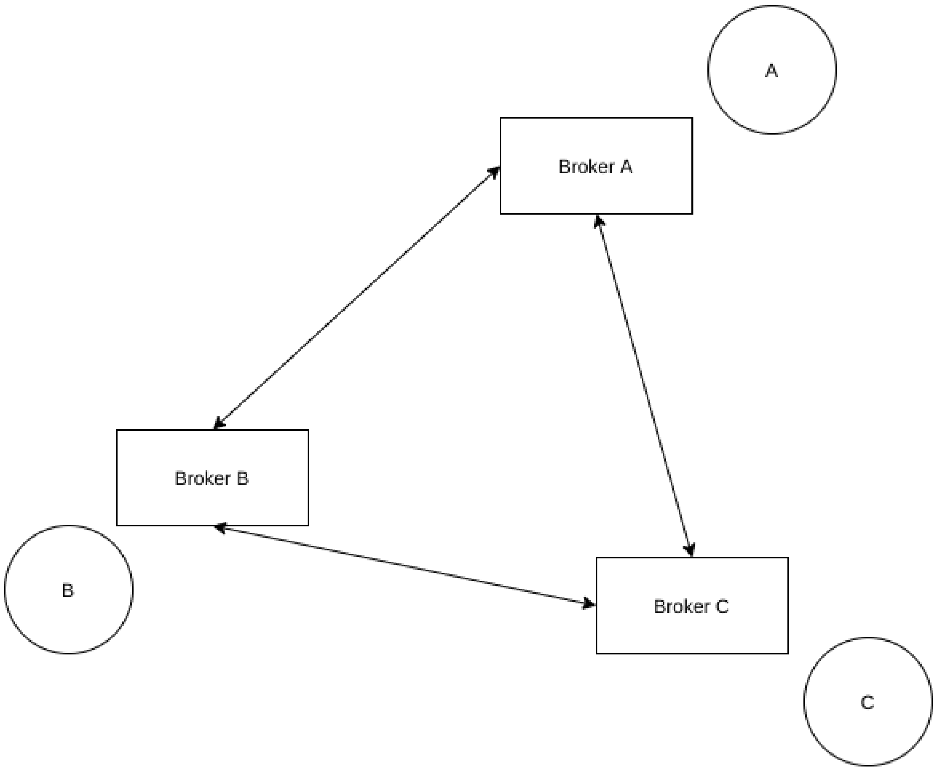


Figure 5.11: Grid of Clouds Network.

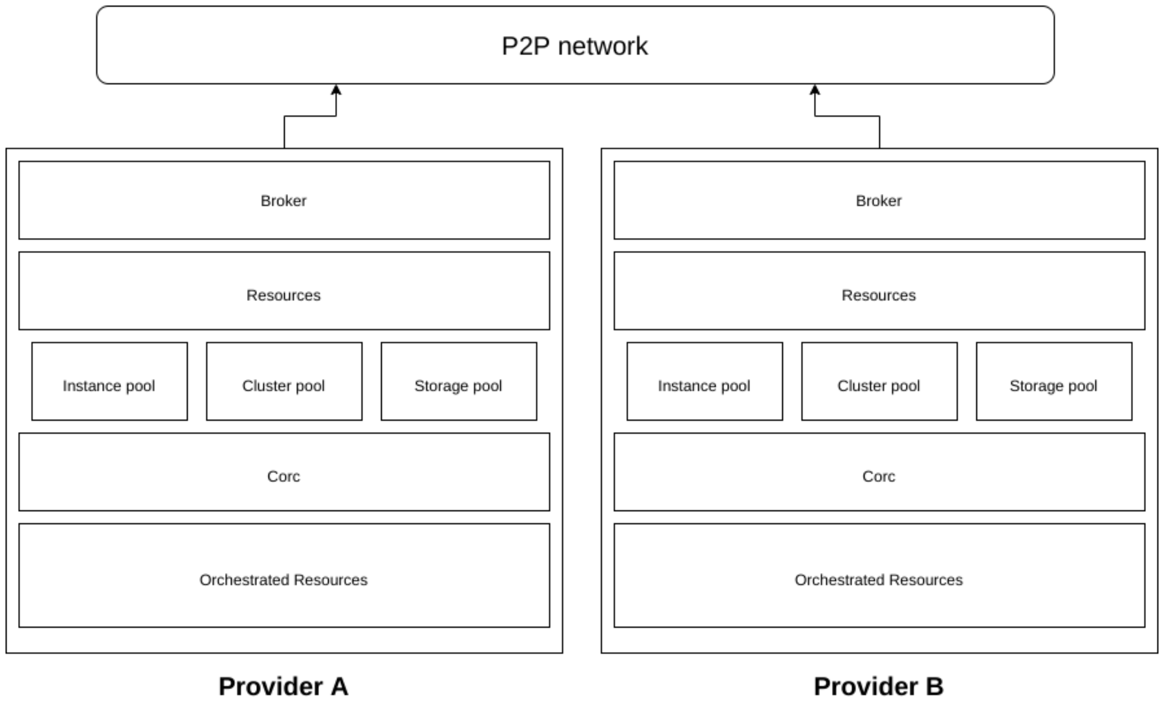


Figure 5.12: Grid of Clouds Architecture.

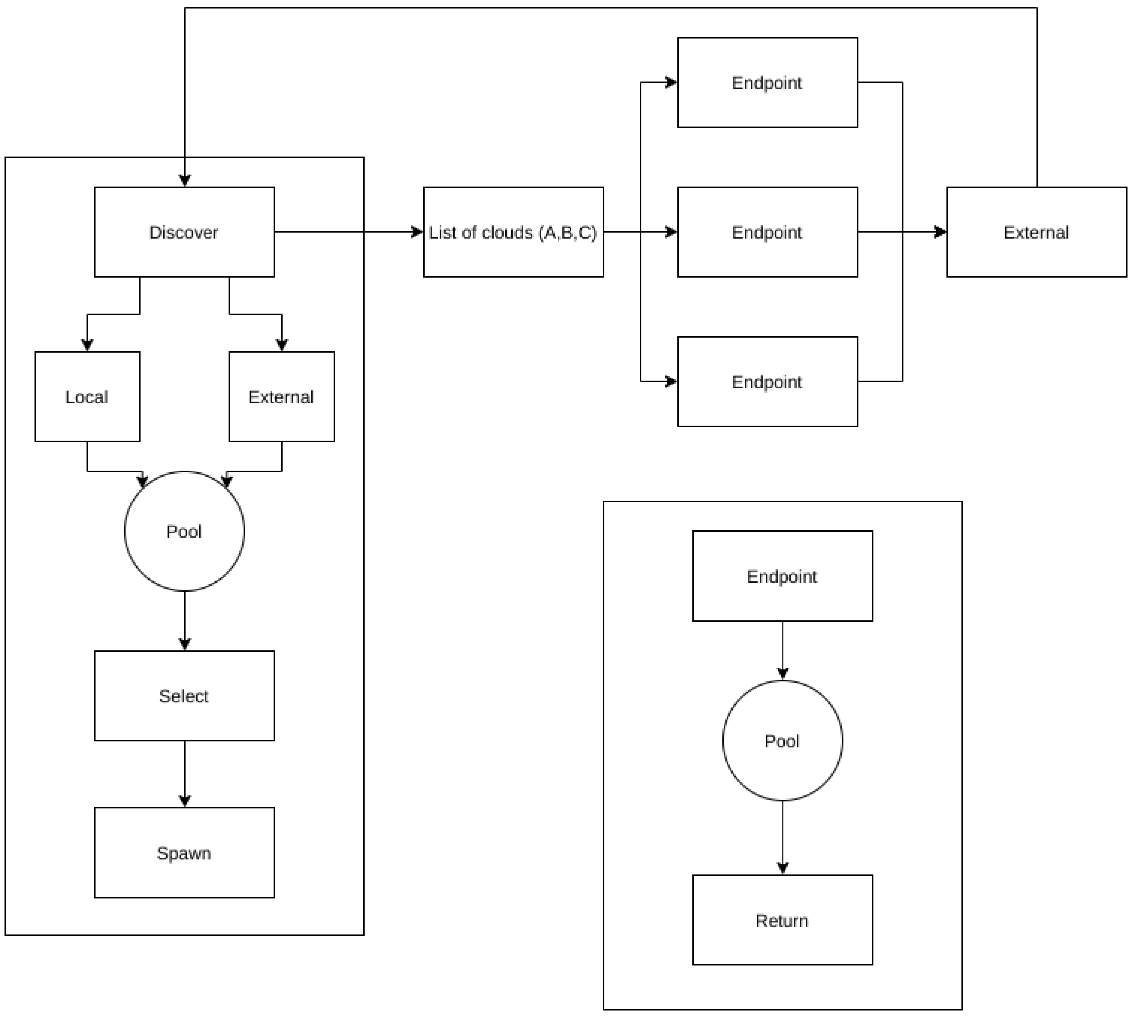


Figure 5.13: Selecting a resource Provider.

5.3.2 Connecting the Grid of Clouds

In terms of establishing connections between the brokers, grid_cloud aims to utilize the IPv8 [173] library to enable a secure interconnected P2P network between the independent instances of the broker. In doing so, it organises these participants into groupings called Communities. To join such a Community, proper trust have to be established with the prospective participants. Trust and mutual authentication and verification in IPv8 is ensured via public key verification, with the initial secret exchange [173]. In doing so, IPv8 is able to establish a decentralized network that grid_cloud can use to messages for resource both resource discovery, selection, in addition to general management requests.

By connecting the participating members as shown in Figure 5.11, each participating member is hosting their own broker. Therefore, it is able to communicate with the other participants via the IPv8 overlay network. Because the grid_cloud is designed to operate on top of corc, it is therefore intended to be inherently able to orchestrate resources at a designated provider. This, of course, means, that if the organisation hosts an internal cloud, this would have to be incorporated into corc before grid_clouds can make use of it. As part of the design, the intended process for selecting a cloud provider can be seen in Figure 5.13. Specifically, that the broker is foremost tasked with discovering a suitable candidate for the particular request. Secondly, that it does a continuous lookup of potential resource providers, the discovery here draws inspiration for how data is distributed in S3. Namely, that the network is made up of a set of broker entities, which are placed in a distributed hash table [198], that ensures that each member is distributed across a figurative circle, which in turn maps each member its supported set of resources. Each member, therefore holds information about every other member, making it possible to do direct lookups for a particular request.

5.3.3 Orchestrating Resources

When orchestrating resources in a network of providers, as highlighted in Section 2.2.1.2, the task of establishing and managing computational resources has typically fallen to the network participant. Namely that each resource that had to be part of the network, has to be provision, configured and maintained by the local administrative body. However, because of the reliance on corc and the added extension, the local administrative body would instead be responsible for configuring for joining the overall network and which particular cloud provider that should be utilized for resources. Thereby alleviating the burden of conducting this task for every member resource. Nevertheless, this does not absolve them from the task of maintaining the resources over time. Because as highlighted, corc in its current incarnation, does not guarantee that an applied configuration state is maintained over time.

In terms of maintaining the state of the orchestrated resource in the current model, the grid_cloud is designated to be responsible for this. In particular, each broker is to maintains their own state, thereby alleviating the requirement for ensuring a consistency across the individual brokers. Instead, the broker is only responsible for maintain its own set of orchestrated resources, as illustrated in the presented framework design in Figure 5.12. Here, the Resources layer is an abstraction layer on top of the set of resource pools that are currently maintained by the individual broker. The Broker layer is subsequently responsible for exchanging and receiving messages from the P2P network, that is, other brokers, in addition to receiving requests from a prospective user.

5.3.4 Grid of Clouds

By implementing the Grid of Clouds model, the prospective organisational members will be able to share resources amongst each other. In particular, this would be beneficial to time limited projects and ephemeral collaborations that do not necessitate the burden of joining a full scale Grid like EGI. Instead, with the continued development of grid_cloud in tandem with the underlying corc, such projects will gain significant improvement for the individual scientists. Because, with the ability to establish such sharing at a local level, their research can gain access to an increased number of resources and could foster further collaboration if such sharing is bidirectional. Furthermore, it would enable the usage of organisational resources to be employed in existing applications that could be empowered with cloud capabilities, as shown with the MultipleSpawner and McStas applications. However, further research and development in this area is necessary before such fruits can be picked, but it is definitely a worthwhile endeavour.

5.4 Summary

In this Chapter, the work conducted by me surrounding Cloud and Grid solutions was presented. Foremost, the Cloud Orchestrator (corc) library was introduced; it is a tool that can be used to orchestrate resources at supported cloud providers. Furthermore, it is able to schedule job at said provider, given that a supported resource for scheduling has been orchestrated. Additionally, it enables the user to stage both the inputs and outputs of these scheduled jobs. Secondly, it was illustrated how corc can and has been used, to cloud-enable both novel and existing applications such as the MultipleSpawner and the McStas neutron ray-tracing simulator, meaning that they were able to utilize cloud resources as part of delivering their purpose.

In addition, the novel MultipleSpawner was introduced. The Spawner allows for the dynamic selection and adaption of cloud providers and configurations in a running JupyterHub service. Furthermore, it enables the Spawner itself to orchestrate resources by cloud enabling the Spawner with the corc library. Furthermore, this Section introduced the augmented SSHSpawner [109], named CloudSSHSpawner [101], that allows for dynamically establishing secure SSH tunnnels both to and from an external resource that hosts a JupyterLab user session. I also showed that the MultipleSpawner was able to spawn a GPU based instance with a with JupyterLab and Tensorflow incoporated. As shown in the benchmarking section, the MultipleSpawner was successful in leveraging the integrated corc library to schedule the JupyterLab user session at the OCI cloud provider, namely the OCI. In terms of future developments, the MultipleSpawner is still in an early stage of development and should be exposed to extensive testing in larger real world scenarios with hundreds of users and additional cloud providers. A roll-out at an institution like UCPH or MAX IV could provide the necessary facilities and requirements to conduct such testing.

Finally, the Grid of Clouds model was introduced, including the developing grid_cloud library. It aims to provide the ability to orchestrate and schedule resources dynamically across individual CSPs via a P2P network. As a prospective mechanism, the design suggests an IPv8 overlay network to establish a decentralized network amongst the set of Grid participants. Furthermore, by utilizing corc, it will be able to orchestrate resource across its set of supported CSPs. By establishing this network, organisations will be able to share resources amongst each other via the creation of small and ephemeral networks that enable such exchange.

Chapter 6

Future Work

The various projects, including Mig Utils, HISS, DAG/MODI, Cloud Orchestrator, MultipleSpawner and GridClouds are by no means finished or perfect in their current state. In this Chapter I will explain how each of these could be further developed, enhanced, or adjusted to provide additional benefit.

6.1 MiG Utils

Even though the Mig Utils library has the basic functionality to allow the staging of inputs and outputs as part of a job execution, tt still could benefit from several enhancements. For instance, at the present date, loading and storing of data only support synchronous calls. Therefore, the introduction of asynchronous capabilities could in certain cases speedup the runtime of jobs via techniques such as latency hiding. Furthermore, currently the conversion of the data streams is achieved by hand. Therefore, it would be beneficial if the MiG Utils provided methods to, load a dataset as a NumPy array. In addition, in the future, MiG Utils could be envisioned to be a prime candidate for loading and storing data to the HISS when the requirement for high throughput is present. Also, because the MiG Utils library loads the input data every time it is executed, it does lead to a substantial amount of downloads for the same particular dataset, if an analysis is executed multiple times. Therefore, it would be of great benefit to implement a caching mechanism, that temporarily stores the dataset on the running resource between executions. To validate that the data has not changed since the last caching, a hash could be calculated and compared, to ensure that they are exactly the same.

6.2 HIgh Throughput Storage System

Since the HISS is currently only at the design and basic prototype stage, substantial work still needs to be put into this project, including additional development. To fully evaluate such a system, specialized hardware such as the presented options in Section 3.4 is required. Specifically, FPGA platforms that would enable the translation of NumPy/Bohrium code into VHDL which could be synthesised onto hardware. In the mean time, the general architecture around the kernel translations could be developed more, but this could prove wasteful if the key element of applying synthesised kernels in-situ is not achievable. However, if feasible, the Go prototype should be developed further to be able to attach computational kernels to each part storage hierarchy. In addition, because Go uses a garbage collector to unallocate unused memory, the language might not scale well with the expected quantities of data. Future benchmarks will determine whether it is necessary to move to a different memory management model that do not rely on a process to detect and cleanup memory. For instance RUST [192] uses a reference counting model, that unallocates memory as soon as a particular object counter reaches zero. This is provided without having to manually allocate or unallocate memory as is the case in C [189].

6.3 Data Analysis Gateway

Currently DAG is a running service at UCPH, and a variant of it is deployed at MAX IV. DAG has proved itself a useful software stack for providing interactive data analysis at an educational and scientific institution. At this point, DAG though benefit from some general quality-of-life improvements, this includes an automatic split between the JupyterHub service itself and the HTTP Proxy as depicted in Figure 4.1. This split will ensure that DAG is able to apply changes without interrupting the current users by default. Furthermore, with the integration of the MultipleSpawner into DAG, the service will be able to provide environments from multiple providers with different Spawners. This would also enable DAG to be utilized with different infrastructure than the current that relies on Docker Swarm and the SwarmSpawner. In doing so, it could suitably be used across additional infrastructure configurations. Another interesting addition, would be to enable DAG administrators to define permissions for specific environments on who should be able to use it and for how long. For instance, when deploying container images, it could be defined such that only a certain set of users would have access to it, and for a limited amount of time. This would add a granularity to the service, which will help serve different needs. For instance, students typically have a short and ephemeral environment need, while scientists require long and persistent sessions to conduct large experiments.

6.4 MPI Oriented Development and Investigation

In the future, similar to DAG, MODI would benefit from general administrative life improvements, which includes the points covered in Section 6.3. In addition, because MODI currently relies on interactions with the backend cluster via shell commands, an improvement to the structure such that these do not have to take place in the specific *~/modi_mount* directory would clear up the typical initial confusion for users. Because the Jupyter instances in MODI are not the intended place for computations to be carried out, an interface integration with the SLURM scheduler would be beneficial. Specifically, enabling the ability for submitting a particular .ipynb Jupyter Notebook as a job to the backend cluster. Developments for this has already been started, with the NERSC SLURM JupyterLab extension [110]. However, this extension is limited in that it is not continuously supported, nor does it currently provide great feedback for how a particular job is doing.

6.5 Cloud Orchestrator

As it currently stands, corc is a useful tool for orchestrating cloud resources at OCI and AWS EC2, in addition to providing job scheduling at a designated Kubernetes cluster and general data staging capabilities. Because of this, corc would benefit from the inclusion of additional schedulers. To this extend, the Schedulers integration with the CLI will have to be refactored such that it fits with the current dynamic structure used for the Orchestrator. Doing so will enable the dynamic extension of multiple Schedulers. For instance, a scheduler of interest would be the inclusion of a DASK [154] oriented scheduler, in addition to being able to orchestrate such a cluster, would enable users to spawn ephemeral compute clusters that support advanced parallelism jobs with intercommunication and not just the current batch-oriented types. Furthermore, additional work should be put into making corc easy to get started with. This could include a CLI setup tutorial that provides an interactive process to setup a particular provider. This, would alleviate the initial huddle of defining the external provider configurations that corc relies on, when for instance authenticating against a particular provider. Also, another aspect would be the ability to define multiple instances in the configuration file; currently it is only able to accept a single instance or cluster. Although orchestrating multiple instances or clusters is still possible via the CLI; it should also be available via the underlying configuration. With such improvements, corc will be both simpler to use for an individual user to orchestrate complex external infrastructures and scheduling both batch and data oriented jobs.

6.6 MultipleSpawner

The current MultipleSpawner can use the improved SSHSpawner [101] to orchestrate resources and schedule JupyterLab sessions at an external cloud provider. Since it has only been tested in a prototype and development environment, it would benefit from a stress test in a larger and more complex scenario. For instance, in the future it should be integrated to DAG and utilized at UCPH to examine how it operates when multiple users are utilizing it. Furthermore, because it allows for the usage of basically every implemented Spawner, it does not mean that these are inherently supported without modifications. This is because the Spawners themselves bring certain expectations to the underlying infrastructure. To account for this, the MultipleSpawner should be tested with additional Spawners such as the KubeSpawner to validate their compatibility.

6.7 GridClouds

The GridClouds extension is by no means finished; therefore the grid_cloud prototype should see significant additional development. In particular, the inclusion of non public CSPs in corc, to extend its capability to utilize typical internal cloud infrastructures such as OpenStack. In addition, grid_cloud requires substantial further development and testing of the IPv8 overlay network, to establish the P2P network amongst the Grid participants. In turn, substantial stress testing of how the extension will function under load with multiple participants. In general, the Grid of Clouds framework should be suitable platform for establishing a loosely coupled set of decentralized brokers that are able to achieve cross organisational resource orchestration and subsequent service or job scheduling.

Chapter 7

Conclusions

In this thesis, I have been involved with quite a number of topics in relation to how to enable scientists to use computational resources to conduct research. For instance, how data should be staged and extracted when executing a particular analysis on an external resource. The MiG Utils data sharing and staging library show how the loading and storage of data for a particular analysis can be defined as part of a particular implementation. It allows the scientists to define the data storage on which their particular dataset is available, in addition to defining where the results should be stored. In coupling the loading and storing of input and output data to the analysis, it ensures that it can be ubiquitously executed on any given resource that is able to reach the designated data storage. Currently, the MiG Utils library provides a set of helper classes that supports the UCPH ERDA and IDMC storage services. This however can be easily extended to include additional storage providers, as long as they support either SFTP or SSHFS. Furthermore, as the benchmarks shows, the not surprising fact that the execution time of a particular analysis, can be greatly impacted by the available bandwidth of the executing resources. Nevertheless, the MiG Utils does enable the scientist to develop their analysis, oblivious to how a particular computational platform is other-wised designed to stage inputs and outputs. Given of course that they allow the external access to the designated data storage. In addition to the MiG Utils library, the HISS model was presented, including how such a model could benefit from high throughput producers such as MAX IV or EuXFEL by applying simple prepossessing tasks during the acquisition phase in-situ before the data is stored on a parallel file system.

Beyond data handling, I also investigated and developed a number of computational platforms to serve as tools in a scientific setting. Specifically, the DAG and MODI services with their complementary projects provides two bundled platforms for data analysis. This platforms allow educational institutions like UCPH or scientific instruments to provide interactive data analysis platforms. As part of the bundled stack, DAG and MODI allows for the automatic integration of external storage providers via the developed HeaderAuthenticator and extension of the SSHFS volume plugin. Furthermore, with the ldap_hooks library, it was possible to integrate the interactive environment of MODI with a classic LDAP based SLURM cluster. Beyond the establishment of computational platforms, the collaboration of how jobs should be organised and scheduled was investigated. This resulted in the establishment of MEOW that, in contrast to static workflows, are inherently dynamic, where workflows are an emergent property that will exhibit themselves after the individual steps have been executed. Another result of this work was the Notebook Parameterizer, that in contrast to Papermill, allows for dynamic overload of parameters throughout a Notebook and not just a designated part of it. This enabled the MEOW to provide proper paramterization, without requiring users to predefine in their analysis which variables that could potentially be overwritten during the workflow execution.

As the final piece of this thesis, I investigated how cloud computing is and could be used in the area of science. This work was inspired by the collaboration with Xnovo Technology and Oracle, on how such computational resources could be made available to scientists, educational institutions, and existing applications or services. The result of which was the corc tool/framework, that allows for orchestration and scheduling of computational jobs at a designated CSP via either its CLI or by using its framework. Furthermore, it allows for the integration with novel or existing applications. It was used to cloud-enable the novel MulitpleSpawner and the existing neutron ray-tracing simulator to utilize cloud resources as part of their execution. However, corc is limited in that it is still a developing tool. For instance, it currently only supports two cloud providers for orchestration, namely OCI and AWS EC2. To extend this should be relatively easy extendable by its usage of the Apache libcloud library which supports 30 somewhat providers. It is implemented in Python 3, which makes restricts the API of the framework to only be usable in such projects. For instance, McStas in its Python 2 implementation was instead cloud enabled by utilizing corc's CLI interface. In addition, the last Section of the thesis presents how the integration of corc into the grid_cloud extension could provide a means to interconnect different organisations, in a Grid of Clouds. Specifically, by establishing a decentralized broker that is solely responsible for discovering and selecting where a particular resource request should be served. In turn, they can be interconnected by utilizing IPv8 overlay networks, which provides advanced networking features such as a decentralized network, which is critical for the broker to work. By establishing such an interconnected Grid, organisations will be able to share surplus or specialized resources dynamically amongst each other in ephemeral projects that do not validate the establishment of a permanent Grid. Thereby potentially furthering the scientific advances in smaller and temporary cross organisational projects such as MUMMERING.

Bibliography

[1] BinderFAQ, 2017.

[2] Mummering. http://www.mummering.eu, 2017.

[3] Google Colab, 2020.

[4] Williams D. N.; A, Ananthakrishnan R.;, Bernholdt D. E.;, Bharathi S.; D.;, Brown, Chen M.; Shoshan, Chervenak A. L.; R.;, Cinquini L.; Drach R.;, Foster I. T.;, Fox P.;, Fraser D.; Garcia J., ; Hankin S.; Jones P.;, Middleton D. E.;, Schwidder J. Schweitzer, ;, and R.; Schuler. The Earth System Grid. *Bulletin of the American Meteorological Society*, (February):195–206, 2009.

[5] Giovanni Aloisio, Massimo Cafaro, Roy Williams, and Paul Messina. A distributed web-based metacomputing environment. pages 480–486. 1997.

[6] Altair Engineering Inc. OpenPBS, 2021.

[7] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludäscher, and Steve Mock. Kepler: An extensible system for design and execution of scientific workflows. *Proceedings of the International Conference on Scientific and Statistical Database Management, SSDBM*, 16:423–424, 2004.

[8] Amazon. Amazon Web Services, 2020.

[9] Amazon Web Services. RunInstances, 2020.

[10] Amazon Web Services. FargateSpawner, 2021.

[11] Anaconda Inc. Anaconda, 2020.

[12] Ansible. Ansible, 2020.

[13] Ansible. Ansible default Connections Methods, 2020.

[14] Ansible. Intro to playbooks, 2020.

[15] Barcelona Supercomputing Center. Barcelona Supercomputing Center, 2021.

[16] Jonas Bardino, Martin Rehr, and Brian Vinter. Event-driven, Collaborative and Adaptive Scientific Workflows on the Grid. *Communicating Process Architecture*, pages 59–78, 2017.

[17] Erick Bauman, Gbadebo Ayoade, and Zhiqiang Lin. A survey on hypervisor-based monitoring: Approaches, applications, and evolutions. *ACM Computing Surveys*, 48(1), 2015.

[18] Erik Bergbäck Knudsen, Andrea Prodi, Jana Baltser, Maria Thomsen, Peter Kjær Willendrup, Manuel Sanchez Del Rio, Claudio Ferrero, Emmanuel Farhi, Kristoffer Haldrup, Anette Vickery, Robert Feidenhans'l, Kell Mortensen, Martin Meedom Nielsen, Henning Friis Poulsen, Søren Schmidt, and Kim Lefmann. McXtrace: A Monte Carlo software package for simulating X-ray optics, beamlines and experiments. *Journal of Applied Crystallography*, 46(3):679–696, 2013.

[19] Jost Berthold, Jonas Bardino, and Brian Vinter. A Principled Approach to Grid Middleware. In *Algorithms and Architectures for Parallel Processing*, volume 7016, pages 409–418. Springer, 2011.

[20] Jost Berthold, Jonas Bardino, and Brian Vinter. A Principled Approach to Grid Middleware: Status Report on the Minimum Intrusion Grid. In Yang Xiang, Alfredo Cuzzocrea, Michael Hobbs, and Wanlei Zhou, editors, *Algorithms and Architectures for Parallel Processing*, pages 409–418. Springer, 2011.

[21] Benjamin Black. EC2 Origins, 2009.

[22] Elise O. Brenne. Tomography workflow, 2019.

[23] Sean Carlin and Kevin Curran. Cloud Computing Technologies. *International Journal of Cloud Computing and Services Science (IJ-CLOSER)*, 1(2), 2012.

[24] CERN. The Large Hadron Collider, 2020.

[25] CERN. The Worldwide LHC Computing Grid, 2020.

[26] Tadeu Classe, Regina Braga, José Maria N. David, Fernanda Campos, and Wagner Arbex. A Distributed Infrastructure to Support Scientific Experiments. *Journal of Grid Computing*, 15(4):475–500, 2017.

[27] Cloud Industry. Cloud Computing Models Demystified, 2020.

[28] CoCalc. Cocalc Docs, 2020.

[29] Phyllis E. Crandall, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed. Input/output characteristics of scalable parallel applications. *Proceedings of the ACM/IEEE Supercomputing Conference*, 2:1643–1673, 1995.

[30] Cwaldbieser. jhub_remote_user_authenticator, 2015.

[31] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Proceedings of 6th Symposium on Operating Systems Design and Implementation*, pages 137–149, 2004.

[32] Debian. Debootstrap.

[33] Desy. Desy Compute Center, 2020.

[34] DIET: The Grid and Cloud Middleware. DIET, 2020.

[35] H Digabel and C Lantuéjoul. Iterative algorithms. In R Verlag, editor, *Actes du Second Symposium Europeen d'Analyse Quantitative des Microstructures en Sciences des Materiaux, Biologie et Medecine*, pages 85–99, 10 1978.

[36] Docker Inc. Deploy services to a Swarm, 2019.

[37] Docker Inc. Docker, 2020.

[38] Docker Inc. Docker Image Specification, 2020.

[39] Dropbox. Dropbox Sharelink, 2018.

[40] EGI. The European Grid Infrastructure, 2020.

[41] M. Ellert, M. Grønager, A. Konstantinov, B. Kónya, J. Lindemann, I. Livenson, J. L. Nielsen, M. Niinimäki, O. Smirnova, and A. Wäänänen. Advanced Resource Connector middleware for lightweight computational Grids. *Future Generation Computer Systems*, 23(2):219–240, 2007.

[42] ESRF. European Synchrotron Radiation Facility, 2020.

[43] European Grid Infrastructure. EGI FedCloud, 2020.

[44] European XFEL. European XFEL, 2020.

[45] Stefano Falcinelli, Andrea Capriccioli, Fernando Pirani, Franco Vecchiocattivi, Stefano Stranges, Carles Martì, Andrea Nicoziani, Emanuele Topini, and Antonio Laganà. Methane production by CO2 hydrogenation reaction with and without solid phase catalysis. *Fuel*, 209(July):802–811, 2017.

[46] Luis Ferreira, Viktors Berstis, Jonathan Armstrong, Mike Kendzierski, Andreas Neukoetter, Richard Bing-wo, Adeeb Amir, Ryo Murakawa, Olegario Hernandez, James Magowan, and Norbert Bieberstein. Introduction to Grid Computing with Globus. *Redbooks*, page 268, 2003.

[47] Ian Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. 2006.

[48] Ian Foster. What is the grid? a three point checklist, {July} 2002. *ThreePoint-Check. pdf*, (January 2002), 2006.

[49] Ian Foster and Carl Kesselman. The history of the grid. *Advances in Parallel Computing*, 20:3–30, 2011.

[50] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

[51] Gartner. Gartner Says Worldwide IaaS Public Cloud Services Market Grew 37,3 % in 2019, 2020.

[52] Ben Glick and Jens Mache. Jupyter Notebooks and User-Friendly HPC Access.

[53] Globus. Globus Toolkit, 2020.

[54] Google. Google Colab FAQ, 2020.

[55] Google Inc. Google Drive Sharing, 2020.

[56] Dion Häfner, René Løwe Jacobsen, Carsten Eden, Mads R. B. Kristensen, Markus Jochum, Roman Nuterman, and Brian Vinter. Veros v0.1 &amp;ndash; a Fast and Versatile Ocean Simulator in Pure Python. *Geoscientific Model Development Discussions*, pages 1–22, 2018.

[57] HashiCorp. TerraForm Providers, 2020.

[58] Michael A Herzog, Zuzana Kubincová, and Gerhard Goos. *Advances in Web-Based Learning – ICWL 2019*. 2019.

[59] Jiangshui Hong, Thomas Dreibholz, Joseph Adam Schenkel, and Jiaxi Alessia Hu. An Overview of Multi-cloud Computing An Overview of Multi-Cloud Computing. (April 2020), 2019.

[60] HTCondor. HTCondor, 2021.

[61] IEEE Spectrum. The Top Programming Languages, 2021.

[62] INDIGO-DataCloud. INDIGO-DataCloud, 2020.

[63] JetBrains. Datalore Documentation, 2020.

[64] Wei Jie, Junaid Arshad, Richard Sinnott, Paul Townend, and Zhou Lei. A review of grid authentication and authorization technologies and support for federated access control. *ACM Computing Surveys*, 43(2), 2011.

[65] Markus Jochum. Physical Oceanography, 2020.

[66] Jupyter Development Team. JupyterHub Authenticators, 2014.

[67] Jupyter Development Team. JupyterHub Authenticate State, 2016.

[68] Kaggle Inc. Kaggle, 2018.

[69] Kaggle Inc. Kaggle GPU Tips and Tricks, 2020.

[70] Kaggle Inc. Kaggle Notebooks Documentation, 2020.

[71] Raj Kettimuthu, Stuart Martin, and Bill Mihalo. Setting up and using a Globus Toolkit 5 based Grid, 2010.

[72] József Kovács and Péter Kacsuk. Occopus: A multi-cloud orchestrator to deploy and manage complex scientific infrastructures. *Journal of Grid Computing*, 16(1):19–37, 2018.

[73] Massimo Lamanna. The LHC computing grid project at CERN. *Nuclear Instruments and Methods in Physics Research, Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 534(1-2):1–6, 2004.

[74] Kim Lefmann. The templateSANS Instrument, 2021.

[75] LIGO. LSC DataGrid, 2020.

[76] John Eric Little, Xiaowen Yuan, and Mark Ian Jones. Characterisation of voids in fibre reinforced composite materials. *NDT and E International*, 46(1):122–127, 2012.

[77] Lund University. LUNARC, 2020.

[78] Aniruddha Marathe, Rachel Harris, David K. Lowenthal, Bronis R. De Supinski, Barry Rountree, Martin Schulz, and Xin Yuan. A comparative study of high-performance computing on the cloud. *HPDC 2013 - Proceedings of the 22nd ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 239–250, 2013.

[79] David Marchant. mig-meow, 2021.

[80] David Marchant and Rasmus Munk. Pre benchmark foam implementations, 2020.

[81] David Marchant, Rasmus Munk, Elise O Brenne, and Brian Vinter. Managing Event Oriented Workflows. In *2020 IEEE/ACM 2nd Annual Workshop on Extreme-scale Experiment-in-the-Loop Computing (XLOOP)*, 2020.

[82] Zdnek Matej and Artur Barczyk. Correspondence with MAX IV, 2020.

[83] Marta Mattoso, Jonas Dias, Kary A.C.S.Ocaña, Eduardo Ogasawara, Flavio Costa, Felipe Horta, Vitor Silva, and Daniel de Oliviera. Dynamic Steering of {HPC} scientific workflows: A survey. *Future Generation Computer Systems*, 46:100–113, 2015.

[84] MAX IV. MAX IV Controls and IT, 2021.

[85] MAXIV. MAXIV, 2018.

[86] Mattheieu Mayran, Lei Jerry Ding, and aleksejlopasov. DataprocSpawner, 2021.

[87] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. *NIST*, 2011.

[88] Matthew Metheny. Introduction to the Federal Cloud Computing Strategy. *Federal Cloud Computing*, pages 1–30, 2013.

[89] Microsoft. Azure Notebooks manage and configure projects, 2020.

[90] Microsoft. Azure Notebooks Overview, 2020.

[91] Michael Milligan. Interactive HPC Gateways with Jupyter and Jupyterhub. *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact - PEARC17*, pages 1–4, 2017.

[92] Felip Moll. Slurm Overview. 2018.

[93] Rasmus Munk. docker-volume-sshfs, 2018.

[94] Rasmus; Munk. jhub-swarmspawner, 2018.

[95] Rasmus Munk. MiG Utils, 2018.

[96] Rasmus Munk. DAG, 2019.

[97] Rasmus Munk. ldap_hooks, 2019.

[98] Rasmus Munk. Cloud Orchestrator, 2020.

[99] Rasmus Munk. HISS prototype, 2020.

[100] Rasmus Munk. mccode-builds, 2020.

[101] Rasmus Munk. CloudSSHSpawner, 2021.

[102] Rasmus Munk. notebook_parameterizer, 2021.

[103] Rasmus Munk and Jonas Bardino. Teaching Parallel and Distributed Techniques at UCPH through JupyterLab.

[104] Rasmus Munk, Jonas Bardino, Mads Ruben Burgdorff Kristensen, and David Marchant. NBI Jupyter Docker Stacks, 2021.

[105] Rasmus Munk, David Marchant, and Brian Vinter. Cloud enabling educational platforms with corc. In *Proceedings of the 8th Workshop on Cloud Technologies in Education (CTE 2020)*, 2020.

[106] Rasmus Munk and Brian Vinter. MUMMERING Platform Idea's and ubiquitous data analysis. In *Concurrent Systems Engineering Series*, volume 70, pages 323–333, 2019.

[107] Rasmus Munk, Brian Vinter, and Zdnek Matej. *From instrument to publication: A First Attempt at an Integrated Cloud for X-ray Facilities*. 2020.

[108] Nalind, Rhatdan, TomSweeneyRedHat, and Vrothberg. Buildah, 2020.

[109] NERSC. SSHSpawner, 2016.

[110] NERSC. JupyterLab Slurm Extension, 2021.

[111] Marco A S Netto, Rodrigo N Calheiros, Eduardo R Rodrigues, Renato L F Cunha, and Rajkumar Buyya. HPC Cloud for Scientific and Business Applications: Taxonomy, Vision, and Research Challenges. 51(1):1–29, 2018.

[112] NIST. National Institute of Standards and Technology, 2021.

[113] NorduGrid Collaboration. A-REX data transfer framework (DTR), 2020.

[114] NorduGrid Collaboration. Advanced Resource Connector Install Guide, 2020.

[115] NorduGrid Collaboration. DTR Design and Implementation Details, 2020.

[116] NorduGrid Collaboration. How to work with data?, 2020.

[117] nteract. papermill.

[118] NumPy. NumPy, 2020.

[119] NVIDIA. TensorFlow 2 Quickstart Notebook, 2020.

[120] OASIS. Topology and Orchestration Specification for Cloud Applications, 2020.

[121] OCCI. OCCI.

[122] OpenNebula Systems. OpenNebula, 2020.

[123] OpenSSH. OpenSSH, 2020.

[124] Oracle Corporation. LaunchInstanceDetails.

[125] Oracle Corporation. Oracle Cloud Infrastructure Compute Shapes, 2020.

[126] Oracle Corporation. Oracle Cloud Python SDK, 2021.

[127] Otsu and N. A threshold selection method from gray-level histograms. *IEEE Trans. on Systems, Man and Cybernetics*, 9(1):62–66, 1996.

[128] Paolo Padoan, Liubin Pan, Mika Juvela, Troels Haugbølle, and Åke Nordlund. The Origin of Massive Stars: The Inertial-inflow Model. *The Astrophysical Journal*, 900(1):82, 2020.

[129] Pandas. Pandas, 2021.

[130] Fernando Perez and Brian E. Granger. IPython: A System for Interactive Scientific Computing, Computing in Science and Engineering. *Computing in Science and Engineering*, 9(3):21–29, 2007.

[131] Dana Petcu. Multi-Cloud. page 1, 2013.

[132] Kristen Pind, Henrik Høy Karlsen, Rasmus Andersen, and Brian Vinter. Minimum Intrusion Grid. pages 0–5, 2005.

[133] Podman. Podman, 2020.

[134] Project Jupyter. LDAPAuthenticator.

[135] Project Jupyter. WrapSpawner.

[136] Project Jupyter. JupyterHub, 2015.

[137] Project Jupyter. JupyterLab, 2018.

[138] Project Jupyter. Project Jupyter, 2019.

[139] Project Jupyter. JupyterHub Spawners, 2020.

[140] Project Jupyter. DockerSpawner, 2021.

[141] Project Jupyter. Jupyter Docker Base Image, 2021.

[142] Project Jupyter. JupyterHub Architecture, 2021.

[143] Project Jupyter. KubeSpawner, 2021.

[144] Project Jupyter. YarnSpawner, 2021.

[145] Andrew Prout, William Arcand, David Bestor, Bill Bergeron, Chansup Byun, Vijay Gadepally, Matthew Hubbell, Michael Houle, Michael Jones, Peter Michaleas, Lauren Milechin, Julie Mullen, Antonio Rosa, Siddharth Samsi, Albert Reuther, and Jeremy Kepner. MIT SuperCloud portal workspace: Enabling HPC web application deployment. *2017 IEEE High Performance Extreme Computing Conference, HPEC 2017*, pages 1–6, 2017.

[146] Puppet. Puppet, 2020.

[147] Puppet. How Puppet orchestrator works, 2021.

[148] Puppet. Setting up the orchestrator workflow, 2021.

[149] pypa. Virtualenv, 2021.

[150] Python Software Foundation. Python Operating system interface, 2020.

[151] RedHat. What is Orchestration, 2020.

[152] David Reinsel, John Gantz, and John Rydning. The Digitization of the World - From Edge to Core. *Framingham: International Data Corporation*, 2018(November):US44413318, 2018.

[153] Ofer Rind, William Strecker-Kellogg, Daniel Allan, Douglas Benjamin, Mizuki Karasawa, and Kristy Li. Integrating Interactive Jupyter Notebooks at the BNL SDCC. *EPJ Web of Conferences*, 245:07054, 2020.

[154] Matthew Rocklin. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. *Proceedings of the 14th Python in Science Conference*, (SCIPY):126–132, 2018.

[155] RStudio. RStudio, 2020.

[156] Herwig Schopper and Luigi Di Lella. *60 Years of CERN Experiments and Discoveries*, volume 23. 2015.

[157] SciPy developers. SciPy, 2021.

[158] František Simančík, Jaroslav Jerz, Jaroslav Kováčik, and Pavol Minár. Aluminium foam - A new lightweight structural material. *Kovove Materialy*, 35(4):265–277, 1997.

[159] SNIA. Cloud Data Management Interface, 2021.

[160] SSH Communications Security Inc. SSH Forwarding, 2021.

[161] Sylabs Inc. Singularity Definition Files, 2019.

[162] Sylabs.io. Singularity, 2019.

[163] Miklos Szeredi and Bartosz Fenski. SSH Filesystem, 2018.

[164] Terasic Inc. DE10-Pro-GH2E2-280, 2021.

[165] Terasic Inc. Xlinx DE10-Pro, 2021.

[166] TerraForm. TerraForm, 2020.

[167] The Apache Software Foundation. Apache Airflow Documentation, 2020.

[168] The Apache Software Foundation. libcloud, 2020.

[169] The Linux Foundation. Open Container Initiative, 2020.

[170] The Linux Foundation. Open Container Initiative FAQ, 2020.

[171] The Open MPI Project. OpenMPI, 2004.

[172] Orazio Tomarchio, Domenico Calcaterra, and Giuseppe Di Modica. Cloud resource orchestration in the multi-cloud landscape: a systematic review of existing frameworks. *Journal of Cloud Computing*, 9(1), 2020.

[173] Tribler. Ipv8, 2021.

[174] Tribler. IPv8 REST, 2021.

[175] Giuseppe Tricomi, Giovanni Merlino, Alfonso Panarello, and Antonio Puliafito. Optimal Selection Techniques for Cloud Service Providers. *IEEE Access*, 8(iv):203591–203618, 2020.

[176] UCPH ERDA team. ERDA user guide. Technical Report June, 2020.

[177] University of Copenhagen. Applied Statistics, 2019.

[178] University of Copenhagen. Introduction to Computing for Physicists, 2019.

[179] University of Copenhagen. HPC Center at the University of Copenhagen, 2021.

[180] Marc André Vef, Nafiseh Moti, Tim Süß, Markus Tacke, Tommaso Tocci, Ramon Nou, Alberto Miranda, Toni Cortes, and André Brinkmann. GekkoFS — A Temporary Burst Buffer File System for HPC Applications. *Journal of Computer Science and Technology*, 35(1):72–91, 2020.

[181] Brian Vinter, Jonas Bardino, Martin Rehr, Klaus Birkelund, and Mads Ohm Larsen. Imaging Data Management System. In *Cloud NG:17 Proceedings of the 1st International Workshop on Next Generation of Cloud Architectures*. Belgrade, Serbia, 2017.

[182] Wikipedia. Amazon Web Services Wiki, 2020.

[183] Wikipedia. Desktop Virtualization, 2020.

[184] Wikipedia. File Transfer Protocol Secure, 2020.

[185] Wikipedia. Linux Containers, 2020.

[186] Wikipedia. OS-level virtualization, 2020.

[187] Wikipedia. SSH File Transfer Protocol, 2020.

[188] Wikipedia. WebDAV, 2020.

[189] Wikipedia. C, 2021.

[190] Wikipedia. Cloud Infrastructure Management Interface, 2021.

[191] Wikipedia. Lightweight Directory Access Protocol, 2021.

[192] Wikipedia. RUST, 2021.

[193] Peter Kjær Willendrup. McWeb, 2021.

[194] Peter Kjær Willendrup, Erik Knudsen, Kim Lefmann, Emmanuel Farhi, Uwe Filges, Kristian Nielsen, Per-Olof Åstrand, Klaus Lieutenant, and Peter Christiansen. McStas, 2021.

[195] Xilinx and Samsung. Samsung SmartSSD, 2021.

[196] YAML. YAML Ain't Markup Language, 2020.

[197] Katherine Yelick, Susan Coghlan, Brent Draney, and Richard Shane Canon. The Magellan Report on Cloud Computing for Science The Magellan Report on Cloud Computing for Science. *Energy*, page 170, 2011.

[198] Hao Zhang, Yonggang Wen, Haiyong Xie, and Nenghai Yu. *Distributed hash table: Theory, platforms and applications*, volume 0. 2013.

Appendices

Appendix A

Publications

A.1 MUMMERING Platform Idea's & Ubiquitous Data Analysis

Munk, Rasmus; Vinter, Brian. Communicating Process Architectures 2017 and 2018, WoTUG-39 and WoTUG-40 - Proceedings of CPA 2017 (WoTUG-39) and Proceedings of CPA 2018 (WoTUG-40). red. / Jan Baekgaard Pedersen; Kevin Chalmers; Jan F. Broenink; Brian Vinter; Kevin Vella; Peter H. Welch; Marc L. Smith; Kenneth Skovhede. IMIA and IOS Press, 2019. s. 323-333 (Concurrent Systems Engineering Series, Bind 70).

Communicating Process Architectures 2018
K. Chalmers, J.B. Pedersen et al. (Eds.)
IOS Press, 2018

MUMMERING Platform Idea's & Ubiquitous Data Analysis

Rasmus MUNK [a,1], Brian VINTER [a]
[a] *Niels Bohr Institute, University of Copenhagen*

Abstract. The growth from terabytes of 3D imaging data and soon approaching petabytes from material analysis has left the scientists involved with a set of challenges. In particular, the ability to efficiently analyze an ever growing collection of material tomography scans. The MUMMERING research project aims to solve this by providing ability to submit workflows to automate the process of analyzing the collected data. We explore and present our initial design thoughts in this endeavor. This includes a proposal to utilize the IDMC system developed at UCPH to provide an efficient method in terms of scheduling and execution of workflows. Beyond this mere exploration of thoughts about a potential solution for the MUMMERING project, this paper will also we introduce our initial work in providing ubiquitous access to the produced datasets. The aim here is to provide a simple API for loading/storing datasets during an image analysis, by providing this universal data access library (i,e. mig_utils). With this we hope to help the non computer scientists involved in defining imaging analysis programs that easily either can executed locally during the experimentation phase or subsequently scheduled by a workflow scheduler.

Keywords. MUMMERING, X-ray Imaging, Scientific Workflows, Data Analysis

Introduction

This paper sets out to describe the initial design thoughts and the work completed so far in providing a ubiquitous data sharing/analysis gateway for the MultiScale Multimodal and Multidimensional for Engineering aka MUMMERING research project. The overall aim of this research project is to create an open platform that can empower scientist efficiency by automating a number of standardized workflow tasks. The aim here is to extract meaning from terabytes and soon to be petabytes of 3D imaging tomography data of various materials. This is in contrast to the status quo of letting each scientist being let to their own accord, i.e. managing and figuring out how to efficiently exploit the distributed and parallel capabilities of modern compute resources with continuous growing datasets.

In this paper we will lay out our of initial design thoughts and decisions in delivering the stated aim. This includes how we should provide seamless sharing to the collected datasets between the various project participant organizations such MAX IV, University of Copenhagen, SAERTEX etc.

This sharing is suppose to both enable computational access to the data across numerous compute devices for later analysis, ranging from a scientist playing with a subset of the dataset on his workstation at home while developing a particular analysis, i.e. not really caring about performance but correctness, to being able to scale up an experiment to require terabytes of input data, while only imposing a minimal amount of necessary adjustments.

[1]Corresponding Author: *Rasmus Munk, sterbrogade 89, 5th.* Tel.: +45 61780755; E-mail: `munk1@live.dk`.

Furthermore, we will also explore how users should be able to define and submit workflows, how they will be executed, and in turn stored on the platform while providing the possible speedup benefits of exploiting an underlying distributed set of machines. This is to happen while having the subsequent users of the system in mind, especially in regards to the user interface and the required know-how it imposes on the users. Instead the scientist should be able to efficiently conduct their experiments without regard for trying to manually optimize their analysis, e.g. manually parallelize their algorithm to exploit a multi-core system or having to think about the physical data placement.

In regard to sharing datasets, an initial solution for the ubiquitous data analysis will be presented in the form of a Python library. This library is provided as a optional helper in transitioning existing data analysis scripts from depending on locally available storage, to reading resources from an external storage with only minor modifications. This prototype has for now only been evaluated with a number of runtime benchmarks, these will be presented to illustrate the expected runtime penalty that is imposed by loading data over the network instead of from a local disk drive.

Finally a set of conclusions will be drawn, based on the experience gained so far and the future steps from this point towards fulfilling the MUMMERING aims.

1. Background

As mentioned, MUMMERING is about analyzing various composite engineering materials at high imaging resolutions. e.g. analysis the strength properties of novel materials in a pressure simulation. This process normally involves scanning the material in question by firing X-rays at the object and subsequently using the detected wavelength variance to produce a set of high-resolution tomography images of the object.

These images are then collected and analyzed by executing either an existing analysis or having a domain expert develop one that parses the dataset for areas of particular interest to the project. For instance, investigating voids in fiber reinforced composite materials [1]. To conduct such an analysis, a number of steps has to be completed. This includes scanning the material in question at a facility that can generate the required X-ray's (e.g. a synchrotron such as [2]). Between scans either the material itself or the scanning instruments are continuously rotated to generate a set of images that can be used to create a 3D model of the sample. For example, in [3] a scan is performed for every 1/3 of a degree for 360 degrees resulting in 1080 images, which depending on the resolution and pixel data type can accumulate to several GB of data per image. For instance, a single high-resolution image can easily reach 50GB for a single image. Resulting in that only 20 material scans can produce $\tilde{1}$TB of data.

1.1. Workflows and Analysis development

After the collection has been completed, the next step is typically to produce a 3D physical modeling of the object/material in question. To produce such a model, a series of typical image transformation steps has to be executed, i.e. 3D reconstruction, segmentation and meshing.

This process is currently being taught to be manually implemented by each scientist/researcher for each of their particular research e.g. at PhD summer schools [4]. The process itself is important to know from a science perspective, but going from a simple experiment on a single image, to scaling up to hundreds of images while still maintaining an efficient and reasonable execution time can be a daunting, tedious and repetitive task. In addition, this process also naturally restricts the amount data that can be explored for features that might be of interest to the research. Furthermore, it also often leads to re-implementations of the same workflow building blocks across similar projects.

What is required here is a workflow framework that is able to schedule and execute imaging analysis tasks. One similar case study to this is the development of the MapReduce framework [5] that has lead/inspired numerous Big Data frameworks such as the batch oriented Hadoop framework that almost replicated it [6] or its data streaming successors such as Storm and Heron [7]. The project was tasked with a similar objective as the MUMMERING project, namely to develop a platform that provides the capabilities that engineer was reimplementing across numerous data analysis problems. In particular modifying existing sequential implementations to exploit the parallel nature of the distributed machines that they had access to, while taking care of the underling tedious fall-pits of a distributed architecture (e.g. fault tolerance, redundancy, etc). This was accomplished by implementing a query processor that accepted batch jobs that got divided into two tasks, namely Map and Reduce. A Map job which is defined as the function that filters the dataset in question for records of interest, whereas Reduce performs some form of reduction operation that results in the final output of the job. What is common is that these are both exposed as user defined functions, i.e. exposed API methods that the user can implement their normal analysis without worrying about job scheduling and the likes as-long as the input/output data structures abide by the API contract. This simplifies the the usage experience and limits the amount of errors that can be introduced by non-computer experts.

Similar to this capability of scheduling and executing jobs, the MiG (Minimum Intrusion Grid) [8] has been developed at UPCH which provides similar capabilities. Specifically it can also accept and manage job scheduling on a set of distributed resources like Hadoop. It differentiates itself by being a middleware system that requires little of the nodes that join the grid as an available compute resource. In this regard it aims to reduce the software complexity that a distributed or grid system would usually impose on every compute resource (e.g. Java and Hadoop has to installed on every node for the cluster to function [6]. It does this by only requiring SSH and secure HTTP access to a given grid resource before the resource is able to pull jobs from the grid [8].

Recently the MiG system was extended with an event-driven and adaptable scientific workflow model well-suited for collaboration between multiple users [9]. This event-driven architecture can schedule a series of workflow tasks (e.g. reconstruction, segmentation and meshing from a set of input images). The cornerstone of this execution chain is that in comparison to a typical workflow manager like Apache Oozie [6] that takes the top-down approach of expecting a series of interdependent jobs to be either defined as a linear job chain or direct acyclic graph [6], which imposes the constraint of knowing beforehand what output will be produced ahead of time to develop the appropriate set of jobs. Instead the data event-driven workflows introduced by MiG will schedule the necessary jobs required to complete a workflow based on a set of preset triggers. The event triggers in this instance are implemented by the MiG system and are built on-top of the inode notify Linux kernel feature [10]. This means that the starting mechanism to a workflow is based on some sort of file operations, whether it be a created/modified/deleted request that is captured by the inotify monitor [11]. This also means that after an initial event is set in motion, the subsequent flow of executed jobs/tasks is dependent on the preset triggers and the output generated from theses processes. The decision about which files should be monitored and what sort of job should be executed is defined by the user themselves. However, currently this has a somewhat laking interface in terms of ease of use because of the complexity involved in defining correct triggers.

Beyond this, UPCH has recently developed an optional extension to the MiG system in terms of a web based development environment. Specifically the MiG is now capable of forwarding users from the native data management platform to a JupyterHub host where users can run individual JupyterLab's [12]. This platform provides a web powered interactive environments that exposes a set of development environments in capable data analysis languages such as Python or R. The aim of this is that scientists, students and similar users with a requirement here have an environment to develop their analysis/programs. This is useful in that Jupyter notebooks have become a popular method to share data analysis programs at places such as Kaggel [13]. This is expected to be useful in MUMMERING in that it can provide easy access to compute resources that can be used by the researchers when they develop and test their analysis. Especially since the integration with the MiG system allows native access to personal files stored in the MiG system within Jupyter environment, making the task of loading in data from it no different than on a local system.

1.2. Data Management

Beyond the workflow capability, the MiG also provides an entire web based interface to support management of project data. In addition the system also provides a number of collaboration tools. This includes the WorkGroup feature [14] that enables sharing files with different users on a project basis (i.e. a shared folder that is often tide to a particular project that only the participants have access to). This is in addition to just managing the ever growing amount of data uploaded to the system. At UCPH the IDMS (Imaging Data Management System) [15] has been developed to face this challenge in particular when it comes to imaging data, at its core, the IDMS is build on-top of the MiG base so it natural inherits the mentioned MiG features. For instance it is possible to setup a trigger that schedules imaging processing jobs when they are uploaded to a specified workgroup on IDMS, e.g. trigger a job that returns a preview for each image, in addition a number of common statistics are calculated and viewable directly in the web interface. [15].

Currently a version of IDMS, namely IDMC (Imaging Data Management Center) is hosted at the UCPH HPC center. It is setup as a shared storage system specifically to solve the mentioned problem of storing large scientific imaging datasets. At present date the system has 1.6 PB of disk storage designated for free use for UCPH employees and accepted affiliated project partners. So far the system has accumulated 1.4 PB of data, which is growing at an estimated 4 TB a day. Based on the presented findings, the IDMC system seems to be a prime candidate for being part of the solution to deliver the goals stated in the MUMMERING project, with the mentioned current lack of usability for features such as the trigger and workflow setup in mind.

1.3. Sharing Data

In terms support for accessing data on the IDMC system a variety of SSH based methods are supported including SSHFS [16] and SFTP [17]. Utilizing these transfer protocols, a user can for instance upload/download datasets via SFTP or synchronize a home drive via SSHFS . This is provided that the user authenticates either via public key authentication or a preset password. Beyond this personal access model, IDMC also provides a feature for users to share access to their own files with external non-users such as a MUMMERING partner. The mechanism for providing this is called Share links [14]. A Share link is defined as a URL that can provide anonymous access to a set of resources on the IDMC system. It is made up of 2 parts, the base URL that defines where the resources are hosted and additional random ID. The ID is string of 10 random characters (e.g. 'FjCd54pWd7'), the 10 characters are picked from a set of 64 possible characters which encompasses both the lower and uppercase English alphabet and digits ranging from 0 to 9. The Share link itself is generated when a user requests to share file resources via the IDMC web interface. In addition to specifying the resource, the user is also presented with the option of specifying access permissions to the resource i.e., either read, write or both. At the end of this process the user is giving a URL (e.g. https://sid.idmc.dk/sharelink/FjCd54pWd7) which can be used to share the resources in question with any external or internal entity.

Essentially this behavior is similar to the Dropbox sharing feature [18], with the described extension of defining access controls. In summary, by both providing sharing capabilities and being an existing integrated part of the IDMC system makes it a suitable candidate for enabling ubiquitous data access to both scientists and external partners involved in the MUMMERING project.

2. Initial Platform Ideas

In this section the proposed software stack for the open platform is presented. This includes what interfaces will be exposed to the users/scientists and our thoughts about how we provide a user friendly way of defining workflows that can be mapped to the underlying MiG job scheduler. This is followed by an overview of the MUMMERING organization and how we propose to provide ubiquitous data analysis/storage between the various partners.

2.1. MUMMERING Platform Stack

To provide an open platform with the capability of executing image analysis workflows. We propose the following design as shown in Figure 1. The stack is divided into 3 areas of responsibility. As defined in the legend section of Figure 1.

The submit layer is where the users can submit workflows, the layer depth here indicate the amount of controllability and options that will be available to the user. For basic users we propose that they define their job as Jupyter Notebook, which can then be submitted with an adjacent DSL description of the input data to be processed and the code within. The initial thought to how this DSL could be structured can be seen Listing 1. As presented here the idea is to provide capabilities to define a set of overall processes that should take place e.g. which code pieces/functions inside an arbitrary Notebook should be executed on the provided dataset. This of course has to be mapped down to the MiG workflow implementation. Whether this is feasible in the shown format is something that has to be explored further.

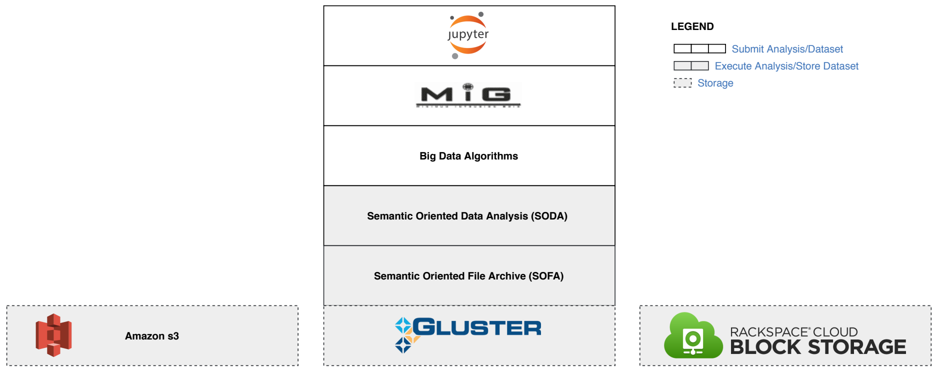


Figure 1. Infrastructure Stack for MUMMERING.

```
input: {path: "images"}
execute:{
    { notebook['function'].ipynb, output_file } ->
    { notebook['function'].ipynb, output_file } ->
    { notebook['function'].ipynb, output_file}
}
output: sharelink/IDMC homedrive
```

Listing 1: Example Job DSL.

In addition beneath the Jupyter job submission interface, a number of existing service layers are defined. MiG in this instance would be the IDMC system that would handle and accept the submitted DSL descriptions. In addition the underlying layers (particular SODA/-SOFA) are defined to optimize the read/write operations of large datasets as presented in [19]. In particular the ability to gain fine grained control of how the data is organized in terms of what defines an individual block in dataset, or the ability to file/process data while it is being uploaded to the system. However, the takeaway at this point is that the layers should be loosely coupled so that an individual service layer can be swapped out. For instance, the designated storage could be a Cloud service such as Amazon s3 or RackSpace Block Storage, as well as the internal storage system at UCPH i.e Gluster.

2.2. Ubiquitous Data Analysis

As introduced earlier, we proposed to utilize the IDMC Share links feature to enable scientist to access their hosted datasets from within their Python image analysis or Jupyter Notebook implementations. This is accomplished by introducing a Python utilities library called mig_utils [20] that abstracts the Share link access method of establishing an SFTP connection to the target Datastore.

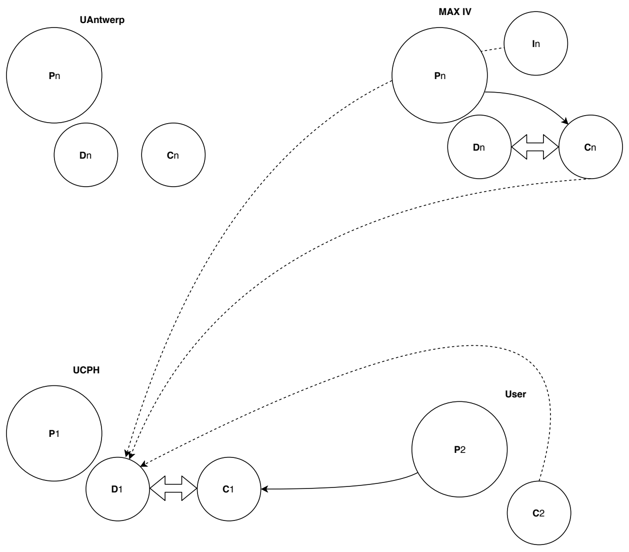


Figure 2. MUMMERING Organizational Overview. P = Partner/Organization, D = Datastore, C = Compute resource, — = Submit execution on remote compute resource, - - - = load or store data from/on data store, $\Longleftrightarrow$ High speed 1 Gbit/s bandwidth.

The design example of this can be seen in Figure 2, where for instance a scientist located at MAX IV, implements an image analysis algorithm to investigate recent X-ray scans of a novel material composition. During the scanning process the instrument is configured to store the data directly at a UCPH storage facility (i.e. D1). When this is completed the researcher implements their analysis as usual, but instead of using the standard Python io module to load in the images. They instead uses the mig_utils.io module to open and read the image from a predefined Share link location. Now, during the acquisition phase the mig_utils library will download the required dataset from where it is hosted, in this case D1 at the UPCH center before proceeding to execute the actual analysis.

To provide the capability of receiving large quantities of data and storing it for an indefinite time frame, the UPCH provides a generic data management cloud system for imaging data, i.e IDMC [3] [15]. At present date the system has 1.4 PB of storage available to users such as UCPH employees and affiliated organizations which could possible include MUMMERING partners.

3. Initial Implementation

Mig_utils [20] is a Python library that is compatible with Python version 2.7, 3.5, 3.6 and 3.6-dev. Currently it provides a single io module, that enables remote file access via SFTP. It provides an API that mimics the built-in file IO syntax of [21]. A simple example of writing and reading a file to/from the IDMC system can be seen in Listing 2. The share open(path, flag) method, supports the following modes of operation 'w', 'r' 'a' and 'b' operates as defined by Python's built-in open method [22]. It's important to note here that the IDMCShare object is a simple child of the general SFTPShare that makes it possible to connect to other host systems that supports SFTP clients. The child class is simply added as syntactic sugar, i.e. making it very simple to use from the perspective of UCPH, MUMMERING and the involved participants and collaborators when interacting with the IDMC system.

```
from mig.io import IDMCShare
# returns an IDMCShare class that inherits from SFTPStore
share = IDMCShare('SHARELINKID')

# open share with the
# open a file for writing, truncate if exist
with share.open('write_test', 'w') as tmp:
    tmp.write("sdfsfsf")

# Read a file, return as a string
with share.open('write_test', 'r') as tmp:
    print(tmp.read())

# close the connection
share.close()

# outputs 'sdfsfsf'
```

Listing 2: Write and Read example.

Beyond the standard 'open' and the 'close' method, the share object also provides additional methods. This is to support client-side exploration and cleanup of files as shown in Listing 3. Here the 'list' path by default is set to '.', which returns a list of files in the root context of the connected Share link.

```
from mig.io import IDMCShare
share = IDMCShare('SHARELINKID')
share.list(path='path/to/sub/dir')
share.remove(path='path/to/file/to/remove')
share.close()
```

Listing 3: Share API.

In terms of the call to 'open' as shown in Listing 2, the return value is an SFTPFileHandle object, the supported methods of which includes 'read' and 'write'. Which depending on the flag of operation either returns strings 'r'/'w' or byte strings 'rb'/'wb'. In addition it also provides access to 'seek', 'tell' and 'close' as defined by the default abstract BaseIO object in the core IO module [23]. With the single caveat that 'seek' does not support the 'whence' flag but can only seek from the current position. An example of their use can be seen in Listing 4.

```
with share.open('another_example', 'w') as tmp:
    tmp.write("Hello remote file")

# open another_example in string read mode
tmp = share.open('another_example', 'r')
# Read from current file position, default 0
print(tmp.read())
# Return current file position
print(tmp.tell())
# Go to file position 0
tmp.seek(0)

# Return current file position
print(tmp.tell())
# Read from current file position
print(tmp.read())
# Go to file position 6
tmp.seek(6)
# Read from current file position
print(tmp.read())

tmp.seek(0)
# Close the current SFTPFileHandle
tmp.close()
# Try to read again
print(tmp.read())

#### The output of which is ###
Hello remote file
17
0
Hello remote file
remote file
```

Listing 4: SFTPFileHandle IO.

Beneath the exposed API, the connection to the share is by default established via a set of SFTP calls. The SFTP connection itself is provided by utilizing the ssh2-python library [24]. This library in turn is a thin wrapper around libssh2, which is a C implementation of the SSH2 protocol [25].

4. Prototype Benchmarks

To evaluate the performance of the mig_utils library, i.e. its ability to extract/store data at native link speed seamlessly with minor adjustment to an existing implementation. Two simple image analysis/overview notebooks provided by [26] and [27] were used for the benchmarking [26]. Each of these notebooks perform their analysis on a single image, respectively large-set.tif at 127.07 MB and small-set.tif at 68.47 MB. In terms of implementation the code itself can be categorized as mostly sequential array programming via Numpy. Therefore it has no benefit of underlying parallelism capabilities.

By default each notebook loads in the image with the skimage library [28] imread function, which takes a path parameter to a local file or URL and returns an N-dimensional Numpy array. This transformation in the instance of a TIF formated image happens by using the

Python Imaging Library [29] to produce an nd_array by first passing the filename or filehandler to the PIL.Image.Image.open method which returns an PIL object that can be convert into an nd_array by using the supplied pil_to_ndarray function. This functionality, as shown in Listing 5 is extracted directly from the libraries. This is to ensure that when the image byte string is downloaded from the remote storage it is transformed into a numpy array which is the format that the notebook examples expects.

```
import time
from PIL import Image
from skimage.io._plugins.pil_plugin import pil_to_ndarray

share = IDMCShare('SHARELINKID')
file1 = 'rec_8bit_ph03_cropC_kmeans_scale510.tif'
start = time.time()
with share.open(file1, 'rb') as fh:
    load_start = time.time()
    # Load image into an PIl.Image.Image obj
    pil_image = Image.open(io.BytesIO(fh.read()))
    # Transform PIL into an ndarray
    nd_image = pil_to_ndarray(pil_image)
    load_stop = time.time()
    # execute notebook
    foam_labelling(nd_image)
stop = time.time()

share.close()
```

Listing 5: Transform image bytestring into an nd_array.

For these benchmarks, the physical location of the required images was on the IDMC system hosted at UCPH i.e. D1 in Figure 2 and the executing compute resource were C1 and C2. This translates into bandwidth connection of 1 Gbit/s and 100 Mbit/s respectively. In addition to highlight the penalty of loading the data externally, the benchmarks were also with the image locally stored on C2.

In terms of underlying compute power the C1 device is a shared compute resource with an Intel 20 core CPU with hyper-threading running at 2.59 GHz and 230 GB of memory. The access is provided by UCPH HPC center as an isolated containerized JupyterLab environment with downscaled access to 8 logical cores and 8 GB of memory. C2 however is a personal workstation with an AMD Ryzen 7 2700x 8 core CPU with hyper-threading running at a base clock of 3.7 GHz. In terms of local storage I/O, C2 has an EVO 970 V-NAND SSD with a theoretical sequential read speed of 3400 MB/s. However, practical tests with hdparm showed an estimated read speed 1500 MB/s equivalent to 12 Gbit/s.

In terms of the number of executions, each benchmark result listed in Table 1 and 2 is the result of 40 runs.

Table 1. Foam Labeling Notebook Performance (large-set.tif).

compute location	*'avg load time'*	*'avg total exec time'*	*'bandwidth'*	*'data location'*
UCPH C1	1.49	207.02	1 Gbit/s	D1
User C2	10.51	147.10	100 Mbit/s	D1
User C2	0.2	138.4	12 Gbit/s	Local Storage

As shown in Table 1 and 2 the results live up to the expectation of the improved performance in tandem with the increased bandwidth. I.e. with a 127.07 MB image loaded in 1,49 seconds, the benchmarks shows an estimated minimum transfer rate of 85.77 MB/s. However

Table 2. Aluminum Overview Notebook Performance (small-set.tif).

compute location	*'avg load time'*	*'avg exec time'*	*'bandwidth'*	*'data location'*
UCPH C1	3.92	7.67	1 Gbit/s	D1
User C2	32.2	35.14	100 Mbit/s	D1
User C2	0.62	3.38	12 Gbit/s	Local Storage

this should be interpreted with the addition that this includes the additional time it takes to transform the image bytes into an numpy array as shown in Listing 5. This means that the load times shows the actual time it takes for the image to be ready for analysis as required by the provided notebooks. However, still with this in mind it is no surprise that local image has both the fastest load and execution time of 0,2 and 138,4 seconds. The reason being both because of the smaller latency and greater bandwidth in terms of load time from a local V-NAND SSD, but also in regards to a substantial lower total execution time with a 1.42 GHz greater clock speed in comparison to C1. When evaluating the displayed performance in Table 2, it is important to note that the provided Aluminum Notebook is inefficiently implemented, such that it reads the same image 5 times during the execution. I.e. a total of 342.35 MB to load, which in the C2 100 Mbit/s bench gives an estimated transfer average of at least 10.6 MB/s and 87.33 MB/s for C1.

5. Conclusions

In this paper we put forward our initial thoughts in terms of how to provide an open platform for the MUMMERING research project. This included that scientists should be able to submit Jupyter Notebooks as workflow jobs in combination with a DSL description. We plan to provide this capability on top of the existing IDMC system at UPHC, which will handle the underlying job scheduling/queuing with it's recent event driven workflow support.

In addition, we propose an initial implementation of how scientists in MUMMERING can get ubiquitous access to their generated imaging datasets via Share links on the IDMC system. This provides the capability of making an data analysis independent of where the data is actually stored. However, with the caveat of a slowdown in performance that is based on the available bandwidth at the point of execution. Furthermore the prototype is also currently limited by the executing nodes physical memory and swap sizes, i.e. presently the entire datasets is downloaded into memory before proceeding with the analysis. However, this is no different than the present constraint when loading data from a local storage drive.

6. Future Work

A number of things have to be accomplished from this point. This includes abstracting the IDMC workflow definitions in a user-friendly manner, that makes them accessible to non computer scientists. Furthermore, the ability to define Jupyter notebooks as jobs that can be scheduled in the underlying MiG infrastructure is a recommended approach because of the user friendly interface and the ability to support numerous languages and the ability to easily export and share notebook kernels. In the best case scenario the ability to submit a notebook as a workflow job should be language agnostic. Currently the mig_utils library only supports typical Python IO operations. However, providing a set of standard image processing functions, such as the ones provided by the pillow library (e.g. imread/imsave) would be preferable. This would ease the transition of redefining a standard notebook that requires data to be locally present to one that utilizes the ubiquitous io operations. Furthermore, initial testing of using coroutines and the async/await capabilities of Python 3.5/6 for pre-fecthing image data shows good promise for bringing down the total runtime of an individual notebook, especially when processing multiple images over the same notebook.

Acknowledgements

The authors would like to thank Rajmund Mokso and Kevin Mader for providing Jupyter notebook examples that were useful for benchmarking the mig_utils library.

References

[1] J. E. Little, X. Yuan, and M. I. Jones, "Characterisation of voids in fibre reinforced composite materials," *NDT and E International*, vol. 46, no. 1, pp. 122–127, 2012. [Online]. Available: http://dx.doi.org/10.1016/j.ndteint.2011.11.011

[2] MAX IV, "MAX IV," 2018. [Online]. Available: https://www.maxiv.lu.se/

[3] B. Vinter, J. Bardino, M. Rehr, K. Birkelund, and M. Ohm Larsen, "Imaging Data Management System," in *CloudNG:17*, 2017, p. 6.

[4] J. W. Andreasen, "CINEMAX IV," 2018. [Online]. Available: http://www.conferencemanager.dk/CINEMAXIV

[5] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Proceedings of 6th Symposium on Operating Systems Design and Implementation*, pp. 137–149, 2004.

[6] T. White, *Hadoop : The Definitive Guide*.

[7] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter Heron," *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD '15*, pp. 239–250, 2015. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2723372.2742788

[8] J. Berthold, J. Bardino, and B. Vinter, "A Principled Approach to Grid Middleware," in *Algorithms and Architectures for Parallel Processing*, vol. 7016. Springer, 2011, pp. 409–418.

[9] J. Bardino, M. Rehr, and B. Vinter, "Event-driven , Collaborative and Adaptive Scientific Workflows on the Grid," pp. 59–78, 2017.

[10] "inotify," 2018. [Online]. Available: https://en.wikipedia.org/wiki/Inotify

[11] M. Kerrisk, "inotify," 2017. [Online]. Available: http://man7.org/linux/man-pages/man7/inotify.7.html

[12] Project Jupyter, "JupyterLab," 2018. [Online]. Available: http://jupyterlab.readthedocs.io/en/stable/

[13] Kaggle Inc, "Kaggle," 2018. [Online]. Available: https://www.kaggle.com

[14] UCPH ERDA team, "ERDA User Guide," Tech. Rep. March, 2018.

[15] ——, "IDMC," 2018. [Online]. Available: www.idmc.dk

[16] M. Szeredi and B. Fenski, "SSHFS," 2018. [Online]. Available: https://linux.die.net/man/1/sshfs

[17] T. Ylonen and S. Lehtinen, "SFTP," 2013.

[18] Dropbox, "Dropbox Sharelink," 2018. [Online]. Available: https://www.dropbox.com/help/files-folders/view-only-access

[19] K. Skovhede, "Big Data Analysis with Skeletons on SOFA," no. August, 2017.

[20] R. Munk, "MiG Utils," 2018. [Online]. Available: https://pypi.org/project/mig-utils/

[21] Software Foundation Python, "Python Input and Output Tutorial," 2001. [Online]. Available: https://docs.python.org/3.6/tutorial/inputoutput.html

[22] Python Software Foundation, "Python Built-in Functions," 2001.

[23] ——, "Python IO Module," 2001. [Online]. Available: https://docs.python.org/3/library/io.html

[24] Pkittenis, "ssh2-python," 2018. [Online]. Available: https://github.com/ParallelSSH/ssh2-python

[25] S. Golemon, M. Gusarov, I. The Written Word, E. Fant, D. Stenberg, and S. Josefsson, "libssh2," 2004. [Online]. Available: https://www.libssh2.org

[26] R. Mokso, "HSC2017_foamLabel notebook," 2017. [Online]. Available: https://www.kaggle.com/rajmund/hsc2017-foamlabel

[27] K. Mader, "Aluminum Overview," 2018. [Online]. Available: https://www.kaggle.com/kmader/aluminum-overview

[28] S. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, and T. Yu, "scikit-image: image processing in Python," *PeerJ*, vol. 2, p. e453, 2014. [Online]. Available: https://peerj.com/articles/453

[29] F. Lundh, A. Clark, A. Houghton, and J. Kun, "Pillow," 1995. [Online]. Available: http://python-pillow.org/

A.2 From instrument to publication: A First Attempt at an Integrated Cloud for X-ray Facilities

Munk, Rasmus; Brian Vinter; Zdenek Matej; Artur Barczyk. In the Book of Abstracts of Cloud Services for Synchronisation and Sharing (CS3) 2020 - the 6th Cloud Synchronization & Sharing Conference (Copenhagen, 27-29 January 2020). red. / Massimo Lamanna; Jakub T. Moscicki;

Europe has to make hard choices in all areas if it wants to maintain its sovereignty to be able to orient, decide and act in the information age. The presentation will elaborate developments that led to the current situation and explore options and consequences that could be taken.
A special focus will be taken towards how a CS3 Stack as precursor could lead to a "European Standard Stack" that could technologically lead to a next generation technology leap akin and in the tradition of the wwww.

Digital Sovereignty / 102

The importance of open source software in delivering large storage services

Author: Alberto Pace[1]

[1] *CERN*

Corresponding Author: alberto.pace@cern.ch

This (short) presentation will address the aspect of on on-premises versus cloud storage and the importance of using open source software in maintaining data sovereignty while delivering large storage services.

Clearly commercially licensed software can also be used as a part of a general complex architecture, but the presentation will discuss the checklist to be validated to avoid vendor lock-in or uncontrolled growing infrastructure costs.

Digital Sovereignty / 124

Digital Sovereignty can only be achieved through pure Open Source Software

Corresponding Author: frank@nextcloud.com

Fabric and platforms for Global Science / 108

From instrument to publication: A First Attempt at an Integrated Cloud for X-ray Facilities

Authors: Rasmus Munk[1] ; Brian Vinter[1] ; Zdnek Matej[2] ; Artur Barczyk[2]

[1] *Niels Bohr Institute*

[2] *MAX IV Laboratory*

Corresponding Authors: rasmus.munk@nbi.ku.dk, vinter@nbi.ku.dk

Large scientific X-ray instruments such as MAX IV [1] or XFEL [2] are massive producers of annual data collections from experiments such as imaging sample materials. MAX IV for instance has 16 fully funded beamlines, where 6 of which can produce up to 40 Gbps of experimental data during a typical 5 to 8 hour time-slot, resulting in up to 90 to 144 TBs for a particular beamline experiment.

Scenarios like this calls for solutions that can manage petabytes of datasets in an efficient manner, while enabling scientists with a path of least resistance to define on the fly and subsequent batch processing that often seeks to find needle answers in the data haystack. General outlier detection, pattern recognition and basic statistics just as bin counting are some of the typically tasks conducted during the post analysis phase. To enable scientist with such capabilities, the current challenges calls for an integrated solution that is both able to scale horizontal in terms of available storage, but also be able to make on the fly informed decisions that could potentially either reduce the experimental data stream before it is persistently stored, or enable feedback mechanisms to the instrument itself about which data is of interest to the scientists and that which has no or little value.

The continuous collaboration between the eScience group [3] at the Niels Bohr Institute and the MAX IV facility through their Data STorage and Management Project (DataSTaMP) [4] and European Open Science Cloud (EOSC) [5] participation aims to provide just such an integrated cloud solution to elevate the combined data services available to researchers in general.

The architecture design to enable this is made of two distinct services. The HIgh Throughput Storage System (HISS) and the Electronic Research Data Archive (ERDA) [6]. HISS is a developing distributed system that is designed as a high speed I/O gateway of storage nodes for stream oriented data collections. The system does this through temporary buffering during storage and retrieval of high bandwidth streams, acting in a sense as a front proxy to a subsequent persistent storage location such as a PFS or tape archive system. In addition to being a mere set of buffer nodes that allows for temporal storage reservations, the system is also being designed to allow for an on the fly scheduling of operations to be conducted during the I/O of datasets by scheduling preprocessing tasks on an FPGA accelerator. This enables for both in situ decisions about particular data points mid stream or general data reduction/prefiltering as specified by a user defined kernel, that may also introduce feedback streams to the data provider itself. A provider in this instance could be a beamline instrument at MAX IV.

The system enables access through a REST API that is inspired by and aims to be compatible with the AWS S3 [7] service and commandline tools. To define the computational kernels that are targeted for accelerated computation the proposal is to transpile python kernels into VHDL through an eScience developed toolchain that consists of Bohrium [8], SMEIL [9] and SME [10].

The target of the HISS offloading in our instance, is the ERDA system that is subsequently responsible for retaining the incoming collections, that is either stored GPFS or tape archives. The archive then on top of this provides a rich set of features for both managing and post-processing of data upon being stored. This includes Dropbox like sharing and synchronization in addition to efficient data access to home and collaborative datasets through standard secure protocols like WebDAV over SSL/TLS (WebDAVS), FTPS and SFTP. For processing the service enables processing of existing datasets through a JupyterHub [11] environment with container based JupyterLab [12] sessions for interactive executions of personal or collaborative resources.

It is the aim of this integrated cloud solution to enable both the receival of instrumentation data streams directly from the source, while allowing user defined decision making to take place before the data is persistently stored. For instance, the user could specify a reduction or statistics kernel that would alleviate the need to schedule such processing upon finishing the experimental phase. Enabling them to immediately interpret the results generated from the computed metadata.

[1] https://www.maxiv.lu.se
[2] https://www.xfel.eu
[3] https://www.nbi.ku.dk/Forskning/escience/
[4] https://www.maxiv.lu.se/accelerators-beamlines/technology/kits-projects/datastamp/
[5] https://ec.europa.eu/research/openscience/index.cfm?pg=open-science-cloud
[6] http://www.erda.dk
[7] https://docs.aws.amazon.com/cli/latest/userguide/cli-services-s3.html
[8] https://bohrium.readthedocs.io
[9] https://github.com/truls/libsme
[10] https://github.com/kenkendk/sme
[11] https://jupyterhub.readthedocs.io/en/stable/
[12] https://jupyterlab.readthedocs.io/en/stable/

A.3 Managing Event Oriented Workflows

Marchant, David; Munk, Rasmus; Brenne, Elise O; Vinter, Brian. XLOOP2020 at Super Computing 2020

Managing Event Oriented Workflows

David Marchant
Niels Bohr Institute
University of Copenhagen
Copenhagen, Denmark
d.marchant@ed-alumni.net

Rasmus Munk
Niels Bohr Institute
University of Copenhagen
Copenhagen, Denmark
rasmus.munk@nbi.ku.dk

Elise O. Brenne
Department of Energy
Conversion and Storage
Technical University of Denmark
Lyngby, Denmark
elbre@dtu.dk

Brian Vinter
Aarhus University
Faculty of Technical Sciences
Aarhus, Denmark
vinter@au.dk

***Abstract*—This paper introduces an event-driven solution for modern scientific workflows. This novel approach enables truly dynamic workflows by splitting them into their constituent parts, defined using combinations of Patterns and Recipes, and lacking any meaningful inter-dependencies. The theory behind this system is set out, and an example workflow is presented. A python package mig_meow, which implements this workflow system is also shown and explained. The use cases of various user groups are considered to asses the feasibility of the design, and it is found to be sufficient, especially in light of recent workflow requirements for dynamic looping, optional outputs and in-the-loop interactions.**

I. Introduction

Scientific workflow systems are an essential part of modern research. They are used to process large datasets on numerous heterogeneous resources. Traditional scientific workflow systems adopt a so-called static structure, where all processing, data and outcomes are set at the beginning of the workflow. This is sufficient for a number of use cases, but some have identified a need for a dynamic structure to their workflows [20].

To address this, this paper proposes Managing Event Oriented Workflows (MEOW), a system for defining event-driven workflows. This is done by splitting workflows into separate steps, each individually defined using Patterns and Recipes. This differs from the more traditional approach, which is typically built on a static pre-runtime analysis of the complete workflow structure. The proposed dynamic structure facilitates recent scientific workflow innovations such as cyclic execution, or workflow branching. It also enables some unique features such as completely optional outputs from individual workflow steps.

To illustrate the system described, this paper outlines a prototype implementation of a MEOW based event-driven workflow scheduler on the Minimum Intrusion Grid (MiG) [5], [6]. Furthermore, it is demonstrated how this system is suitable for a whole range of scientific use cases, and is particularly suitable for modern workflow developments.

This project is developed as part of the MUMMERING[1] [24] research project with the aim of providing researchers with a generic tool of the management and processing of scientific sets of data.

[1]*MU*ltiscale, *M*ultimodal, and *M*ultidimensional imaging for *E*nginee*RING* (MUMMERING)

II. Background

A. Scientific Workflows

To meet the needs of scientists, workflows as a concept have been adapted from the more 'traditional' models used within business. The specific implementations of workflow systems designed for scientists are referred to as 'scientific workflows' with examples such as Kepler [1], Pegasus [9], Taverna [25], and Globus Workflow [15], among others. These efforts usually employ a top-down model when defining how the set of tasks should be processed, which in turn produces a workflow that defines a complete chain of steps [10]. The resulting jobs are then scheduled appropriately according to their various dependencies. However, for tasks such as exploring large datasets [11], it is unlikely that the initial assumptions will produce a successful result without having first explored several workflow permutations. In this situation, a dynamic workflow which can be adapted at runtime would be advantageous [20].

B. Static and Dynamic Workflows

Scientific workflows tend to be data oriented and exploratory in nature [7]. This is as scientists are often running workflows as part of their experiments, and so may be running the same workflow repeatedly to explore an experiment space. The nature of these repeated runs may be unknown at the start of a scientific investigation, unlike in more traditional business workflows where it is expected that a user will already have a full understanding of the work to be carried out. This means that scientific workflows often have a specific need to be dynamic [7], [11], [17], [20]. Dynamic here means that jobs within the workflow may be added, modified, rerun, or removed without having to restart the whole workflow.

The majority of recent scientific workflow developments such as Apache Airflow [4], Dask [28], PyCOMPSs [30], and DagOn [23] employ a data-flow model for their workflows, and commonly represent them as a static Directed Acyclic Graph (DAG) [14]. Static in this regard means that, once the workflow is defined it is immutable, which does not address the need for scientific workflows to be dynamic. As highlighted by [7], [11], this possesses several constraints when the experiment at hand involves the exploration of large datasets.

Despite being referred to as 'static', many workflow systems currently allow for some degree of runtime adaption. For example, DVega [31] allows for exception handling in individual workflow steps. These exceptions are caught by the workflow system, which can recover by scheduling a new and potentially different step in its place. Other solutions in other workflow systems also use a similar method for workflow alteration at runtime, with steps replaced or rescheduled on different resources. Additionally, many existing workflow systems allow for whole workflows to be used as individual steps within a larger workflow. This is a common feature of workflow definitions such as CWL [3], and is demonstrated by the heterogeneous system presented in [16]. The presented system can make use of sub-workflows to loop several times through an in-situ workflow, which is in turn just one step in an end-to-end workflow run using PyCOMPSs [30]. Through techniques such as step swapping and sub-workflows, some dynamism is achieved, though the possible dynamic options must be defined ahead of time and so are limited to expected outcomes. It is suspected that this limitation comes from workflows still being defined in a static paradigm, with adaptions applied later within a model that does not sufficiently support such changes.

A fundamentally different approach would be to design workflows from the ground up to be dynamic. This would allow for easier implementation of recent innovations such as workflow loops, branching, or optional outputs. Requests [20] for human/experiment-in-the-loop workflows would also be suited to a dynamic model as it allows for a myriad of different inputs disparate in time, flow and nature. The proposed MEOW system is intended as such a system.

III. The Proposed Solution

A. Designing MEOW

To create a truly dynamic workflow a DAG cannot be used, or anything that can be reduced down to one before job scheduling. This is as such an expression of the workflow will, by necessity mean we understand the entire path through the workflow, and so have no need of a dynamic workflow. Instead, we will adopt the bottom up approach first demonstrated in [5] to create meaningfully distinct jobs, scheduled according to filesystem events. These jobs can be thought of as individual workflow steps, with their own input and output, and may even be workflows in their own right such as is demonstrated in [16]. However, each step is meaningfully distinct, in that they have no predetermined dependencies or links between them. As a result, a workflow becomes an emergent property of the system, rather than being explicitly defined by the user. This bottom up, event-driven approach shall be referred to as Managing Event-Oriented Workflows, or MEOW.

Within MEOW, users do not define workflows, but create individual steps out of *Patterns* and *Recipes*. Patterns are a description of what events should result in processing. This event description can be as broad or specific as the user requires. For instance, in an implementation based on file system events it could be a specific file path, or be broad enough to cover any instance of a certain file extension. The processing that is triggered by a Pattern is defined in a Recipe. A Recipe should process some input data and then should (but is not required to) produce some output. As well as the event description, a Pattern must also declare a Recipe as the processing that shall be triggered in the event of a match. Taken together, a Pattern and Recipe define one step of a workflow. As a user defines multiple Patterns and Recipes, a workflow emerges from the collection of individual steps.

The advantage of breaking down the workflows into these Patterns and Recipes is that each step of the workflow is now completely independent. This contrasts with other scientific workflow systems, where the entire workflow is processed together[2]. As a result, individual steps could be said to lack meaningful inter-dependencies and are scheduled and completed in isolation. This isolation is what enables the emergent workflow to be completely dynamic, as any job can be changed, cancelled or added regardless of the state of other jobs.

This level of job independence is novel in scientific workflow systems. It allows for some new possibilities in workflows, such as entirely optional job output, and is a better semantic fit for recent efforts in scientific workflows such as branching and looping. This is especially true if the track through said branches and loops is difficult to predict.

B. MEOW Requirements

To properly define a MEOW system, the following definitions for Patterns and Recipes have been constructed. These would be the minimum that would need to be defined in a Pattern or Recipe by a user for them to create a functioning system as described. For a Recipe the requirements are:

- **Name**: This is the unique identifier of the Recipe. It is used by Patterns to identify the linked Recipe, and by the implementation to keep track of changes to an already registered Recipe.
- **Instructions**: User defined code. For instance, a user's analysis algorithm. It may rely on input data or variables, provided by a Pattern.

The requirements for a Pattern are:

- **Name**: This is the unique identifier of the Pattern.
- **Triggering Event**: This is an event description, used to match against system events. In case of a match then a job should be scheduled according to the definition of the Pattern. This job will receive the event source as input.
- **Recipe**: The name of a Recipe, used to define the processing taking place in a job.
- **Variables**: A set of variables to be passed to the Recipe by this Pattern at job creation. These could be any data structure understood by the Recipe and may include additional input files or possible output locations.

Further requirements must be met for a MEOW based workflow system to be truly dynamic. These are:

[2]This may in practice be done in several batches of processing running in parallel, or sequentially. It is nevertheless defined as one holistic system.

- When the system registers a Pattern/Recipe, the system must check to see if any existing Patterns or Recipes require the new Pattern/Recipe. If so, they should be linked.
- Every time that a Pattern and Recipe are linked, the system must create an appropriate event trigger.
- If a new trigger is created, it must be able to check within the system, would any existing event sources activate the trigger, were they created now. If any matching event sources are present they must be treated as though they were just created and so activate the trigger.
- If a Pattern or Recipe is deleted, then any triggers created from it must be deleted.
- If a trigger is ever deleted then any jobs that were scheduled as a result of activating that trigger should be cancelled. This is not strictly necessary but without it the system may quickly bloat with outdated jobs and output, potentially confusing a user.
- If a Pattern/Recipe is ever updated then it should be processed in the system by the existing Pattern/Recipe being deleted, and a new one being registered as though it was created for the first time.

The presented requirements, when taken together ensure that the resulting workflow is adaptive to changes in Patterns, Recipes, and the underlying event sources. It ensures that jobs will be rescheduled automatically according to these changes. If each requirement is implemented correctly then a workflow will, by necessity, be an emergent property of Pattern and Recipe definitions.

IV. Implementing MEOW within MiG

A. mig_meow: A Package for Defining MEOW Workflows

A Python package was developed allowing users to define Patterns and Recipes, called mig_meow [19]. This package is based on a file event system and contains the definition of a Pattern object, along with a number of helper functions for defining a Pattern's event paths, Recipes, outputs[3], and variables.

Recipes are defined as Jupyter Notebooks [18]. This is as they are already commonly used in scientific computing, and can offer a user-friendly and interactive interface. According to the specification, Recipes require a Name and Instructions, which Notebooks already have with their filename and the code cells within them. Variables can be passed to a Recipe by a Pattern allowing the same Recipe to be used by multiple jobs/Patterns with different results, in the manner of a function or method.

To aid in the management of MEOW workflows, a Jupyter Notebook widget is provided as part of mig_meow, allowing for Pattern and Recipe construction. This widget also provides a visualisation of the emergent workflow from defined Patterns and Recipes. This is especially important due to the separated nature of the individual workflow steps, as they are not definitionally linked like in a traditional workflow. Having a method for checking that the outputs of one Pattern leads into the inputs of another is therefore helpful. Each defined input and output path is attached to a Pattern via arrows, showing the route along which data is processed. Where these inputs and outputs overlap they will point to the same location, such as in Figure 1. The visualisation also helps users identify potential loops in the workflow. As discussed in more detail in Section VI, loops are a useful feature of MEOW so are not discouraged, but an unintended one can cause a cascade of jobs. The visualisation can help identify these before they happen, but a secondary job monitoring widget can also be used to view, cancel and resubmit individual jobs scheduled from existing Patterns and Recipes. These widgets are intended as the primary means for a user to manage a MEOW workflow on the MiG, though the MiG's internal job status and feedback tools may also be used in conjunction if a user prefers.

A final contribution from mig_meow is a model workflow runner, which can run MEOW workflows locally on a user's machine. This is useful as a demonstration of the MEOW model, as a way of exploring the dynamic system, and of testing workflows locally. It is not however, intended as a deployment system and so shall not be considered in further detail. A fuller explanation can be found within mig_meow [22] if required.

B. mig_meow on the Minimum Intrusion Grid (MiG)

To both test and demonstrate how MEOW might be integrated into a mature scientific platform, an implementation was developed on the MiG [6]. This is a middleware grid oriented data management and processing platform. The MiG is a feature rich, production grade system serving researchers, students and external collaborators at the University of Copenhagen across a wide range of scientific fields. It was therefore deemed a good basis for the MEOW workflow implementation. The ability to provide runtime adaptable job scheduling was also recently introduced in [5]. While this notion is suitable for providing runtime adaptability, the initial implementation suffers from being highly complex in terms of being able to define individual trigger rules correctly and organising these independent rules to produce fully fledged workflows. To address this mig_meow is used to define Patterns and Recipes which can be exported to the MiG, whereupon the necessary triggers are created, updated or deleted as appropriate.

The heart of the MiG system from a user's perspective, is the concept of Virtual Grids (VGrids). These are a super-set of a typical grid organisation for storing, processing, sharing, and managing data in collaborative groups. The MiG also comes with a number of options to connect to it, or mount VGrid directories. This means that heterogeneous hardware such as user workstations or experiment instruments can instigate MEOW workflow events by uploading data to the MiG. This is an especially useful feature as modern scientific workflows are sought after that enable human/experiment-

[3]Note that a Pattern's outputs can be defined in mig_meow, despite this not being necessary according to MEOW. These do not have any effect other than aiding workflow visualisation, and actual outputs are not limited to those defined, nor are the defined outputs expected.

in-the-loop workflows, with decisions taken external to the workflow system.

V. A Worked Example

An example MEOW workflow is presented here. This example examines the size and distribution of the pores within an artificial dataset representing 3D X-ray computed tomography (CT) data of 100 samples of aluminium foam [29]. The goal is to analyse the pore radius distribution in all samples. Some samples have very few pores. We want to discard these samples and exclude them from the final analysis. This can be done effectively in a dynamic system, meaning we can setup the whole workflow at once and do not need to pre-sort our data sets as would often be required in a static system.

To perform this analysis, we need to segment the data, i.e. label the image voxels according to the two phases present; aluminium and air. Then, we identify the individual pores and estimate their radii. This is a time consuming process, so before we attempt these two steps we can perform an initial check to ensure that the porosity is within the desired range. This will exclude defect samples from the time consuming analysis.

1) The Recipes: The following items of processing were required before any workflow could be constructed. Each was written as a Jupyter Notebook and registered under the given name. Recipe code is available in [32].

- **Recipe 'porosity_check' linked to Pattern 'initial_porosity_check':** A two-component Gaussian Mixture Model is fitted to a small sample (around 1 %) of the intensity data, providing a rough idea of the air-to-aluminium ratio through the model component weights.
- **Recipe 'segmentation' linked to Pattern 'segment_foam_data':** In the first step of the segmentation process, noise is reduced using a Median filter. The filter kernel size is defined as a variable whose value is set in the Pattern. Thereafter, the image is segmented using Otsu thresholding [26]. Finally, a morphological closing operation is performed to remove possible remaining single-voxel noise.
- **Recipe 'pore_analysis' linked to Pattern 'foam_pore_analysis':** To investigate the pore size distribution, the individual pores are identified using the watershed algorithm [12] with local peaks in a distance transform of the segmented data as seeds.

2) The Foam Analysis Workflow: The final implementation of the workflow is illustrated in Figure 1. Each of the three created Patterns is shown in green circles. Each Pattern has as an input path a file type in a directory, as shown by the white rounded rectangles with arrows pointing to the Pattern. Any output locations are shown with the arrows leading out of the Pattern. Each Pattern specifies the corresponding Recipe, as stated in the previous paragraph.

100 artificial CT datasets to be analysed were generated using the Python package foam_ct_phantom [27] and the ASTRA toolbox [33]. 20 phantoms were generated with insufficient porosity compared to the remaining 80 phantoms. To start processing using the workflow, all 100 datasets were uploaded to the 'foam_ct_data' directory in the '.npy' NumPy array format. This triggers the first Pattern, 'initial_porosity_check'. The Recipe linked to this Pattern classifies each dataset as either "accepted" or "discarded" depending on some predefined porosity threshold, and accordingly, a text file with the dataset filename is created in one of the directories 'foam_ct_data_accepted' or 'foam_ct_data_discarded'. This was done to avoid copying the whole dataset needlessly, as this would result in gigabytes of additional space being used up. The creation of each text file triggers the next Pattern, 'segment_foam_data'. The segmentation method described in the linked Recipe is applied to the accepted datasets and the result stored in the directory 'foam_ct_data_segmented'. Finally, the pore analysis is performed on the segmented data, producing the final plots stored in the directory 'foam_ct_data_pore_statistics' as determined by the 'foam_pore_analysis' Pattern.

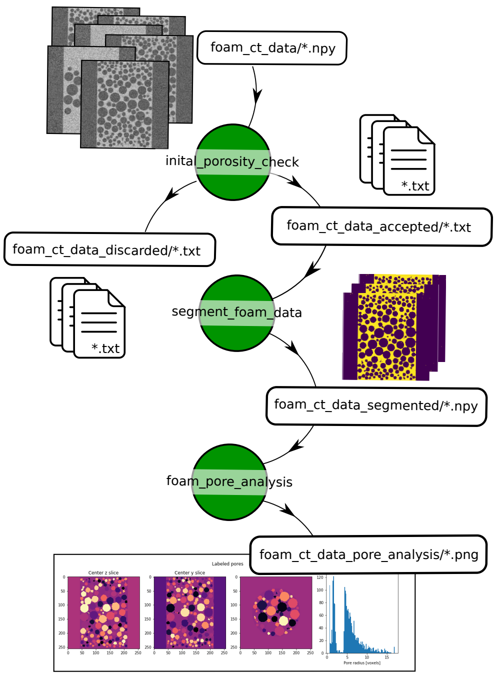


Fig. 1: The foam analysis workflow. Note that this image is based on the visualisation described in Section IV-A, but additional file images have been added to make the data state clearer at the different stages.

3) Results: All 100 datasets, totalling 6.25GB were generated and uploaded to the MiG. This triggered 100 jobs initially, and then a further 78 from the accepted data sets, and a final 78 from the segmentation. To complete all 256 resulting jobs took approximately 4 hours and 15 minutes using a 4 core, 3.4 GHz resource. 2 files of the original 100 were corrupted during their upload and so their initial jobs failed. Note that the processing of the other 98 jobs completed, demonstrating that the dynamic nature of this workflow can cope even when errors not predicted by the user occurs. Once all jobs were completed it was obvious that the results of 2 were missing from the final data and so the 2 files were re-uploaded. All 6 appropriate jobs were re-scheduled without having to re-run the entire workflow and all expected 80 final analysis images were present within 5 minutes of the corrected upload.

4) Suitability: The toy workflow presented here demonstrates how an event-driven workflow may be constructed, and a scenario in which it would be advantageous to do so. This is as we can setup one continuous workflow without any prior sorting of the data. In addition, individual job scheduling is completely separated so one dataset may be sorted, segmented and analysed before all datasets have been sorted. Lastly, as shown in the previous paragraph, the workflow is now an active part of the VGrid and is able to process any further data that is created or modified without any need of additional user interaction. This makes the analysis easily repeatable if further data is generated.

VI. Users and Use Cases

When using MEOW to define a dynamic workflow rather than a static workflow system, different design paradigms are possible. Some possibilities and use cases are presented here as inspiration. These are not intended as improvements per se over any other methodology, but merely as possibilities within MEOW that are either not possible in other systems or very difficult to achieve.

An event-driven, dynamic workflow system such as MEOW is ideally suited to repeatable jobs, especially if they have unpredictable outputs. This could be extremely useful for facilities such as CERN [8], MAX IV [21] or EuXFEL [13], where external users can book experiment time. These users are often an eclectic mix of specialists in their own field but may have limited technical understanding of complex computing environments such as high performance computing systems. For some users, their only concern is with the finally produced experiment data on which they shall perform their analysis. They are largely unconcerned with pre-processing tasks such as reformatting or cleaning data inputs. MEOW could be a valuable tool to ensure this is automatically scheduled before being handed off to data analysers.

Similarly, a dynamic workflow would be suited to continuous monitoring systems such as WIFIRE [2], a system for simulating, monitoring and predicting wildfires in southern California. WIFIRE uses Kepler, a static workflow system to create workflows where heterogeneous data from cameras, satellites, weather stations and previous data sets is used to predict/simulate fire risks. Notably, the output state of one run can be used as input for the next run so as to generate an updating and continuous fire risk simulation. If additional sensors are brought online, or existing ones removed, the workflow needs to be modifiable at runtime without the system needing extensive downtime for modification. This continuous, looping, dynamic nature maps well to a workflow system such as MEOW.

A dynamic system is also well suited to the increasing calls for human or experiment in-the-loop interactions within workflows. This is where workflow progression depends on decision making by either a user, or by hardware/software external to the workflow system. As MEOW functions use events, any external interaction can be enabled from a wide range of heterogeneous systems, as long as it can produce an event within a MEOW implementation.

An example of some of these use cases is presented in Figure 2. The specifics of the workflow will not be considered as it is the structure rather than the processing that is the focus here. In this workflow we can start with some experiment, E, such as an X-ray detector. This detector will produce potentially hundreds of data files in D1. Each file in D1 triggers a MEOW event, scheduling a job (purple) which assesses if the D1 data is worth keeping, e.g. as was shown in the 'initial_porosity_check' Pattern in Figure 1. The acceptable data is written to D2, whilst invalid data might trigger more experiment reads from E. These reads could be given new input parameters according to the results of the purple job. Writing to D2 will trigger 2 MEOW events. The first event schedules the green job which produces D3, and is analogous to the sort of processing shown in the 'segment_foam_data' and 'foam_pore_analysis' Patterns in Figure 1. The blue MEOW event schedules a job requiring some human interaction, such as to finely calibrate some input, or identifying particularly regions of interest. Any identified data is written to D4. This then triggers further, more in depth processing to D5 or could be the basis for more experiment reads.

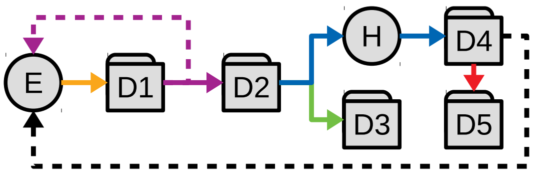


Fig. 2: An example workflow. Data directories and external actors are shown as nodes. Solid arrows are MEOW workflow jobs. Dotted arrows show calls for an experiment run.

VII. Limitations and Future Work

The presented workflow system has several limitations. The separated nature of individual jobs means that it is more difficult for job execution to be optimised, such as by compressing together directly linked jobs. On a related note, within the MiG, the choice of file based events means that keeping data in memory on job processing resources between jobs is also difficult to achieve. Both of these limitations are inherent to the current design of MEOW, and whilst solutions may be possible it is expected that they would be more difficult to achieve than in a static workflow system. Mitigating these limitations could be a significant avenue of future work.

Another limitation with the current implementation is a lack of user friendly provenance reporting. Reporting is currently available but it is hard to access, and is only done on a per job basis. A workflow wide report should connect linked jobs into common overview traceback reports that details what processing happened based on the defined Patterns and Recipes. A final point of future work would be to demonstrate a more complex, real world example of a MEOW workflow in action.

VIII. Conclusions

This paper has proposed one possible solution to some of the problems of modern scientific workflows. Unlike static workflows that require all steps to be defined at the very beginning, MEOW is proposed as a bottom up approach which breaks workflows down into Patterns and Recipes. This allows for an event-driven workflow system, which is fully dynamic at runtime. This enables a whole host of new ways of thinking about scientific workflows and how they are structured.

A demonstration system was implemented and described working in conjunction with the MiG. An example dynamic workflow using mig_meow was presented. The design repercussions of MEOW were also considered, with it being of particular note that the event-driven nature enables workflow structures for which traditional static workflow systems are unsuited. For example, optional branching and cyclic workflows are possible within this system.

This work is worth continuing as it presents scientists with a dynamic paradigm for workflows, enabling new ways of interacting with and exploring even extremely large datasets.

References

[1] Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludascher, B., Mock, S.: Kepler: An Extensible System for Design and Execution of Scientific Workflows. In: Proceedings. 16th International Conference on Scientific and Statistical Database Management, 2004. IEEE (2014)

[2] Altintas, I., Block, J., de Callafon, R., Crawl, D., Cowart, C., Gupta, A., Nguyen, M., Braun, H.W., Schulze, J., Gollner, M., Trouve, A., Smarr, L.: Towards an Integrated Cyberinfrastructure for Scalable Data-driven Monitoring, Dynamic Prediction and Resilience of Wildfires. Procedia Computer Science **51**, 1633 – 1642 (2015), international Conference On Computational Science, ICCS 2015

[3] Amstutz, P., Crusoe, M.R., Tijanić, N., Chapman, B., Chilton, J., Heuer, M., Kartashov, A., Leehr, D., Ménager, H., Nedeljkovich, M., et al.: Common workflow language, v1.0 (Jul 2016). https://doi.org/10.6084/m9.figshare.3115156.v2, https://figshare.com/articles/Common_Workflow_Language_draft_3/3115156/2

[4] Apache Airflow documentation. https://airflow.apache.org/docs/stable/ (2020)

[5] Bardino, J., Rehr, M., Vinter, B.: Event-driven, collaborative and adaptive scientific workflows on the grid. Communicating Process Architecture pp. 59–78 (2017)

[6] Berthold, J., Bardino, J., Vinter, B.: A principled approach to grid middleware: Status report on the minimum intrusion grid. In: Xiang, Y., Cuzzocrea, A., Hobbs, M., Zhou, W. (eds.) Algorithms and Architectures for Parallel Processing, pp. 409–418. Springer (2011)

[7] Caeiro Rodriguez, M., Priol, T., Nemeth, Z.: Dynamicity in scientific workflows. Institute on Grid Information, Resource and Workflow Monitoring Services, CoreGRID-Network of Excellence, Tech. Rep. TR-0162, August (01 2008)

[8] CERN. https://home.cern (2020)

[9] Deelman, E., Blythe, J., Gil, Y., Kesselman, C.: Pegasus: Planning for execution in grids. Tech. Rep. Technical Report 2002-20, GriPhyN (2002), http://pegasus.isi.edu/publications/ewa/pegasus_overview.pdf

[10] Deelman, E., Gannon, D., Shields, M.S.: Workflows for e-Science. Workflows for e-Science (2007). https://doi.org/10.1007/978-1-84628-757-2

[11] Dias, J., Guerra, G., Rochinha, F., Coutinho, A.L., Valduriez, P., Mattoso, M.: Data-centric iteration in dynamic workflows. Future Generation Computer Systems **46**, 114–126 (2015)

[12] Digabel, H., Lantuéjoul, C.: Iterative algorithms. In: Proc. 2nd European Symp. Quantitative Analysis of Microstructures in Material Science, Biology and Medicine. vol. 19, p. 8. Stuttgart, West Germany: Riederer Verlag (1978)

[13] EuXFEL. https://www.xfel.eu (2020)

[14] Fakhfakh, F., Kacem, H.H., Kacem, A.H.: Workflow scheduling in cloud computing: A survey. In: 2014 IEEE 18th International Enterprise Distributed Object Computing Conference Workshops and Demonstrations. pp. 372–378 (Sep 2014). https://doi.org/10.1109/EDOCW.2014.61

[15] Foster, I.: Globus Toolkit Version 4: Software for Service-Oriented Systems. In: IFIP International Conference on Network and Parallel Computing. pp. 2–13. Springer-Verlag (2006)

[16] amd Francesc Lordan, C.R.C., Ejarque, J., Badia, R.M.: A programming model for hybrid workflows: Combining task-based workflows and dataflows all-in-one. Future Generation Computer Systems **113**, 281–297 (2020)

[17] Gil, Y., Deelman, E., Ellismand, M., Fahringer, T., Fox, G., Gannon, D., Goble, C., Livny, M., Moreau, L., Myers, J.: Examining the challenges of scientific workflows. IEEE Computer **40**(12), 24–32 (2007)

[18] Project Jupyter. http://jupyter.org (2020)

[19] Marchant, D.: mig_meow. https://pypi.org/project/mig-meow (2020)

[20] Mattoso, M., Dias, J., A.C.S.Ocaña, K., Ogasawara, E., Costa, F., Horta, F., Silva, V., de Oliviera, D.: Dynamic steering of HPC scientific workflows: A survey. Future Generation Computer Systems **46**, 100–113 (2015)

[21] MAX IV. https://www.maxiv.lu.se (2020)

[22] mig_meow examples. https://tinyurl.com/y687mfvn (2020)

[23] Montella, R., Di Luccio, D., Kosta, S.: DagOn*: Executing Direct Acyclic Graphs as Parallel Jobs on Anything. In: Proceedings of WORKS 2018: 13th Workshop on Workflows in Support of Large-Scale Science, Held in conjunction with SC 2018: The International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 64–73 (2019). https://doi.org/10.1109/WORKS.2018.00012

[24] Mummering. http://www.mummering.eu (2020)

[25] Oinn, T., Addis, M., Ferris, J., Marvin, D., Senger, M., Greenwood, M., Carver, T., Glover, K., Pocock, M.R., Wipat, A., Li, P.: Taverna: a tool for the composition and enactment of bioinformatics workflows. Bioinformatics **20**(17), 3045–3054 (2004)

[26] Otsu, N.: A threshold selection method from gray-level histograms. IEEE transactions on systems, man, and cybernetics **9**(1), 62–66 (1979)

[27] Pelt, D.M., Batenburg, K.J., Sethian, J.A.: Improving tomographic reconstruction from limited data using mixed-scale dense convolutional neural networks. Journal of Imaging **4**(11), 128

[28] Rocklin, M.: Dask: Parallel computation with blocked algorithms and task scheduling. In: Proceedings of the 14th python in science conference. pp. 126–132 (01 2015). https://doi.org/10.25080/Majora-7b98e3ed-013

[29] Simancik, F., Jerz, J., Kovacik, J., Minar, P.: Aluminium foam-a new light-weight structural material. METALLIC MATERIALS **35**, 187–194 (1997)

[30] Tejedor, E., Becerra, Y., Alomar, G., Queralt, A., Badia, R.M., Torres, J., Cortes, T., Labarta, J.: PyCOMPSs: Parallel computational workflows in Python. International Journal of High Performance Computing Applications **31**(1), 66–82 (2017). https://doi.org/10.1177/1094342015594678

[31] Tolosana-Calasanz, R., Bañares, J.A., Rana, O.F., Álvarez, P., Ezpeleta, J., Hoheisel, A.: Adaptive exception handling for scientific workflows. Concurrency and Computation: Practice and Experience **22**(5), 617–642 (2010). https://doi.org/10.1002/cpe.1487, https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1487

[32] Use case code. https://sid.idmc.dk/sharelink/a2Fki4iVbA (2020)

[33] Van Aarle, W., Palenstijn, W.J., De Beenhouwer, J., Altantzis, T., Bals, S., Batenburg, K.J., Sijbers, J.: The astra toolbox: A platform for advanced algorithm development in electron tomography. Ultramicroscopy **157**, 35–47 (2015)

A.4 Cloud enabling educational platforms with corc

Munk, Rasmus; David, Marchant; Vinter, Brian. Proceedings of the 8th Workshop on Cloud Technologies in Education (CTE2020)

Cloud enabling educational platforms with corc

Rasmus Munk[a], David Marchant[b] and Brian Vinter[c]

[a]*Niels Bohr Institute, Blegdamsvej 17, Copenhagen, 2100, Denmark*
[b]*Niels Bohr Institute, Blegdamsvej 17, Copenhagen, 2100, Denmark*
[c]*Aarhus University, Ny Munkegade 120, Aarhus C, 8000, Denmark*

Abstract

In this paper, it is shown how teaching platforms at educational institutions can utilize cloud platforms to scale a particular service, or gain access to compute instances with accelerator capability such as GPUs.

Specifically at the University of Copenhagen (UCPH), it is demonstrated how the internal JupyterHub service, named Data Analysis Gateway (DAG), could utilize compute resources in the Oracle Cloud Infrastructure (OCI). This is achieved by utilizing the introduced Cloud Orchestrator (corc) framework, in conjunction with the novel JupyterHub spawner named MultipleSpawner. Through this combination, we are able to dynamically orchestrate, authenticate, configure, and access interactive Jupyter Notebooks in the OCI with user defined hardware capabilities. These capabilities include settings such as the minimum amount of CPU cores, memory and GPUs the particular orchestrated resources must have. This enables teachers and students at educational institutions such as UCPH to gain easy access to the required capabilities for a particular course. In addition, we lay out how this groundwork, will enable us to establish a Grid of Clouds between multiple trusted institutions. This enables the exchange of surplus computational resources that could be employed across their organisational boundaries.

Keywords

Teaching, Cloud Computing, Grid of Clouds, Jupyter Notebook

1. Introduction

The availability of required computational resources in organisations, such as scientific or educational institutions, is a crucial aspect of delivering the best scientific research and teaching. When teaching courses involving data analysis techniques it can be beneficial to have access to specialized platforms, such as GPU accelerated architectures.

At higher educational institutions, such as the University of Copenhagen (UCPH) or Lund University (LU), these centers are substantial investments, that are continuously maintained and upgraded. However, the usage of these resources often varies wildly between being fully utilized to sitting idly by.

We therefore propose, that these institutional resources be made available (with varying priority) across trusted educational and scientific organisations. Foremost, this is to enable the voluntary sharing of underused resources to other institutions, thereby potential establishing greater scalability than can be found within each individual institution.

CTE 2020: 8th Workshop on Cloud Technologies in Education, December 18, 2020, Kryvyi Rih, Ukraine
EMAIL: rasmus.munk@nbi.ku.dk (R. Munk); d.marchant@ed-alumni.net (D. Marchant); vinter@au.dk (B. Vinter)
URL:
https://research.ku.dk/search/result/?pure=en%2Fpersons%2Frasmus-munk(a72145a7-0203-4791-bb8f-6073bc23fe2f).html (R. Munk); https://research.ku.dk/search/result/?pure=en%2Fpersons%2Fdavid-gray-marchant(ff6af890-33df-4414-9a9d-c3d33258ad1f).html (D. Marchant); https://pure.au.dk/portal/en/persons/brian-vinter(a4fe861f-5a04-4e93-a5b3-c633045eb82e).html (B. Vinter)
ORCID: 0000-0003-0333-4295 (R. Munk); 0000-0003-4262-7138 (D. Marchant); 0000-0002-3947-9878 (B. Vinter)

1.1. Basic IT

Within institutions such as UCPH, there is a mixture of services that each provides. At the very basic level, there are infrastructure services such as networking, account management, email, video conferencing, payroll management, license management, as well OS and software provisioning. In this paper, we define these as Basic IT services. At educational institutions, additional services can be added to this list, these include services for handling student enrollment, submissions, grading, course management, and forum discussions. As with the initial Basic IT services, these are typically off the shelf products that needs to be procured, installed, configured and maintained on a continuous basis.

A distinguishing trait of Basic IT services, in an education context, is that they are very predictable in terms of the load they will exhibit, both in times of high and low demand. For instance, there will be busy junctions, such as assignment hand in days, release of grades, student enrollment, and so on. In contrast, holiday and inter-semester periods will likely experience minor to no usage. Given this, these services are classic examples of what cloud computing was developed to provide. Efficient utilization of on-demand resources, with high availability and scalability to handle fluctuating usage in a cost effective manner.

1.2. Science IT

Science IT services, in contrast, revolve around the institutions scientific activities whether by researchers or students. They include services such as management, sharing, transferring, archiving, publishing, and processing of data, in order to facilitate the scientific process. In addition, these facilities also enable lecturers to utilize their research material in courses, giving students access to the same platform and resources.

What distinguishes these services, is that they impose different constraints compared to Basic IT services. These typically involve areas such as, computational load, security, budgetary, scientific, and legal requirements, among others. For example, it is often too inefficient, or costly to utilize public cloud resources for the storing and processing of large scientific datasets at the petabyte scale. In this case, a more traditional approach such as institutional compute resources is required. [1].

Research fields such as climate science [2], oceanography [3], and astronomy [4], often employ experimental simulations as a common scientific tool. These simulations produce output up to petabytes in size, that still need to be stored for subsequent postprocessing and analysis. Upon a scientific discovery from this process, the resulting datasets needs to be archived in accordance with regulatory requirements, which in the case of UCPH is 5 years [5] (only available in Danish).

1.3. Institutional Resources

High Performance Computing (HPC) and regular compute centers are often established at higher educational institutions to provide Science IT services. The UCPH [6], University of Antwerp [7], and LU [8] compute centers are examples of this. In addition, institutions can also gain access to similar resources through joint facilities like the Vienna Scientific Cluster [9], which supports 19 institutions, 10 of which are higher educational institutions. Finally there are national and pan-national resources such as ARCHER (UK) [10] or the EuroHPC [11] that review applications before access is granted.

These established centers are very expensive to build and have a limited lifespan before they need to be replaced. Even smaller educational compute platforms follow a similar life-cycle. For instance, at the UCPH a typical machine has a lifetime of 5 years before it needs to be replaced. This is whether the machine has been heavily utilized or not. Therefore, it is important that these systems across institutions are utilized, not only efficiently, but at maximum capacity throughout their lifetime.

For organising the sharing of resources across trusted educational and scientific organisations, inspiration is drawn from the way traditional computational Grids have been established [12]. The difference is, that instead of establishing a Grid where individual resources are attached, this model will instead be based on each institution establishing a Cloud of resources that are shared via a Grid. This means that the Grid is responsible for interconnecting disjointed clouds, whether they be institutional or public cloud platforms. The result being an established model for sharing cloud resources across educational institutions in support of cloud services for bachelor and master courses, general workshops, seminars and scientific research.

In this paper, we present how an existing teaching and research service at UCPH could be enabled with access to a cloud framework, which is the first step towards a Grid of Clouds resources. We accomplish this by using the Cloud Orchestrator (corc) framework [13]. Through this, we are able to empower the DAG service with previously inaccessible compute resources across every course at UCPH. This was previously not feasible with internal resources alone. Since we do not have access to other institutional resources at this point in time, we utilized a public cloud provider to scale the service with external resources.

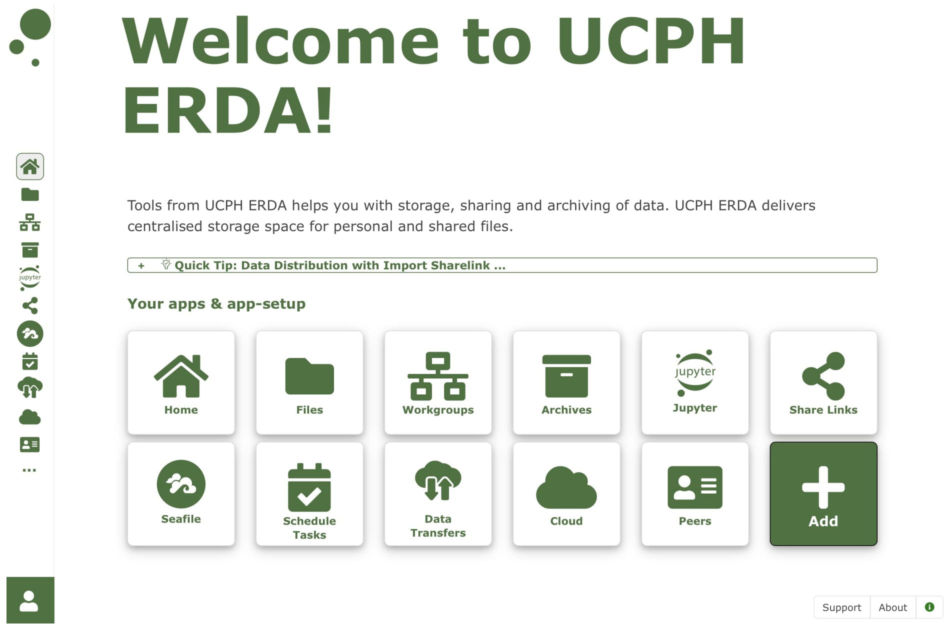
2. Background

At the Niels Bohr Institute (NBI), part of UCPH, we host a number of Science IT services that are part of providing a holistic educational platform for researchers, teachers, students, and general staff. A subset of these Science IT services have been especially beneficial across all levels of teaching. Namely, services such as the University Learning Management System (LMS), called Absalon, which is based on Canvas [14] for submissions and grading. The Electronic Research Data Archive (ERDA) [15] for data management and sharing tasks. In addition to the Data Analysis Gateway (DAG) [16], which is a JupyterHub powered platform for interactive programming and data processing in preconfigured environments.

2.1. Teaching Platforms

The combination of these subset services, in particular the combination of ERDA and DAG, has been especially successful. Teachers have used these to distribute course material through ERDA, which made the materials available for students to work on at the outset of the course. This ensures that students can get on with the actual learning outcomes from the get go, and not spend time on tedious tasks such as installing prerequisite software for a particular course. Due to budgetary limitations, we have only been able to host the DAG service with standard servers, that don't give access to any accelerated architectures.

Across education institutions, courses in general have varying requirements in terms of computing resources, environments, and data management, as defined by the learning outcomes of the course. The requirements from computer science, data analysis, and physics oriented courses are many, and often involve specialized compute platforms. For example, novel data analysis techniques, such as Machine Learning or Deep Learning have been employed across a wide range of scientific fields. What is distinct about these techniques is the importance of the underlying compute platform on which it is being executed. Parallel architectures such as GPUs in particular are beneficial in this regard, specifically since the amount of independent linear systems that typically needs to be calculated to


Welcome to UCPH ERDA!

Tools from UCPH ERDA helps you with storage, sharing and archiving of data. UCPH ERDA delivers centralised storage space for personal and shared files.

+ Quick Tip: Data Distribution with Import Sharelink ...

Your apps & app-setup

Figure 1: ERDA Interface

give adequate and reliably answers are immense. The inherent independence of these calculations, makes them suitable for being performed in parallel, making it hugely beneficial to utilize GPUs. [17].

Given that the DAG service was an established service at UCPH for data analysing and programming in teaching bachelor and master students, it seemed the ideal candidate to enable with access to cloud resources with accelerator technology. For instance, courses such as Introduction to Computing for Physicists (abbreviated to DATF in Danish) [18], Applied Statistics: From Data to Results (APP-STAT) [19], and High Performance Parallel Computing (HPPC) [20], all would benefit from having access to GPU accelerators to solve several of the practical exercises and hand-in assignments.

2.2. ERDA

ERDA provides a web based data management platform across UCPH with a primary focus on the Faculty of Science. Its primary role is to be a data repository for all employees and students across UCPH. Through a simple web UI powered by a combination of an Apache webserver and a Python based backend, users are able to either interact with the different services through its navigation menu, or a user's individual files and folders via its file manager. An example of the interface can be seen in Figure 1. The platform itself is a UCPH-specific version of the open source Minimum Intrusion Grid (MiG) [21], that provides multiple data management functionalities. These functionalities includes easy and secure upload of datasets, simple access mechanisms through a web file manager, and the ability to establish collaboration and data sharing between users through Workgroups.

2.3. Jupyter

Project Jupyter [22] develops a variety of open source tools. These tools aim at supporting interactive data science, and scientific computing in general. The foundation of these is the IPython Notebook (.ipynb) format (evolved out of the IPython Project [23]). This format is based on interpreting special segments of a JSON document as source code, which can be executed by a custom programming language runtime environment, also known as a kernel. The JupyterLab [24] interface (as shown in Figure 2) is the standard web interface for interacting with the underlying notebooks. JupyterHub [25] is the de-facto standard to enable multiple users to utilize the same compute resources for individual Jupyter Notebook/Lab sessions. It does this through its own web interface gateway and backend database, to segment and register individual users before allowing them to start a Jupyter session.

In addition, JupyterHub allows for the extension of both custom Spawners and Authenticators, enabling 3rd party implementations. The Authenticator is in charge of validating that a particular request is from an authentic user. The responsibility of the Spawner is how a Jupyter session is to be scheduled on a resource. Currently there exist only static Spawners that utilize either preconfigured resources that have been deployed via Batch, or Container Spawners, or at selective cloud providers such as AWS [26]. As an exception to this, the WrapSpawner [27] allows for dynamic user selections through predefined provides. However, these profiles cannot be changed after the JupyterHub service is launched, making it impossible to dynamically change the set of supported resources and providers. Therefore it would be of benefit if a Spawner extended the WrapSpawner's existing capabilities with the ability to dynamically add or remove providers and resources.

3. Related Work

As presented in [28], Web-based learning by utilizing cloud services and platforms as part of the curriculum is not only feasible, but advisable. In particular, when it comes to courses with programming activities for students, educational institutions should enable access to innovative Web-based technologies that supports their learning. These include interactive programming, version control and automated programming assessments to ensure instant feedback.

3.1. Interactive Programming Portals

Research in cloud computing for education typically revolves around using Web-enabled Software as a Service (SaaS) applications. Examples of such include platforms such as GitHub [29], Google Docs [30], Google Colaboratory [31], Kaggle [32], and Binder [33]. Each of these can fill a particular niche in a course at the teacher's or student's discretion. Nevertheless, the provided capability often does come with its own burdens, in that the administration of the service is often left to the teaching team responsible for the course. This responsibility typically includes establishing student access, course material distribution to the specific platform, guides on how to get started with the service and solving eventual problems related to the service throughout the course. In addition, many of the external cloud services that offer free usage, often have certain limitations, such as how much instance utilisation a given user can consume in a given time span. Instead, providing such functionalities as Science IT services, could reduce these overheads and enable seamless integration into the courses. Furthermore, existing resources could be used to serve the service by scaling through an established Grid of Clouds.

In terms of existing public cloud platforms that can provide Jupyter Notebook experiences, DAG is similar to Google Colaboratory, Binder, Kaggle, Azure Notebooks [34], CoCalc [35], and Datalore

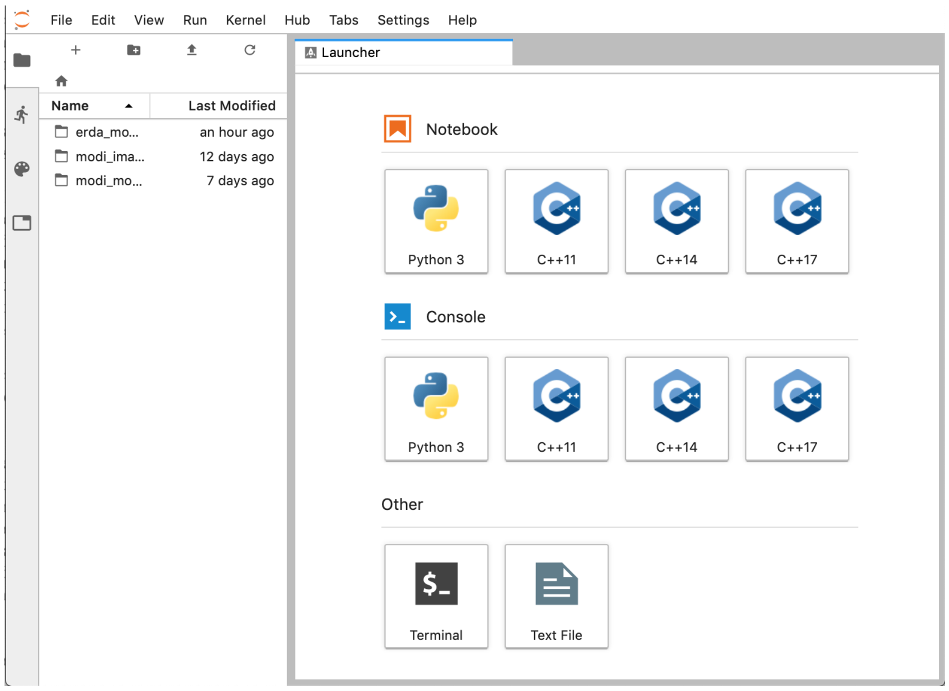


Figure 2: JupyterLab Interface

[36]. All of these online options, have the following in common. They all have free tier plans available with certain hardware and usage limitations. All are run entirely in the web browser and don't require anything to be installed locally. At most they require a valid account to get started. Each of them present a Jupyter Notebook or Notebook like interface, which allows for both export and import of Notebooks in the standard format. An overview of a subset of the supported features and usage limits across these platforms can be seen in Table 1, and their hardware capabilities in Table 2. From looking at the features, each provider is fairly similar in terms of enabling Languages, Collaborating, and Native Persistence (i.e. the ability to keep data after the session has ended). However, there is a noticeable difference, in the maximum time (MaxTime) that each provider allows a given session to be inactive before it is stopped. With CoCalc being the most generous, allowing 24 hours of activity before termination. In contrast, internal hosted services such as DAG allow for the institution to define this policy. At UCPH, we have defined this to be 2 hours of inactivity, and an unlimited amount of active time for an individual session. However, as Table 2 shows, we currently don't provide any GPU capability, which is something that could be changed through the utilisation of an external cloud with GPU powered compute resources.

Given this, the DAG service seemed as the ideal candidate to empower with external cloud resources. Both because it provides similar features as the public cloud providers in terms of Languages and Collaborate ability, but also since it is integrated directly with UCPHs data management service.

Table 1
Subset of Jupyter Cloud Platforms Features

Provider	Native Persistence	Languages	Collaborate	MaxTime (inactive,max)
Binder[37]	None	User specified [1]	Git	10m, 12h[2]
Kaggle [38]	Kaggle Datasets	Python3,R	Yes	60m, 9h
Google Colab [39]	GDrive, GCloud Storage	Python3,R	Yes	60m,12h* [3]
Azure Notebooks [40] [41]	Azure Libraries	Python{2,3},R,F#	NA	60m,8h* [4]
CoCalc [42]	CoCalc Project	Python{2,3},R,Julia,etc	Yes*	30m, 24h
Datalore [43]	Per Workbook	Python3	Yes	60m, 120h [5]
DAG [44]	ERDA	Python2,3,R,C++,etc	Yes	2h, unlimited [6]

Table 2
Hardware available on Jupyter Cloud Platforms

Provider	CPU	Memory (GB)	Disk Size (GB)	Accelerators
Binder	NA	1 Min, 2 MAX	No specified limit*	None
Kaggle1	4 cores	17	5	None
Kaggle2	2 cores	14	5	GPU [7] or TPU [8] [45]
Google Colab Free	NA	NA	GDrive 15	GPU or TPU (thresholded access)
Azure Notebooks (per project)	NA	4	1	GPU (Pay)
Cocalc (per project)	1 shared core	1 shared	3	None
Datalore	2 cores	4	10	None
DAG	8 cores	8	unlimited [9]	None

3.2. Cloud Orchestration

Cloud resources are typically provided by the infrastructure service through some form of orchestration. Orchestration is a term for providing an automated method to configure, manage and coordinate computer systems [46]. Through orchestration, an organisation or individual is able to establish a complex infrastructure through a well defined workflow. For instance, the successful creation of a compute node involves the processing of a series of complex tasks that all must succeed. An example of such a workflow can be seen in Figure 3. Here a valid Image, Shape, Location and Network has to be discovered, selected, and successfully utilized together in order for the cloud compute node to be established. An Image is the target operating system and distribution, for instance Ubuntu 20.04 LTS. A Shape is the physical configuration of the node, typically involving the amount of CPU cores, memory and potential accelerators. Location is typically the physical location of where the resource is to be created. Cloud providers often use the term Availability Zone instead but it generally defines which datacenter to utilize for the given task. Network encompasses the entirety of the underlying network configuration, including which Subnet, Gateway, and IP address the compute node should utilize. In the context of a federated network like a Grid, the orchestration would ideally involve the automated provisioning of the computational resource, the configuration of said resource, and ensure that the resource is correctly reachable through a network infrastructure.

Multiple projects have been developed that automate development and system administration tasks such as maintenance, testing, upgrading, and configuration. These includes packages such as TerraForm [47], Puppet [48], Chef [49], and Ansible [50], all of which open source projects that can be utilized across a range of supported cloud providers. Nevertheless, in terms of enabling workflows that can provide orchestration capabilities, these tools are limited in that they typically only focuses on a

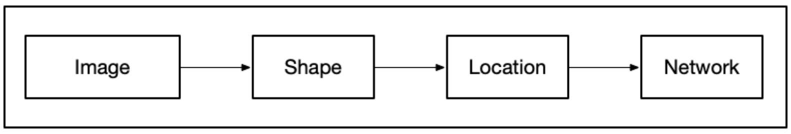


Figure 3: Workflow for orchestrating a compute node

subset of the orchestration functionalities such as provisioning and deployment or configuration and maintenance. For instance TerraFrom is a tool that focuses on infrastructure deployment whereas Puppet, Chef and Ansible are primarily concerned with configuration and maintenance of existing systems. In contrast commercial cloud providers typically also provide their own orchestration-like tools and Software Development Kits (SDK)s, enabling the ability to interact with their respective cloud system. For instance, Oracle provides the Oracle Cloud Infrastructure CLI [51] tool that can interact with their infrastructure. The same applies to the Amazon AWS CLI [52], in addition to a vast complement of tool-kits [53] that provide many different AWS functionalities including orchestration. In contrast, commercial cloud provided tools are often limited to only support the publishing cloud vendor and do not offer cross-cloud compatibility, or the ability to utilize multiple cloud providers interchangeably.

Cloud orchestration developments for the scientific community, especially those aiming to provide cross-cloud deployments, have mostly been based on utilizing on premise cloud IaaS platforms such as OpenStack [54] and OpenNebula [55]. Developments have focused on providing higher layers of abstraction to expose a common APIs that allow for the interchangeable usage of the underlying supported IaaS platforms. The infrastructure is typically defined in these frameworks through a Domain Specific Language (DSL) that describes how the infrastructure should look when orchestrated. Examples of this include cloud projects such as INDIGO-cloud [56] [57], AgroDAT [58] and Occupus [58]. These frameworks, nonetheless do not allow for the utilization of commercial or public cloud platforms, since they rely on the utilization of organisationally defined clouds that are traditionally deployed, managed, and hosted by the organisation itself. Although required, if as stated, we are to establish a Grid of Clouds which should allow for the inclusion of public and commercial cloud platforms. The corc framework was developed and designed to eventually support the scheduling of cloud resources across both organisations and public cloud providers.

4. The first cloud enabled service

To establish a Grid of Cloud resources, we started with enabling the usage of a single public cloud provider to schedule DAG Notebooks on. Through this we created the foundations for the eventual Grid structure that would allow the resources to be scheduled across multiple clouds and organisations.

4.1. Corc

The corc framework was implemented as a Python package. The package establishes the foundations for essential functions such as orchestration, computation, configuration, and authentication against supported cloud providers and cloud resources. Overall, corc is a combination of an Infrastructure as a Service (IaaS) management library, and a computation oriented scheduler. This enables the ability

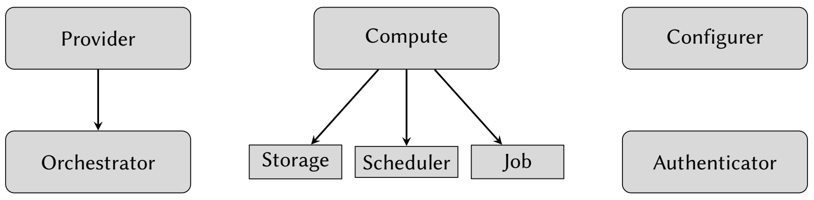


Figure 4: Cloud Orchestrator Framework Overview

to schedule services on a given orchestrated resource. An overview of the architecture can be seen in Figure 4.1.

The first provider to be integrated into the framework was the OCI IaaS. This was chosen, because the UCPH had a preexisting collaboration with Oracle, that enabled the usage of donated cloud resources for testing and development. As also highlighted, this does not limit the integration of other cloud providers into the framework, which the framework was designed for. Furthermore, as explored in section 2.3. A new Spawner, named MultipleSpawner was introduced, to provide the necessary dynamic selection of cloud providers.

As Figure 4.1 indicates, for each provider that corc supports, an orchestrator for that provider needs to be defined within corc. In addition, the framework defines three other top level components, namely Compute, Configurer, and Authenticator. All three are abstract definitions allowing for specific implementations to support the targeted resources which they apply to. A service can therefore be enabled with the ability to utilize cloud resources by integrating the corc components into the service itself. This method is limited to services that are developed in Python. In addition, corc also defines a Command Line Interface (CLI), that can be used to interact with the cloud provided resources directly. Details about how the framework and CLI can be used will not be presented in this paper, but can be found in [13].

```
{
    "virtual_machine": [
        {
            "name": "oracle_linux_7_8",
            "provider": "oci",
            "image": "Oracle Linux 7.8"
        }
    ]
}
```

Listing 1: Spawner Deployment configuration

4.2. MultipleSpawner

MultipleSpawner [59] is a Python package allowing for the selection of dynamic Spawners and resources. Structurally, it is inspired by the WrapSpawner [27], through the MultipleSpawner integrates corc into the Spawner ifself. This enables the JupyterHub service to manage and utilize cloud resources on a dynamic set of providers. In order to enable the MultipleSpawner to support these dynamic resources providers, two JSON configuration files needs to be defined. One of these is shown in Listing 1, and defines the specific resource type that should be deployed on the provider. Currently the MultipleSpawner supports deploying, ‘virtual_machine‘, ‘container‘, and ‘bare_metal‘ resources. The other configuration file is shown in Listing 2. It defines the template configuration settings that specify which Spawner, Configurer, and Authenticator the MultipleSpawner should use to spawn, configure and connect to the deployed resource.

```
[
    {
        "name": "VirtualMachine Spawner",
        "resource_type": "virtual_machine",
        "providers": ["oci"],
        "spawner": {
            "class": "sshspawner.sshspawner.SSHSpawner",
            "kwargs": {
                "remote_hosts": ["{endpoint}"],
                "remote_port": "22",
                "ssh_keyfile": "~/.corc/ssh/id_rsa",
                "remote_port_command": "/usr/bin/python3
                /usr/local/bin/get_port.py"
            }
        },
        "configurer": {
            "class": "corc.configurer.AnsibleConfigurer",
            "options": {
                "host_variables": {
                    "ansible_user": "opc",
                    "ansible_become": "yes",
                    "ansible_become_method": "sudo",
                    "new_username": "{JUPYTERHUB_USER}"
                },
                "host_settings": {
                    "group": "compute",
                    "port": "22"
                },
                "apply_kwargs": {
                    "playbook_path": "setup_ssh_spawner.yml"
                }
            }
        },
        "authenticator": {
            "class": "corc.authenticator.SSHAuthenticator",
            "kwargs": {"create_certificate": "True"}
        }
    },
]
```

Listing 2: Spawner Template configuration

5. Results

By integrating corc into the MultipleSpawner, we enabled the architecture shown in Figure 5, where the DAG service is able to dynamically schedule Jupyter Notebooks across the two resource providers. As is indicated by Figure 5, the UCPH and OCI providers are defined to orchestrate resources, in this case cloud compute instances, in preparation for scheduling a requested Notebook. In order to validate that the architecture worked as expected, we setup a test environment on a separate machine. This machine was configured with a corc and JupyterHub environment, where OCI was defined as a corc provider and the MultipleSpawner as the designated JupyterHub Spawner. With this in order, the JupyterHub service was ready to be launched on the machine.

The MultipleSpawner was configured to use the template and deployment settings defined in Listing 1 and 2. This enables the MultipleSpawner to create Virtual Machine cloud resources at the OCI. Subsequently, the MultipleSpawner uses the SSHSpawner [60] created by the National Energy Research Scientific Computing (NERSC) Center to connect and launch the Notebook on the orchestrated resource. Prior to this, it uses the corc defined SSHAuthenticator and AnsibleConfigurer to ensure that the MultipleSpawner can connect to a particular spawned resource and subsequently configure it with the necessary dependencies.

An example of a such a spawn with the specified requirements can be seen in Figure 6. To validate that this resource had been correctly orchestrated, the corc CLI was utilized to fetch the current allocated resources on OCI. Listing 3 shows that an instance with 12 oracle CPUs, 72 GB of memory and one NVIDIA P100 GPU had been orchestrated. This reflects the minimum shape that could be found in the EU-FRANKFURT-1-AD-2 availability domain that met the GPU requirement.

```
rasmusmunk$ corc oci orchestration instance list
{
    "instances": [
    {
        ...
        "availability_domain": "lfcb:EU-FRANKFURT-1-AD-2",
        "display_name": "instance20201018103638",
        "image_id": "ocid1.image.oc1.eu-frankfurt....",
        "shape": "VM.GPU2.1",
        "shape_config": {
            ...
            "gpus": 1,
            "max_vnic_attachments": 12,
            "memory_in_gbs": 72.0,
            "ocpus": 12.0,
        },
    }
    ],
    "status": "success"
}
```

Listing 3: Running OCI Notebook Instance

As shown in Figure 7, the JupyterHub spawn action redirected the Web interface to the hosted Notebook on the cloud resources. Relating this to the mentioned courses at UCPH, this then enabled the students with access to an interactive programming environment via the JupyterLab interface.

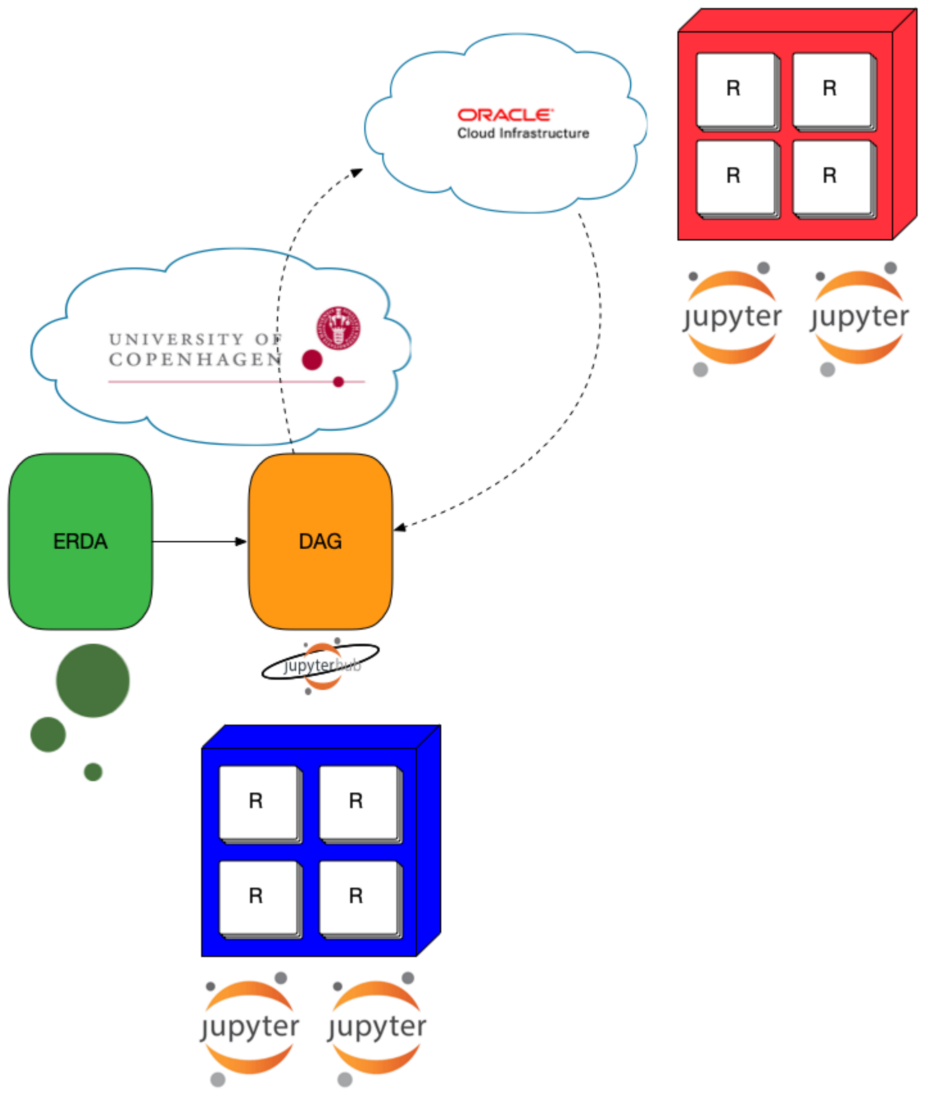
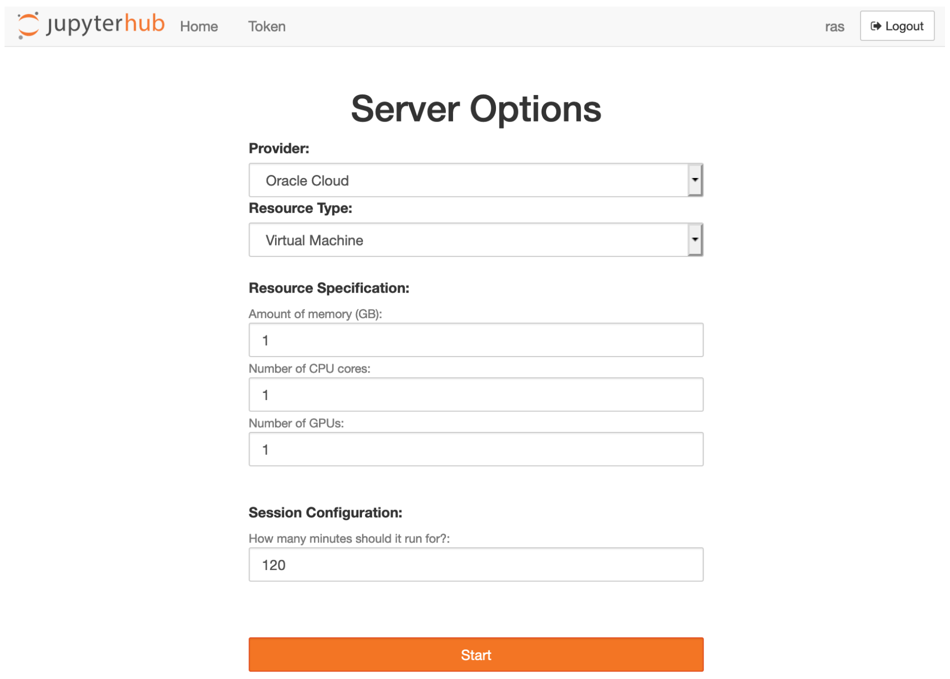


Figure 5: DAG MultipleSpawner Architecture, R = Resource

Building upon this, a simple benchmark was made to evaluate the gain in getting access to a compute resource with a NVIDIA P100 GPU. A Notebook with the Tensorflow and Keras quick start application [61] was used to get a rough estimate of how much time would be saved in building a simple neural network that classifies images. Listing 5, shows the results of running the notebook on the GPU powered compute resource for ten times in a row, and Listing 4 shows the results of running



Server Options

Provider: Oracle Cloud

Resource Type: Virtual Machine

Resource Specification:

Amount of memory (GB): 1

Number of CPU cores: 1

Number of GPUs: 1

Session Configuration:

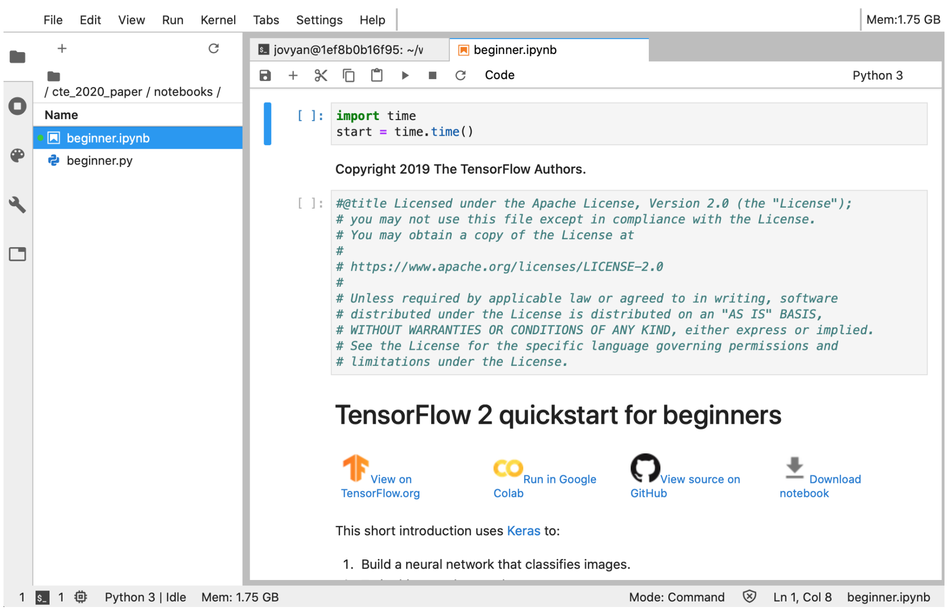
How many minutes should it run for?: 120

Figure 6: MultipleSpawner Interface

the same benchmark on an existing DAG resource. As this shows, the GPU version was on average 24,7 seconds faster or in other words gained on average a 2,8 speedup compared to the DAG resource without a GPU.

```
(python3) jovyan@d203812f76e8:~/work/cte_2020_paper/notebooks$ \
> python3 beginner.py
Took: 38.107945919036865
Took: 36.123350381851196
Took: 37.37455701828003
Took: 37.69051790237427
Took: 41.16242790222168
Took: 37.24052095413208
Took: 38.685391902923584
Took: 40.02782320976257
Took: 38.40936994552612
Took: 39.34704780578613
Average: 38.41689529418945
```

Listing 4: DAG compute resource Tensorflow times



```
[ ]: import time
     start = time.time()
```

Copyright 2019 The TensorFlow Authors.

```
[ ]: #@title Licensed under the Apache License, Version 2.0 (the "License");
     # you may not use this file except in compliance with the License.
     # You may obtain a copy of the License at
     #
     # https://www.apache.org/licenses/LICENSE-2.0
     #
     # Unless required by applicable law or agreed to in writing, software
     # distributed under the License is distributed on an "AS IS" BASIS,
     # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
     # See the License for the specific language governing permissions and
     # limitations under the License.
```

TensorFlow 2 quickstart for beginners

This short introduction uses Keras to:

1. Build a neural network that classifies images.

Figure 7: A Tensorflow + Keras Notebook on an OCI resource

```
(python3) jovyan@56e3c30c2af6:~/work/cte_2020_paper/notebooks$ \
> python3 beginner.py
Took: 19.479900360107422
Took: 12.859123706817627
Took: 13.047293186187744
Took: 13.296776056289673
Took: 13.002363204956055
Took: 13.118329048156738
Took: 13.067508935928345
Took: 13.089284658432007
Took: 13.160099506378174
Took: 13.032178401947021
Average: 13.715285706520081
```

Listing 5: OCI GPU compute resource Tensorflow times

From this simple benchmarking example, we can see that by utilizing the MultipleSpawner in combination with corc, users are able to get access through a simple gateway to the expected performance gains of accelerators like a GPU. Expanding on this, the teachers and students at UCPH will now be able to request a compute resource with a GPU on demand, thereby gaining simple access to achieving similar faster runtimes in their exercises and assignments.

6. Conclusions and Future Work

In this paper, we presented our work towards establishing a Grid of Clouds that enables organisations, such as educational institutions to share computational resources amongst themselves and external collaborators. To accomplish this, we introduced corc as a basic building block enables the ability to orchestrate, authenticate, configure, and schedule computation on a set of resources by a supported provider.

OCI was the first provider we chose to support in corc, foremost because of the existing collaboration with UCPH and the associated credits that got donated to this project. This enabled us to utilize said provider to cloud enable part of the DAG service at UCPH. This was made possible through the introduction of the MultipleSpawner package that utilized corc to dynamically chose between supported cloud providers. We demonstrated that the MultipleSpawner was capable of scheduling and stopping orchestrated and configured resources at OCI via a local researcher's machine.

In terms of future work, the next step involves the establishment of a Grid layer on top of the UCPH and OCI clouds. This Grid layer is planned to enable the establishment of a federated pool of participating organisations to share their resources. By doing so, we will be able to dynamically utilize cross organisation resources for services such as DAG, allowing us for instance to spawn Notebooks across multiple institutions such as other universities. Enabling the sharing of underused resources across the Grid participants. To accomplish this, corc also needs to be expanded to support additional providers, foremost through the integration of the Apache libcloud [62] library which natively supports more than 30 providers, we will allow corc and subsequently the MultipleSpawner to be utilized across a wide range of cloud providers.

Acknowledgments

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 765604. Furthermore, many thanks is given to Oracle for donating the cloud resources that made this project possible.

References

[1] A. Gupta, L. V. Kale, F. Gioachin, V. March, C. H. Suen, B. S. Lee, P. Faraboschi, R. Kaufmann, D. Milojicic, The who, what, why, and how of high performance computing in the cloud, Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom 1 (2013) 306–314. doi:10.1109/CloudCom.2013.47.

[2] B. Vinter, J. Bardino, M. Rehr, K. Birkelund, M. O. Larsen, Imaging Data Management System, in: Cloud NG:17 Proceedings of the 1st International Workshop on Next Generation of Cloud Architectures, Belgrade, Serbia, 2017.

[3] D. Häfner, R. L. Jacobsen, C. Eden, M. R. B. Kristensen, M. Jochum, R. Nuterman, B. Vinter, Veros v0.1 &amp;ndash; a Fast and Versatile Ocean Simulator in Pure Python, Geoscientific Model Development Discussions (2018) 1–22. doi:10.5194/gmd-2018-3.

[4] P. Padoan, L. Pan, M. Juvela, T. Haugbølle, Nordlund, The Origin of Massive Stars: The Inertial-inflow Model, The Astrophysical Journal 900 (2020) 82. doi:10.3847/1538-4357/abaa47.

[5] University of Copenhagen, University of Copenhagen policy for scientific data, Technical Report, Copenhagen, 2014. URL: https://kunet.ku.dk/arbejdsomraader/forskning/data/forskningsdata/Documents/Underskrevetogendeligversionafpolitikforopbevaringafforskningsdata.pdf.

[6] University of Copenhagen, SCIENCE AI Centre, 2020. URL: https://ai.ku.dk/research/.
[7] University of Antwerp, High Performance Computing CalcUA, 2020. URL: https://www.uantwerp.be/en/core-facilities/calcua/.
[8] Lund University, LUNARC, 2020. URL: https://www.maxiv.lu.se/users/lunarc/.
[9] Vienna Scientific Cluster, Vienna Scientific Cluster, 2009. URL: https://vsc.ac.at//access/.
[10] ARCHER, 2019. URL: https://www.epcc.ed.ac.uk/facilities/archer.
[11] European Union, European Commission, EuroHPC JOINT UNDERTAKING, Technical Report, 2020. URL: https://op.europa.eu/en/publication-detail/-/publication/dff20041-f247-11ea-991b-01aa75ed71a1/language-en. doi:10.2759/26995.
[12] I. Foster, C. Kesselman, The history of the grid, Advances in Parallel Computing 20 (2011) 3–30. doi:10.3233/978-1-60750-803-8-3.
[13] R. Munk, Cloud Orchestrator, 2020. URL: https://github.com/rasmunk/corc.
[14] Instructure, Canvas, 2020. URL: https://www.instructure.com/canvas/about.
[15] J. Bardino, M. Rehr, B. Vinter, R. Munk, ERDA, 2019. URL: https://www.erda.dk.
[16] R. Munk, Jupyter Service, 2019.
[17] G. Zaccone, R. Karim, A. Menshawy, Chapter 7: GPU Computing, in: Deep Learning with TensorFlow, 1 ed., Packt Publishing, Limited, 2017, p. 316.
[18] University of Copenhagen, Introduction to Computing for Physicists, 2019. URL: https://kurser.ku.dk/course/nfya06018u/.
[19] University of Copenhagen, Applied Statistics, 2019. URL: https://kurser.ku.dk/course/nfyk13011u.
[20] High Performance Parallel Computing, 2020. URL: https://kurser.ku.dk/course/nfyk18001u/.
[21] J. Berthold, J. Bardino, B. Vinter, A Principled Approach to Grid Middleware, in: Algorithms and Architectures for Parallel Processing, volume 7016, Springer, 2011, pp. 409–418. doi:https://doi.org/10.1007/978-3-642-24650-0{_}35.
[22] Project Jupyter, Project Jupyter, 2019. URL: https://jupyter.org/about.
[23] F. Perez, B. E. Granger, IPython: A System for Interactive Scientific Computing, Computing in Science and Engineering, Computing in Science and Engineering 9 (2007) 21–29. URL: http://scitation.aip.org/content/aip/journal/cise/9/3/10.1109/MCSE.2007.53. doi:10.1109/MCSE.2007.53.
[24] Project Jupyter, JupyterLab, 2018. URL: http://jupyterlab.readthedocs.io/en/stable/.
[25] Project Jupyter, JupyterHub, 2015. URL: https://pypi.org/project/jupyterhub/.
[26] Project Jupyter, JupyterHub Spawners, 2020. URL: https://github.com/jupyterhub/jupyterhub/wiki/Spawners.
[27] Project Jupyter, WrapSpawner, ???? URL: https://github.com/jupyterhub/wrapspawner.
[28] S. L. Proskura, S. H. Lytvynova, The approaches to Web-based education of computer science bachelors in higher education institutions, CEUR Workshop Proceedings 2643 (2020) 609–625.
[29] GitHub, GitHub, 2020. URL: https://www.github.com.
[30] Google, Google Docs, 2020. URL: https://docs.google.com.
[31] Google Colab, 2020. URL: https://colab.research.google.com.
[32] Kaggle Inc, Kaggle, 2018. URL: https://www.kaggle.com.
[33] Project Jupyter, Binder, 2017. URL: https://mybinder.org.
[34] Microsoft, Azure Notebooks, 2020. URL: https://notebooks.azure.com.
[35] CoCalc, CoCalc, 2020. URL: https://cocalc.com.
[36] JetBrains, Datalore, 2020. URL: https://datalore.jetbrains.com.
[37] BinderFAQ, 2017. URL: https://mybinder.readthedocs.io/en/latest/faq.html.
[38] Kaggle Inc, Kaggle Notebooks Documentation, 2020. URL: https://www.kaggle.com/docs/

notebooks.
[39] Google, Google Colab FAQ, 2020. URL: https://research.google.com/colaboratory/faq.html.
[40] Microsoft, Azure Notebooks Overview, 2020. URL: https://docs.microsoft.com/en-us/azure/notebooks/azure-notebooks-overview.
[41] Microsoft, Azure Notebooks manage and configure projects, 2020. URL: https://docs.microsoft.com/en-us/azure/notebooks/azure-notebooks-overview.
[42] CoCalc, Cocalc Docs, 2020. URL: https://doc.cocalc.com/index.html.
[43] JetBrains, Datalore Documentation, 2020. URL: https://datalore.jetbrains.com/documentation.
[44] R. Munk, DAG, 2019. URL: https://github.com/ucphhpc/jupyter_service.
[45] Kaggle Inc, Kaggle GPU Tips and Tricks, 2020. URL: https://www.kaggle.com/page/GPU-tips-and-tricks.
[46] RedHat, What is Orchestration, 2020. URL: https://www.redhat.com/en/topics/automation/what-is-orchestration.
[47] TerraForm, TerraForm, 2020. URL: https://www.terraform.io/docs/index.html.
[48] Puppet, Puppet, 2020. URL: https://puppet.com.
[49] Chef, Chef, 2020. URL: https://www.chef.io/products/chef-infra.
[50] Ansible, Ansible, 2020. URL: https://www.ansible.com.
[51] Oracle Corporation, Oracle Cloud Infrastructure CLI, 2020. URL: https://github.com/oracle/oci-cli.
[52] Amazon, AWS Command Line Interface, 2020. URL: https://aws.amazon.com/cli/.
[53] Amazon, Tools to build on AWS, 2020. URL: https://aws.amazon.com/tools/.
[54] OpenStack, OpenStack, 2020. URL: https://www.openstack.org.
[55] OpenNebula Systems, OpenNebula, 2020. URL: https://opennebula.io.
[56] INDIGO-DataCloud, INDIGO-DataCloud, 2020. URL: https://www.indigo-datacloud.eu.
[57] M. Caballer, S. Zala, López García, G. Moltó, P. O. Fernández, M. Velten, Orchestrating complex application architectures in heterogeneous clouds, Journal of Grid Computing 16 (2017) 3–18. doi:10.1007/s10723-017-9418-y.
[58] J. Kovács, P. Kacsuk, Occopus: A multi-cloud orchestrator to deploy and manage complex scientific infrastructures, Journal of Grid Computing 16 (2018) 19–37. doi:10.1007/s10723-017-9421-3.
[59] R. Munk, MultipleSpawner, 2020. URL: https://github.com/ucphhpc/multiplespawner.
[60] NERSC, SSHSpawner, 2016. URL: https://github.com/NERSC/sshspawner.
[61] NVIDIA, TensorFlow 2 Quickstart Notebook, 2020. URL: https://www.tensorflow.org/tutorials/quickstart/beginner.
[62] The Apache Software Foundation, libcloud, 2020. URL: https://libcloud.apache.org.

A.5 Further Developments in Event Oriented, Emergent Workflows*

Marchant, David; Munk, Rasmus; Vinter, Brian. Submitted to Euro-Par2021 (Euro-Par: 27th International European Conference on Parallel and Distributed Computing), Track 03: Scheduling and Load Balancing.

Appendix B

Non published papers

B.1 Teaching Parallel and Distributed Techniques at UCPH through JupyterLab

Munk, Rasmus; Bardino, Jonas

Teaching Parallel and Distributed Techniques at UCPH through JupyterLab

1st Rasmus Munk
Niels Bohr Institute
University of Copenhagen
Copenhagen, Denmark
rasmus.munk@nbi.ku.dk

2nd Jonas Bardino
Niels Bohr Institute
University of Copenhagen
Copenhagen, Denmark
bardino@nbi.ku.dk

***Abstract*—Teaching parallel computing traditionally comes with a significant amount of setup and maintenance work overhead for the students, often taking away focus and time from the core learning activities. At UCPH we created a small scale HPC sandbox service to remedy this situation and used it excessively during the 2019 High Performance Parallel Computing course. The service itself was designed to provide a user-friendly web environment to teach novice users about HPC systems and techniques without worrying a lot about the underlying software setup details. A tailor-made JupyterLab infrastructure was at the core of enabling this experience, while also allowing the use of a traditional command-line oriented HPC workflows via its shell. Containers were added to the mix to isolate the compute environment of users and control resources. For the storage part, a scalable and persistent user data storage was provided through integration of the universitys scientific data management platform, ERDA. As a result, the students and the scientific staff at the university have been empowered with a quick and easily accessible small scale HPC system to explore and exploit the potential performance benefits of e.g. multi-threaded or MPI based solutions.**

***Index Terms*—Teaching, Computing, HPC, Parallel and Distributed, Programming, Jupyter+JupyterLab, Docker, Shifter, SLURM**

I. Introduction

The effective and efficient usage of High Performance Computing (HPC) Systems is essential to conduct experimental research across various scientific fields. This includes research in energy, climate, physics, and life-sciences where the use of HPC Systems is increasingly common and necessary. The use case across such fields typically involves some kind of complex environment or phenomenon that, because of real world limitations including infeasibility, associated cost, or legal constraints is instead simulated on an HPC system. Examples include ocean modelling, weather and climate forecasts, nuclear energy research and astrophysics simulations, all of which suffer from one or more of these limitations. In the field of HPC Systems, there is a push towards developing bigger exascale machines, i.e. systems that can perform 10^{18} floating point operations per second. To achieve it, future HPC Systems will likely utilize heterogeneous compute capabilities including GPUs and similar accelerator devices to reach such scale. This development will impose additional programmer complexity in developing applications that can utilize such heterogeneous systems, which is in contrast to the homogeneous CPU based systems of the past. With that in mind, the challenge of effective and efficient usage of HPC systems is not going away but is only an increasing challenge to both veteran and novice users.

Typically when teaching both students and researchers alike in how to accomplish this at the Niels Bohr Institute (NBI), as part of the University of Copenhagen (UCPH), the task often has not been limited to simply instructing the students on how such a system can be efficiently used. Instead a substantial part of the teaching resources has been allocated to ensure that the students had access to an environment representative of a real HPC system, for example in terms of reflecting the workflow of using a batch queuing system to schedule jobs for computation. This environment would typically be either their own machines which had to be configured separately or a temporary shared system available only for the duration of the teaching period. It imposed a number of steps that were not directly related to the teaching of using HPC systems, but involved system administrative tasks such as installing development dependencies, which in itself is neither particularly relevant nor interesting for students not in the Computer Science field. Furthermore it often complicates and distracts from the lessons critical to learn as a user rather than an administrator.

To reduce the complexity of getting prepared for the course and to provide a user-friendly environment for exploring the world of HPC computation, the MPI [1] Oriented Development and Investigation (MODI) service has been introduced at UCPH. It aims to provide such an environment while still allowing for a classical interaction with a HPC system via for instance a Unix-like shell and a batch queuing system. The service is based on utilizing a range of technologies to wrap a classic SLURM [2] based cluster setup with Message Passing Interface (MPI) capabilities. It also includes numerous technologies that the Jupyter [3] project provides, which mainly focuses on providing a user-friendly interactive data analysis and scientific computing platforms. In our instance, we made use of both the JupyterHub multi-user gateway and the recently introduced JupyterLab interface to allow both

This project has received funding from the European Unions Horizon 2020 research and innovation programme under the Marie Skodowska-Curie grant agreement No 765604.

students and researchers to interact with the small scale HPC system via their local web browser. Beyond being intended as the default environment for teaching HPC technologies and techniques at UCPH, the service is also dedicated to act as a University resource to conduct suitable experiments. For instance, running small scale MPI powered physics simulations that aren't feasible in terms of cost nor scale to schedule on a fully fledged HPC environment like the ones operated by the Partnership for Advanced Computing in Europe (PRACE) [4]. In addition to providing a set of computational resources, the MODI service has also been integrated with the university's existing data management platform to allow automatic access to the individual user datasets which span from giga to tera bytes in size for subsequent processing. This paper introduces this new service, covering both the provided user experience, the technologies used, and the general architecture behind it to produce the complete environment. It includes a newly developed library (ldap_hooks), that extends JupyterHub with the novel functionality of allowing the creation of Directory Informatsion Tree (DIT) entities via LDAP during the spawning process, which allowed for the integration of externally generated profiles with a classic Unix-like user setup. This is followed by how the service was used in a limited roll-out for teaching the 2019 High Performance Parallel Computing (HPPC) course to a set of Physics and Computer Science Master's students at NBI. Furthermore, the paper also discusses how the course was received by the students in learning HPC techniques and how such system was subsequently exposed to the Universities researchers and collaboration partners as a general resource.

II. Related Work & Background

A web based platform for data processing is not a novel idea, at least since 1997 it has been accomplished with [5] and in various forms of systems and research as also reiterated by [6]. In addition, versions of utilizing JupyterHub and Jupyter Notebook based platforms to give access to HPC resources have also previously been discussed in various papers [7], [8].

Our approach is similar to the system presented by [7], in that our service also utilizes a JupyterHub gateway behind an optional front-end proxy server which handles authentication via the HTTP/HTTPS header authentication mechanism. However, we add to this functionality through the introduction of a new Authenticator module. It allows both a user defined authentication header and the ability to supply custom metadata to an authenticated user from an external system to effectively support deeper integration with external services.

In addition, like [6] we aim at providing a fully integrated platform that acts as a unified interface which is easy for users with non-HPC backgrounds to get initial hands-on experience with, such as scheduling tasks to a classic batch system. This is in contrast to [8], which focuses on providing a single functionality such as allowing the users to host a Jupyter Notebook via the existing batch queuing system. However, in our case it does not mean that we provide a custom interface to submit jobs or workflows on a back-end HPC cluster. Instead, we rely on providing the capability to integrate the JupyterHub gateway with existing web services, such as the UCPH's existing data management platform, Electronic Research Data Archive (ERDA) [9]. By integrating the new MODI service with ERDA it enables both easy access to computational resources for processing of existing and future datasets on the platform. It also enables the same data management capabilities of results generated on the MODI service, either through various supported languages via the JupyterLab's interactive interface for non-intensive add-hoc data analysis, or through the supported shell interface to submit classic batch jobs to a backend HPC system.

III. User workflow

From the user perspective, the result of this overall process was a fully web based platform, that enables users to access both their regular datasets and services that ERDA provides. Additionally it enables them to access the MODI service via the regular ERDA web interface as shown in the top right corner of Figure 3. Upon such a selection the user is then redirected to a JupyterHub gateway website from ERDA, which enables them to select a particular pre-built Jupyter Notebook from a drop-down menu and spawn it. When they do so they are redirected to their own notebook instance, which will expose the default JupyterLab interface. Figure 1 shows an example of this where an instance of the 'HPC Notebook' has been spawned. As seen, the user is provided with a set of default home directories, i.e. *erda_mount* which contains the user's ERDA content, *modi_images* that contains the set of pre-built image environments available to the user for job submissions, and *modi_mount* which is the mount point of the user's staging scratch space. The scratch space is used to share files such as job scripts with each of the HPC nodes and to temporarily store the generated results from the jobs before moving them to a permanent destination i.e. the ERDA home. Depending on the particular image the user selects from the initial drop-down menu, the JupyterLab session will provide support for various programming languages. As indicated by Figure 1, the 'HPC Notebook' for instance supports multiple types of programming languages for interactive computation, i.e. Python3 and C++11,14,17. These can be utilized either through the interactive Console to typically execute one-off statements, or through the Notebook to define a document of multiple code blocks that can be executed independently. In terms of scheduling tasks to the backend HPC system, the user is expected to utilize the JupyterLab's Terminal shell to interact with the cluster via regular SLURM commands.

IV. Technologies

To provide the highlighted workflow experience, several different technologies and components were used and integrated. This section will describe how they were all used to develop the service.

A. ERDA

ERDA provides a web based data management platform across UCPH with a primary focus on the Faculty of Science.

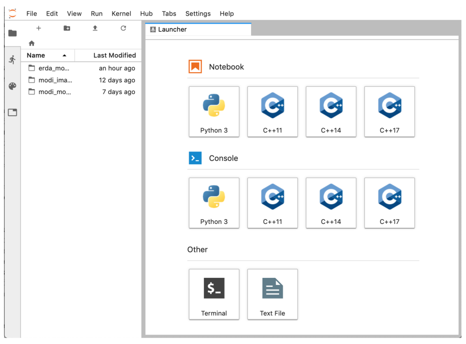


Fig. 1: MODI JupyterLab Interface

Its primary role is to be a data repository for students, researchers and general employees across UCPH. It enables this through a simple web UI powered by a combination of an Apache webserver and a Python based backend that enables the users to either interact with the different services through its navigation menu, or the user's individual files and folders via its file manager. An example of the interface can be seen in Figure 3. The platform itself is a UCPH-specific version of the open source Minimum intrusion Grid (MiG) [10] project, that provides multiple data management functionalities, such as easy and secure upload of datasets, simple access mechanisms through a web file manager, and the ability to enable internal collaboration and data sharing between users through Workgroups/VGrids. The platform also provides users with efficient IO access to their data through standard secure protocols like WebDAV over SSL/TLS (WebDAVS), FTPS and SFTP. WebDAVS additionally allows users to natively mount or map their ERDA home as a network drive and on all commonly used platforms. On Linux/Unix the SSHFS software can similarly even do so through the more efficient SFTP protocol. All these capabilities have turned it into an intrinsic data repository for the University's researchers, students, and partners. With a current userbase of $\sim$2000 users and a usage of 2.3 PB out of 3.2 PB available. An upgrade to the system is currently being developed/implemented which up-scales the available capacity to 8.7 PB.

B. *Jupyter*

Project Jupyter [3] is the overall open source project that develops multiple tools aiming to support interactive data science and scientific computing. The foundation of these is the IPython Notebook (.ipynb) format (evolved out of the IPython Project [11]). It is based on interpreting special segments of a JSON document as source code, which can be executed by a custom programming language runtime environment (also known as a *kernel*). The Jupyter Notebook was the subsequently developed web based interface to both allow the creation and execution of such documents through a browser, while also providing numerous features such as access to an

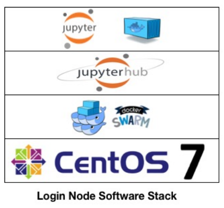


Fig. 2: Login Node Software Stack

IPython shell, a classic shell or a regular text document to name a few. In time this interface will be superseded by the JupyterLab [12] interface (as shown in Figure 1). Both of these web interfaces are based on providing a single web-based user experience. JupyterHub [13] is the de-facto standard to enable multiple users to utilize the same compute resources for individual Jupyter Notebook/Lab sessions. It does this through its own web interface gateway and backend database to segment and register individual users before allowing them to start/spawn a Jupyter session. In addition, it allows for the extension of both custom Spawners and Authenticators to allow for site specific implementations on how the users should be allowed to spawn an instance and how either the Jupyter Notebook or JupyterLab instances should be spawned.

C. *SLURM*

SLURM or historically Simple Linux Utility for Resource Management [14] [2] [15] is a highly scalable open source job scheduling and cluster workload manager frequently used in both small and large HPC Systems. It is typically used to allocate parallel tasks (compute jobs) to resources (compute nodes), with subsequent job lifetime management from start to finish of the individual tasks. It does this through a selected job scheduling algorithm to pick individual tasks from a dedicated queue of user submitted jobs. As a minimum configuration, SLURM requires a cluster control daemon and a compute node daemon. In MODI the **login node** hosts the dedicated control daemon managing the task queue, while the backend **SLURM Nodes** host the compute daemon which executes the user jobs. The JupyterLab instance Terminal enables the users to do typical SLURM tasks such as inspecting nodes, the queue, or adding job tasks to name a few.

D. *Architecture*

The underlying architecture supporting all of the different components can be seen in Figure 3. As highlighted, the MODI service itself is provided by two sets of systems, a non-computation intensive part of four interconnected virtual nodes, and eight bare-metal SLURM nodes dedicated to the computational tasks that the users submit to them.

When a user requests access to the MODI service on ERDA, the user is redirected to the node responsible for providing it. As shown in Figure 3 the **login node** has this responsibility. To provide this, the **login node** hosts both the JupyterHub

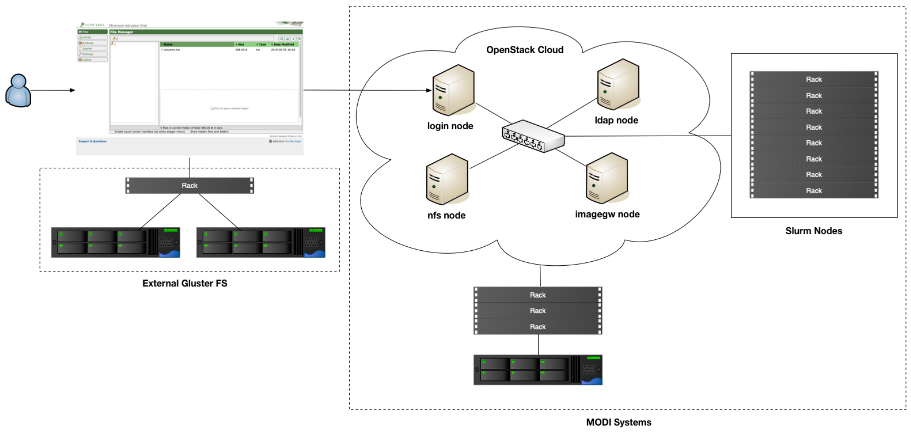


Fig. 3: MODI Architecture

gateway and the user's individual JupyterLab instances. It does this through the software-stack as shown in Figure 2, to enable the scheduling of both the JupyterHub gateway and the user instances as individual Docker Services [16] via the Docker Swarm Orchestration engine. This will result in the services being scheduled in isolated containers. In that way we are able to limit the amount of resources the individual services can claim on the host through cgroups [17] restrictions. It also enables us to isolate the individual user's perspective of the operating system environment through namespaces, including both their perspective of the file system and their process tree. In turn this also provides us with the capability to scale the service to a multi-node service if so needed in the future.

The running JupyterHub service itself is a custom Docker image build [18], that includes the JupyterHub SwarmSpawner [19], which enables the scheduling of JupyterLab sessions as Docker Swarm services. Furthermore, the JupyterHub build also comes with the new HeaderAuthenticator as provided by [20] and inspired by [21]. It enables the sought after integration with ERDA through its API by allowing additional state information to be provided to a particular user.

E. Authentication and Security

The primary responsibility of ensuring that only authenticated users are granted access remains with the ERDA platform. It ensures this through one of its supported authentication mechanisms, which currently includes OpenID and X.509 certificates.

On MODI, the introduced HeaderAuthenticator configures the JupyterHub service such that users are able to authenticate themselves by providing a required 'auth' header for authenticating. By default the authenticator is set to expect the 'Remote-User' header, similar to what [7] uses. However in [20] this preset header is changeable, i.e it can be customized as per the specific architecture demands. In our instance, the default 'Remote-User' header fitted the purpose adequately. Furthermore, the *allowed_headers* option, allows for the administrator to specify additional header values that the user, during the authentication phase is able to provide to the subsequent Spawner. Additionally the underlying Linux/Unix environment can also be provided with this information for the lifetime of the authenticated session or as a permanent state. Details on how either of these can be achieved and configured can be found at [20].

The HeaderAuthenticator enables ERDA to share the necessary information to integrate the ability for users to access their external ERDA data, but also to supply the MODI setup with the user details necessary to instantiate a unique and valid MODI Linux profile. To access the external data, ERDA was extended with the capability to generate and supply limited lifetime SSH key sets for an individual user, which could subsequently be re-used by MODI to mount a users ERDA home via SSHFS. In the Linux profile case, it enabled us to share the ERDA user profile information, which, because of its original authentication method is based on the x.509 certificate format and their associated Distinguished Name strings. Therefore, every user no matter the choice of authentication is represented by such a data-structure on ERDA. We chose to provide this identifier to the JupyterHub user state via the HeaderAuthenticator, which could subsequently be used for creation/lookup of that user's Linux profile information during the spawning phase.

It is important to note, this type of header authentication should only be allowed when the JupyterHub service is deployed behind a proxy (in our case ERDA). It takes care of proper authentication and ensures that the user can't forge the header before it is forwarded to the service. The reason being that on its own, the HeaderAuthenticator doesn't attempt to validate whether the user isn't impersonating another, by for instance requiring secret information which only the real user should known such as a password, or other forms of external verification such as directly with the OpenID authentication provider. Also of note in regards to this is that the web requests in general should only be allowed to come from the proxy itself, i.e. the system should limit who is allowed to make HTTP/HTTPS request against the service. In our instance we imposed iptables firewall rules to enforce this, i.e. by only exposing the JupyterHub webserver to the ERDA frontend proxy. If similar safeguards can't be enforced, another Authenticator should be used, a list of which can be found at [22].

F. Providing Linux Profiles

In the Linux profile case, it was necessary to have them uniformly distributed across multiple services, in particular across the user's Lab session, and the SLURM compute nodes for job execution, and on the shared NFS storage to ensure proper file access between the services. Here, we used the typical LDAP based authentication through a combination of the Name Service Switch (NSS) module and Pluggable Authentication Module (PAM) approach [23] against a centralized Directory server. This provided us with an internal DIT service that could both manage external and internal LDAP requests. In MODI, the **ldap node** as highlighted in Figure 3 was dedicated to provide this functionality by being configured with the OpenLDAP directory server.

The task of storing the supplied user information on the **ldap node's** DIT, was accomplished by utilizing the JupyterHub 'pre_spawn_hook' [24] mechanism. It allows for a pre-stage function to be executed before the user's JupyterLab session is created. The ldap_hooks [25] leverages this to interact with an LDAP service during the spawning phase to either lookup or create a user entry inside the LDAP DIT.

Ldap_hooks accomplishes it by utilizing the ldap3 [26] library to either, add, update, or delete entries in an external DIT by communicating over LDAP to a specified host. Furthermore, the library enables the JupyterHub configuration to search the existing DIT for an entry with a particular *objectclass* and *attributes* combination before an entry is created. The search operation can also be used to perform subsequent operations on the retrieved results, such as generating unique UIDs for new users.

```
objectclass ( ObjectIdentifier
    NAME 'x-nextUserIdentifier'
    DESC 'An object containing an uid
        attribute, can be used with an atomic
        delete+add operation to generate a new
        uid'
    SUP top STRUCTURAL
    MUST ( cn $ uidNumber )
)
```

Listing 1: x-nextUserIdentifier

In MODI, this was accomplished by introducing a custom objectclass, i.e. x-nextUserIdentifer that with its attribute 'uidNumber', keeps track of the most recent allocated UID to a user. The DIT schema definition for this objectclass can be seen in Listing 1. This enabled the creation of the 'uidNext' entry instance. Which allowed for the generation and allocation of unique UID's by utilizing the built-in atomic LDAP operation 'modify-delete-add' [27]. To support ERDA's specific selection of x509 certificate Distinguished Name attributes to define a user, an additional structural objectclass was created to support every field that the user could potentially provide, with a minimum requirement of providing a common name attribute. The schema definition for this particular class can be seen in Listing 2. The MUST and MAY fields should be translated as follows; cn='common name', c=country, s=state, l=language, o=organisation, ou=organisational-unit.

```
objectclass ( ObjectIdentifier
    NAME 'x-certsDistinguishedName' SUP top
    STRUCTURAL
    DESC 'An object containing the attributes
        for a common x509 Distinguished Name
        '
    MUST cn
    MAY ( c $ st $ l $ o $ ou $ emailAddress )
)
```

Listing 2: x-certsDistinguishedName

In addition, this objectclass type was combined with the predefined schema class 'PosixAccount' [28] to enable that the subsequent user entry could be assigned the generated UID, a potential Group Identifying Digit (GID), and a default 'homeDirectory' value.

G. Staging Scratch Space

To allow the authenticated users and their associated JupyterLab services to share resources with the SLURM nodes, the **nfs node** provides a per user scratch space. This is utilized by a user's scheduled container to stage files they either produced or collected from their ERDA home before a particular job is scheduled to the SLURM nodes. The mounting of the external NFS share was provided by dedicating the **login node** to mount the share containing the user's home directories. Subsequently the individual user's home directory is mounted into the scheduled JupyterLab service when spawned. It ensures that we would not introduce the substantial overhead of establishing an individual mount session/connection for each spawned JupyterLab user session.

H. Shifter

Finally there is a need to ensure that the users environment on MODI provides consistency and compatibility between the place where the users develop their HPC applications and the actual execution environment on the SLURM nodes. A solution to this, was to allow for pre-built Docker Notebook image environments as part of the job execution on the SLURM nodes.

This was accomplished by introducing Shifter [29] on the SLURM nodes on MODI. It ensured that the users, as part of their SLURM jobs, would be enabled to specify which Shifter pre-converted image the job tasks should be executed within.

This meant the users were able to compile their source code inside the JupyterLab service session, stage it to the NFS shared directory and subsequently submit the job to the SLURM cluster without issues of potential mismatches. The Shifter container runtime environment, requires that the images are in a supported format such as ext4 or squashfs [30] before they can be successfully executed. To maintain and accomplish this, the setup encourages the use of an Image Gateway [31], either on a dedicated or shared system. The Image Gateway will then pre-stage the container images, either by pulling and converting them directly from the official DockerHub repository automatically during execution or manually pre-pulled by a user. It does introduce the additional complexity of having a required daemon running to manage this, which has to be maintained in terms of keeping the service running and up to date. Once established, the individual SLURM nodes can accept jobs that utilize the user's applications and a prepared image to execute the specific task within.

V. Additional compute services

Beyond the introduced MODI service, a prior developed service named Data Analysis Gateway (DAG) was also available to the ERDA users, which included the HPPC students during the course. DAG consists of an eight node setup all part of a combined Docker Swarm cluster configuration. In this service, the user can similarly spawn containerized JupyterLab sessions with intermediate session lifetime for computation. However, on DAG the user is limited to the resource granted by that particular session, which as of May 2019 was eight GB of memory and 100% utilization of eight logical cores on a single machine. In contrast to MODI, these sessions have no access to external compute resources such as a SLURM cluster and are limited to a single node environment. This setup has been supporting other courses and teaching activities at UCPH, including Introduction to Computing for Physicists [32] in 2018, and Applied Statistics [33] in 2019 and several intermediate workshops.

This service was also highlighted to the students as a potential resource for prototyping, testing and benchmarking their applications that do not utilize parallel and distributed techniques, such as MPI, MPI-IO, or excessive Threading beyond the limited eight logical cores.

VI. Courses and Curriculum's

The HPPC course intends to teach the postgraduate/masters students in several HPC topics. This includes areas such as Basic Computer Architecture, Vectorization, Shared Memory Architecture/Programming, Distributed Memory and Networked Architecture and I/O related topics such as disk, tape, parallel I/O, staging and file systems. During the 2019 edition, the course had 26 students enrolled from Computer Science, Physics and the Biology departments.

The course itself was taught over an eight week period, where the students had two weekly taught lectures. One of these was a classical two hour presentation which introduced that week's topic, while the second was a two hour hands-on practical session. Over the course of these eight weeks the students were given a total of four assignments, each one including a C++ application that the students were tasked with optimizing. The applications included an N-body NICE simulation, a Climate Model, a Shallow Water Simulation and a CT Imaging Reconstruction assignment [34].

For each assignment, the students were expected to utilize the taught techniques to improve the performance of that particular implementation. For instance, in the N-body assignment, the task was to speedup a basic sequential implementation by for instance adapting it to utilize Vectorization to update the velocity and position of each object in the solar system during each timestep [34]. The results of the achieved benefits and how they were accomplished was to be described and documented in a subsequent assignment report. The grading would then be based on the average of the 3 best passing reports produced by the student.

To complete these assignments, the students also had to get familiar with associated subjects such as writing basic C++ code, compilation, debugging (typical print statements or GDB) in addition to using techniques such Single Program Multiple Data (SPMD) through MPI or Threading through either explicit POSIX threads (pthreads) or implicit via OpenMP.

Since the MODI system was still being developed while the course was running, the initial 2 assignments were not able to utilize the backend SLURM cluster for performance benchmarks. Instead the students were instructed to utilize the existing DAG service with the mentioned limitations. This did not pose a big issue, since the 2 initial assignments were focused on using vectorization and shared memory programming i.e. Threading/OpenMP to optimize the implementation, which didn't require the availability of SLURM cluster to execute or perform adequate benchmarks.

However, for the final two assignments, i.e. the Shallow Water Simulation and CT Imaging Reconstruction the MODI service was available to all course students for both testing and benchmarking their solutions on the backend SLURM cluster. To ease the transition of using the Shifter images to execute the student solutions, Makefiles were provided as part of the assignments. These could produce a SLURM ready job script that would specify the developed HPC image as the default environment in which the individual nodes should execute the compiled binary.

During the course, the students could raise questions and issues either directly during the teaching sessions or through the UCPH course platform, Absalon which enables both the distribution of course material and the ability to raise Discussions or Announcements on. It was used to resolve issues throughout the course including MODI and DAG related problems. This included trouble such as incorrect execute permission on compiled binaries that were submitted to the SLURM queue, reporting of OpenID authentication problems, and service down times and general compile and programming errors.

Once the course was completed, the students were given

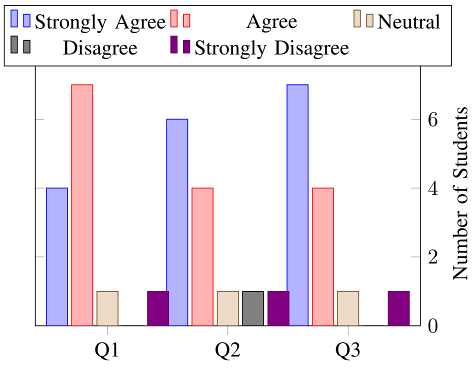


Fig. 4: Student Feedback Questions [35]
Q1=*"I believe that I have acquired the competencies described in the course objectives"*
Q2=*"In my opinion, the teaching material was relevant to the course"*
Q3=*"Overall, I find that the course has been useful"*

the opportunity to give feedback on the teaching experience. A summary of some of this feedback can be seen in Figure 4. What can be drawn from it is that the responding students had a general good experience during the course, with both an adequate workload for ten out of 13 respondents and general agreement from 11 out of 13 students that the course was useful. Another good impression is that ten out of 13 found the teaching material relevant and that 11 out of 13 felt that they had learned the course objectives.

Furthermore, ten out of 13 responding students thought that the course workload was adequate, while two thought it was too high, and one that it was too low.

Beyond this, the students were given the opportunity to give comments on the course. These comments were posed as responses to the two phrases, "What was good about the course? Why?" and "I would like to suggest the following improvements". The good aspects included that the students in general liked the weekly assignments, the notebook handouts which explained the tasks at hand, and the useful topics. In relation to the introduced DAG and MODI services, the students highlighted that additional information/documentation about these systems, such as general explanations of their terminologies would be helpful. Also additional minor practical exercises would improve the understanding of the presented material, especially for those without a Computer Science background. Furthermore, that the two final assignments could benefit for more time to accomplish the tasks at hand, since they impose a steeper learning curve compared to the two initial assignments.

VII. Conclusions

The ability to both utilize current and future HPC systems capabilities, from existing tera- and peta scale systems to developing exascale systems is important across many research and industry fields. Through the effective and efficient use of these systems existing and developing research can be explored at increasing scale and detail. At UCPH we have developed the MODI service which consist of an eight node small scale HPC-like SLURM cluster. The service is designed to act both as a sandbox where UPCH students and employees can gain experience with such a system, but also as a general service where HPC-like jobs can be developed and benchmarked before being scheduled to an expensive dedicated HPC resource such as PRACE.

The service itself allows users to conduct both web-based interactive and classical terminal computing via individual containerized JupyterLab sessions. In addition, the development of this service introduced the ldap_hooks library, which makes it possible for JupyterHub spawners to automatically create and load entries in a DIT via LDAP. This can be utilized to provide an automatic conversion from external user profiles to appropriate Linux profiles that via nss-pam-ldap can be used to validate their subsequent access to shared resources such as submitting jobs on a SLURM cluster. In addition, by utilizing Shifter to provide the pre-built container image environments as part of the job submissions, the service ensures that environment-specific expectations such as shared library dependencies, version, and file system locations are consistent between the systems involved.

During this introduction, the service was used in teaching the HPPC course at UCPH for a set of 26 enrolled students from Computer Science, Physics and Biology. The students utilized the system to develop, test, and benchmark various programs as part of their course assignments. Specifically the students were taught in HPC related subjects such as Vectorization, Shared Memory Architecture/Programming, Distributed Memory and Networked Architecture, and Distributed I/O.

To accomplish this, two Jupyter powered services, namely MODI and DAG were introduced in the beginning of the course as resources to complete these assignments. Here the students were instructed to complete the assignments relating to the single node techniques (Vectoraction and Shared Memory Programming) on the existing DAG service while the multiple node assignment and techniques (both Distribute Memory and I/O) were designated to utilize the newly introduced MODI service. Across all assignments the students were instructed to optimize the performance of the provided implementations with the taught techniques such as Threading, OpenMP and MPI.

During the 2019 instance, the course was completed by 24 students. As highlighted in Figure 4, these students were given the opportunity to submit feedback to the overall experience. The result of which was that the responding students had a general favourable experience during the course. For instance, the majority agreed to having acquiring the competencies of the course objectives, they found the course useful, and the teaching material relevant. The students also gave commentary feedback in regards to both the good aspects of the course and potential improvements. Specifically, the assignments, notebook handouts and how the topics were relevant to the HPPC field were highlighted as good aspects. In relation to the Jupyter services, the students requested additional documentation about both the DAG and MODI services and that minor practical exercises to complement the hand-in assignments would be of benefit. Overall the course was well rated across the responding students, both in terms of the workload required, the level at which the course was taught, and that the vast majority strongly agreed with the course being useful.

Subsequent to course completion, MODI was made available to every ERDA user on UCPH to utilize as a sandbox for intermediate HPC tasks. For this rollout, the SLURM cluster setup was augmented to a 3 partition queue split (*devel*, *short*, *long*), with a decreasing job priority and increasing allowed job time (*20 mins*, *48 hours*, *one week*) from *devel* to *long*. This was to ensure that future HPPC course instances and short term development and benchmarking tasks were prioritized above hour and multiple day tasks. In addition, as per student feedback about lacking documentation about the MODI service, a user manual was created as a getting started guide to both future instances of the course and the ERDA users in general. Furthermore, the introduced Shifter service was replaced by a Singularity [36] powered setup to provide the same benefits of containerized SLURM jobs while removing the dependency of a dedicated Image Gateway daemon. The setup was simplified by this change by removing the additional **imagegw node** while providing the same functionality.

VIII. Future Work

In the future we will focus on supporting the MODI service as a permanent service at the UCPH. In this regard, we would like to introduce an image management service to handle the preparation of the Singularity images upon changes, updates or upgrades, which would reduce the general manual management required to keep the service up to date. In terms of the ldap_hooks library, the current usage relies on simple username+password authentication from the **login node**, in the future this should be changed to utilize the SSL/TLS-based authentication via temporary LetsEncrypt certificates to improve the security of the system. Finally in relation to the next instances of the HPPC course, we are considering swapping the order of assignment three and four as a response to the student feedback about the difficulty jump from assignment two to three.

References

[1] The Open MPI Project, "OpenMPI," 2004. [Online]. Available: https://www.open-mpi.org

[2] F. Moll, "Slurm Overview," 2018.

[3] Project Jupyter, "Project Jupyter," 2019. [Online]. Available: https://jupyter.org/about

[4] PRACE, "PRACE," Ph.D. dissertation, 2019.

[5] G. Aloisio, M. Cafaro, R. Williams, and P. Messina, "A distributed web-based metacomputing environment," 1997, pp. 480–486.

[6] B. Glick and J. Mache, "Jupyter Notebooks and User-Friendly HPC Access."

[7] M. Milligan, "Interactive HPC Gateways with Jupyter and Jupyterhub," *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact - PEARC17*, pp. 1–4, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3093338.3104159

[8] A. Prout, W. Arcand, D. Bestor, B. Bergeron, C. Byun, V. Gadepally, M. Hubbell, M. Houle, M. Jones, P. Michaleas, L. Milechin, J. Mullen, A. Rosa, S. Samsi, A. Reuther, and J. Kepner, "MIT SuperCloud portal workspace: Enabling HPC web application deployment," *2017 IEEE High Performance Extreme Computing Conference, HPEC 2017*, pp. 1–6, 2017.

[9] J. Bardino, M. Rehr, and B. Vinter, "ERDA," 2019. [Online]. Available: www.erda.dk

[10] J. Berthold, J. Bardino, and B. Vinter, "A Principled Approach to Grid Middleware," in *Algorithms and Architectures for Parallel Processing*, vol. 7016. Springer, 2011, pp. 409–418.

[11] F. Perez and B. E. Granger, "IPython: A System for Interactive Scientific Computing, Computing in Science and Engineering," *Computing in Science and Engineering*, vol. 9, no. 3, pp. 21–29, 2007. [Online]. Available: http://scitation.aip.org/content/aip/journal/cise/9/3/10.1109/MCSE.2007.53

[12] Project Jupyter, "JupyterLab," 2018. [Online]. Available: http://jupyterlab.readthedocs.io/en/stable/

[13] ——, "JupyterHub," 2015. [Online]. Available: https://pypi.org/project/jupyterhub/

[14] M. Jette and M. Grondona, "SLURM: Simple Linux Utility for Resources Management," 2003.

[15] SchedMD LLC., "SchedMD — Slurm Support and Development," 2017. [Online]. Available: https://www.schedmd.com/

[16] Docker Inc., "Docker Swarm overview," 2019. [Online]. Available: https://docs.docker.com/engine/swarm/

[17] Wikipedia, "cgroups," 2013. [Online]. Available: https://en.wikipedia.org/wiki/Cgroups

[18] R. Munk, "NBI JupyterHub," 2018. [Online]. Available: https://cloud.docker.com/u/nielsbohr/repository/docker/nielsbohr/jupyterhub

[19] ——, "jhub-swarmspawner," 2018. [Online]. Available: https://github.com/rasmunk/SwarmSpawner https://pypi.org/project/jhub-swarmspawner/

[20] ——, "jhub-authenticators," 2018. [Online]. Available: https://github.com/rasmunk/jhub-authenticators https://pypi.org/project/jhub-authenticators/

[21] Cwaldbieser, "jhub_remote_user_authenticator," 2015. [Online]. Available: https://pypi.org/project/jhub-remote-user-authenticator/

[22] Jupyter Development Team, "JupyterHub Authenticators," 2014. [Online]. Available: https://github.com/jupyterhub/jupyterhub/wiki/Authenticators

[23] A. d. Jong, "nss-pam-ldapd," 2009. [Online]. Available: https://arthurdejong.org/nss-pam-ldapd/

[24] Jupyter Development Team, "pre_spawn_hook," 2019. [Online]. Available: https://jupyterhub.readthedocs.io/en/stable/api/spawner.html

[25] R. Munk, "ldap_hooks," 2019. [Online]. Available: https://pypi.org/project/ldap-hooks/

[26] G. Cannata, "ldap3," 0. [Online]. Available: https://pypi.org/project/ldap3/

[27] JSPWiki, "LDIF Atomic," 2013. [Online]. Available: https://ldapwiki.com/wiki/LDIF Atomic Operations

[28] SERVICES.WILLEKE.BIZ, "PosixAccount," Ph.D. dissertation. [Online]. Available: https://ldapwiki.com/wiki/PosixAccount

[29] NERSC, "Shifter," 2015. [Online]. Available: https://github.com/NERSC/shifter

[30] M. Fasel, J. Porter, D. Jacobsen, L. Gerhardt, M. Mustafa, W. Bhimji, S. Canon, and V. Tsulaia, "Shifter: Containers for HPC," *Journal of Physics: Conference Series*, vol. 898, p. 082021, 2017.

[31] R. S. Canon and D. Jacobsen, "Shifter : Containers for HPC," 2016.

[32] University of Copenhagen, "Introduction to Computing for Physicists," 2019. [Online]. Available: https://kurser.ku.dk/course/nfya06018u/

[33] ——, "Applied Statistics," 2019. [Online]. Available: https://kurser.ku.dk/course/nfyk13011u

[34] M. R. B. Kristensen, "madsbk/hpc_course (github)."

[35] University of Copenhagen, "Results for High Performance Parallel Computing B3-3F19 - Block 3, 2018/2019," University of Copenhagen, Tech. Rep., 2019.

[36] Sylabs.io, "Singularity," 2019. [Online]. Available: https://www.sylabs.io/singularity/

Appendix C

Compute Systems Specifications

Table C.1: System specifications for benchmark environments

System	Num Nodes	CPU	Cores	Ghz	Memory (GB)	Internal Network
Internal OpenStack VM	1	Intel Xeon E3-12xx	12	2.6	25	1 Gbps
DAG	8	AMD EPYC 7501	8	2	256	25 Gbps
MODI	8	2x AMD EPYC 7501	64	2	256	25 Gbps RoCE (1 μs)

Appendix D

Benchmark Environments

Table D.1: System specifications for benchmark environments

System	CPU	Cores	Ghz	Memory (GB)
PC	AMD Ryzen 7 2700X	8	2.2	16
Data Analysis Gateway	AMD EPYC 7501	8	2	8
MPI Oriented Development and Investigation	2x AMD EPYC 7501	64	2	256
Oracle Standard VM.Standard2.2 [125]	Intel Xeon Platinum 8167M	2	2	16

Table D.2: McStas Benchmark Specifications

System	Num Nodes	CPU	Cores	Ghz	Memory (GB)
Laptop	1	i7-8550U	4	1.8	8
OCI Cluster 1	1	EPYC 7551	4	2.0	60
OCI Cluster 2	20	EPYC 7551	24	2.0	320

Appendix E

Benchmark Examples

E.1 Mig Utils

```
#!/usr/bin/env python
# coding: utf-8
import argparse
import os

parser = argparse.ArgumentParser()
parser.add_argument("sharelink",
    default="", type=str)
parser.add_argument("input_filename",
    default="", type=str)
parser.add_argument("--data_dir",
    default="foam_ct_data", type=str)
parser.add_argument("--bench_type",
    default="default", type=str)
parser.add_argument("--output_accepted_dir",
    default="foam_ct_data_discarded", type=str)
parser.add_argument("--output_discarded_dir",
    default="foam_ct_data_accepted", type=str)
parser.add_argument("--timing_dir",
    default="timings", type=str)
parser.add_argument("--index",
    default="0", type=str)
args = parser.parse_args()

share_link = args.sharelink
bench_type = args.bench_type
output_filedir_accepted = args.output_accepted_dir
output_filedir_discarded = args.output_discarded_dir
timing_dir = args.timing_dir
index = args.index
input_filename = os.path.join(args.data_dir,
"{}.npy".format(args.input_filename))

# input_filename = 'foam_ct_data/foam_000_ideal_CT.npy'
# input_filename = 'foam_ct_data/foam_027_big_CT.npy'
# input_filename = "foam_ct_data/
# foam_0{}_ideal_CT_I0_1000.npy".format(iteration_number)
# input_filename = 'foam_ct_data/foam_045_few_CT.npy'
# output_filedir_accepted = "foam_ct_data_accepted"
# output_filedir_discarded = "foam_ct_data_discarded"
print("Loading: {}".format(input_filename))
porosity_lower_threshold = 0.8
utils_path = "idmc_utils_module.py"

import numpy as np
import importlib
import matplotlib.pyplot as plt
import time
import importlib.util
from mig.io import IDMCShare

file_name = os.path.basename(__file__).split(".")[0]
bench_file = os.path.join(timing_dir, "{}_{}_bench.csv".format(bench_type,
file_name))
start = time.time()

datastorage = IDMCShare(share_link)

spec = importlib.util.spec_from_file_location("utils", utils_path)
utils = importlib.util.module_from_spec(spec)
spec.loader.exec_module(utils)

# Parameters
n_samples = 10000

# Load data
ct_data = None
with datastorage.open(input_filename, "rb") as _file:
    ct_data = np.load(_file)

assert ct_data is not None
utils.plot_center_slices(ct_data)

sample_inds = np.random.randint(0, len(ct_data.ravel()), n_samples)
n_components = 2
# Perform GMM fitting on samples from dataset
means, stds, weights = utils.perform_GMM_np(
    ct_data.ravel()[sample_inds],
    n_components,
    plot=True,
    title="GMM fitted to "
    + str(n_samples)
    + " of "
    + str(len(ct_data.ravel()))
    + " datapoints",
)
print("weights: ", weights)

# Classify data as 'accepted' or 'dircarded' according to porosity level
# Text file named according to the
# dataset will be stored in appropriate directories
filename_withouth_npy = input_filename.split("/")[-1].split(".")[0]

output_file = None
if np.max(weights) > porosity_lower_threshold:
    if not datastorage.exists(output_filedir_accepted):
        datastorage.mkdir(output_filedir_accepted)
    output_file = os.path.join(
        output_filedir_accepted, "{}_{}.txt".format(bench_type,
        filename_withouth_npy)
    )
else:
    if not datastorage.exists(output_filedir_discarded):
        datastorage.mkdir(output_filedir_discarded)
    output_file = os.path.join(
        output_filedir_discarded, "{}_{}.txt".format(bench_type, filename_withouth_
    )


with datastorage.open(output_file, "wb") as _file:
    data = str(np.max(weights)) + " " + str(np.min(weights))
    _file.write(data)

stop = time.time()

if not datastorage.exists(timing_dir):
    datastorage.mkdir(timing_dir)

if not datastorage.exists(bench_file):
    with datastorage.open(bench_file, 'w') as _file:
        _file.write("index, start, stop, time, size\n")

# Save timing data
with datastorage.open(bench_file, "a") as _file:
    content = "{},{},{},{},{}\n".format(
        index, start, stop, stop - start, ct_data.nbytes
    )
    _file.write(content)
```

Listing 13: initial_porosity_check.py.

```
#!/usr/bin/env python
# coding: utf-8
import argparse
import os
```

```
# foam_ct_data_accepted/default_foam_000_ideal_CT_I0_1000.txt
parser = argparse.ArgumentParser()
parser.add_argument("sharelink", default="", type=str)
parser.add_argument("input_filename", default="", type=str)
parser.add_argument("--bench_type", default="default", type=str)
parser.add_argument("--data_dir", default="foam_ct_data_accepted", type=str)
parser.add_argument("--output_dir", default="foam_ct_data_segmented", type=str)
parser.add_argument("--timing_dir", default="timings", type=str)
parser.add_argument("--index", default="0", type=str)
args = parser.parse_args()

share_link = args.sharelink
bench_type = args.bench_type
output_filedir = args.output_dir
timing_dir = args.timing_dir
index = args.index
input_base = "{}_{}".format(bench_type, args.input_filename)
required_file = os.path.join(args.data_dir, "{}.txt".format(input_base))

# Variables that will be overwritten accoring to pattern:
print("Checking for: {}".format(required_file))
input_filedir = "foam_ct_data"
utils_path = "idmc_utils_module.py"

import numpy as np
import importlib
import matplotlib.pyplot as plt
import time
import scipy.ndimage as snd
import skimage
import importlib.util
from mig.io import IDMCShare

script_name = os.path.basename(__file__).split(".")[0]
bench_file = os.path.join(timing_dir,
    "{}_{}_bench.csv".format(bench_type, script_name))
start = time.time()

datastorage = IDMCShare(share_link)
# No accepted data for that file exists
if not datastorage.exists(required_file):
    print("Failed to find required file: {}".format(required_file))
    exit(1)

spec = importlib.util.spec_from_file_location("utils", utils_path)
utils = importlib.util.module_from_spec(spec)
spec.loader.exec_module(utils)

# Segmentation method used:
```

```
# - Median filter applied to reduce noise
# - Otsu thresholding applied to get binary data
# - Morphological closing performed to remove remaining single-voxel noise

# Parameters
median_filter_kernel_size = 2

# Load data
filename_withouth_txt = input_base.split(bench_type)[1][1:]
input_data = os.path.join(input_filedir, filename_withouth_txt + ".npy")
print("Loading: {}".format(filename_withouth_txt))

ct_data = None
with datastorage.open(input_data, "rb") as _file:
    ct_data = np.load(_file)

utils.plot_center_slices(ct_data, title=filename_withouth_txt)

# Median filtering
data_filtered = snd.median_filter(ct_data, median_filter_kernel_size)
utils.plot_center_slices(
    data_filtered, title=filename_withouth_txt + " median filtered"
)

# Otsu thresholding
threshold = skimage.filters.threshold_otsu(data_filtered)
data_thresholded = (data_filtered > threshold) * 1
utils.plot_center_slices(
    data_thresholded, title=filename_withouth_txt + " Otsu thresholded"
)

# Morphological closing
data_segmented = skimage.morphology.binary_closing((data_thresholded == 0)) == 0
utils.plot_center_slices(
    data_segmented, title=filename_withouth_txt + " Otsu thresholded"
)

# Save data
filename_save = input_base + "_segmented.npy"
if not datastorage.exists(output_filedir):
    datastorage.mkdir(output_filedir)

output_file = os.path.join(output_filedir, filename_save)
with datastorage.open(output_file, "wb") as _file:
    np.save(_file, data_segmented)

stop = time.time()

if not datastorage.exists(timing_dir):
    datastorage.mkdir(timing_dir)
```

```python
if not datastorage.exists(bench_file):
    with datastorage.open(bench_file, 'w') as _file:
        _file.write("index, start, stop, time, size\n")

# Save timing data
with datastorage.open(bench_file, "a") as _file:
    content = "{},{},{},{},{}\n".format(
        index, start, stop, stop - start, ct_data.nbytes
    )
    _file.write(content)
```

Listing 14: segment_foam_data.py.

```python
#!/usr/bin/env python
# coding: utf-8
import argparse
import os

parser = argparse.ArgumentParser()
parser.add_argument("sharelink", default="", type=str)
parser.add_argument("input_filename", default="", type=str)
parser.add_argument("--data_dir",
    default="foam_ct_data_segmented", type=str)
parser.add_argument("--bench_type",
    default="default", type=str)
parser.add_argument("--output_dir",
    default="foam_ct_data_pore_analysis", type=str)
parser.add_argument("--timing_dir",
    default="timings", type=str)
parser.add_argument("--index",
    default="0", type=str)
args = parser.parse_args()

share_link = args.sharelink
bench_type = args.bench_type
output_filedir = args.output_dir
timing_dir = args.timing_dir
index = args.index
input_filename = os.path.join(
    args.data_dir, "{}_{}_segmented.npy".format(bench_type, args.input_filename)
)

# Variables that will be overwritten accoring to pattern:
# input_filename =
# "foam_ct_data_segmented/foam_000_ideal_CT_I0_1000_segmented.npy"
print("Checking for: {}".format(input_filename))
utils_path = "idmc_utils_module.py"

import numpy as np
import importlib
import matplotlib.pyplot as plt
import time
import scipy.ndimage as snd
import importlib.util

from skimage.morphology import watershed
from skimage.feature import peak_local_max
from matplotlib import cm
from matplotlib.colors import ListedColormap, LinearSegmentedColormap
from mig.io import IDMCShare

script_name = os.path.basename(__file__).split(".")[0]
bench_file = os.path.join(timing_dir,
"{}_{}_bench.csv".format(bench_type, script_name))
start = time.time()

datastorage = IDMCShare(share_link)
if not datastorage.exists(input_filename):
    print("Failed to find required file: {}".format(input_filename))
    exit(1)

spec = importlib.util.spec_from_file_location("utils", utils_path)
utils = importlib.util.module_from_spec(spec)
spec.loader.exec_module(utils)

# # Foam pore analysis
#
# - Use Watershed algorithm to separate pores
# - Plot statistics
#

# Load data
data = None
with datastorage.open(input_filename, "rb") as _file:
    data = np.load(_file)

utils.plot_center_slices(data, title=input_filename)
# Watershed: Identify separate pores

# distance map
distance = snd.distance_transform_edt((data == 0))

# get watershed seeds
local_maxi = peak_local_max(
    distance, indices=False, footprint=np.ones((3, 3, 3)), labels=(data == 0)
)
markers = snd.label(local_maxi)[0]

# perform watershed pore seapration
```

```
labels = watershed(-distance, markers, mask=(data == 0))

# Pore color mad
somecmap = cm.get_cmap("magma", 256)
cvals = np.random.uniform(0, 1, len(np.unique(labels)))
newcmp = ListedColormap(somecmap(cvals))

utils.plot_center_slices(-distance, cmap=plt.cm.gray, title="Distances")
utils.plot_center_slices(labels, cmap=newcmp, title="Separated pores")

# Plot statistics: pore radii
volumes = np.array([np.sum(labels == label) for label in np.unique(labels)])
volumes.sort()
# ignore two largest labels (background and matrix)
radii = (volumes[:-2] * 3 / (4 * np.pi)) ** (
    1 / 3
)  # find radii, assuming spherical pores
_ = plt.hist(radii, bins=200)

# Save plot
filename_withouth_npy = input_filename.split("/")[1].strip(".npy")
filename_save = filename_withouth_npy + "_statistics.png"

print("Loading: {}".format(filename_withouth_npy))

fig, ax = plt.subplots(1, 3, figsize=(15, 4))
ax[0].imshow(labels[:, :, np.shape(labels)[2] // 2], cmap=newcmp)
ax[1].imshow(labels[:, np.shape(labels)[2] // 2, :], cmap=newcmp)
_ = ax[2].hist(radii, bins=200)
ax[2].set_title("Foam pore radii")

if not datastorage.exists(output_filedir):
    datastorage.mkdir(output_filedir)

print(output_filedir)
print(filename_save)
with datastorage.open(os.path.join(output_filedir,
filename_save), "wb") as _file:
    plt.savefig(_file)

stop = time.time()
if not datastorage.exists(timing_dir):
    datastorage.mkdir(timing_dir)

if not datastorage.exists(bench_file):
    with datastorage.open(bench_file, 'w') as _file:
        _file.write("index, start, stop, time, size\n")

# Save timing data
with datastorage.open(bench_file, "a") as _file:
    content = "{},{},{},{},{}\n".format(index, start, stop,
    stop - start, data.nbytes)
    _file.write(content)
```

Listing 15: foam_pore_analysis.py.

```
import os
import time
import subprocess
from mig.io import IDMCShare


def run_bench(program, args):
    cmd = ["python3", program]
    cmd.extend(args)
    subprocess.run(cmd)

def ensure_datastorage_dir(datastorage, directory):
    if not datastorage.exists(directory):
        datastorage.mkdir(directory)


if __name__ == "__main__":
    prefix = "foam_processing"
    steps = [
        "initial_porosity_check.py",
        "segment_foam_data.py",
        "foam_pore_analysis.py",
    ]
    throttle = True

    if throttle:
        steps_per_min = 3

    steps_since_delay = 0
    delay_timestamp = int(time.time())
    share_link = "SHARELINK"
    bench_type = "default"
    datastorage = IDMCShare(share_link)
    files = [f.strip(".npy") for f in sorted(datastorage.list("foam_ct_data"))]
    iterations = 10

    base_data_dir = "foam_ct_data"
    if not datastorage.exists(base_data_dir):
        print("Required data dir: {} is missing".format(base_data_dir))
        exit(1)

    for i in range(iterations):
        # Create the required directories
        accepted_dir = "{}_{}_{}_accepted".format(bench_type, i,
        base_data_dir)
        discarded_dir = "{}_{}_{}_discarded".format(bench_type, i,
        base_data_dir)
        pore_analysis_dir = "{}_{}_{}_pore_analysis".format(bench_type, i, base_dat
        segmented_dir = "{}_{}_{}_segmented".format(bench_type, i,
        base_data_dir)
        timing_dir = "{}_{}_times".format(bench_type, i)

        ensure_datastorage_dir(datastorage, accepted_dir)
        ensure_datastorage_dir(datastorage, discarded_dir)
        ensure_datastorage_dir(datastorage, pore_analysis_dir)
        ensure_datastorage_dir(datastorage, segmented_dir)
        ensure_datastorage_dir(datastorage, timing_dir)

        for idx, input_filename in enumerate(files):
            for step in steps:
                program_path = os.path.join(prefix, step)
                args = [
                    share_link,
                    input_filename,
                    "--bench_type",
                    bench_type,
                    "--index",
                    str(idx),
                    "--timing_dir",
                    timing_dir
                ]

                if step == "initial_porosity_check.py":
                    args.extend([
                        "--output_accepted_dir", accepted_dir,
                        "--output_discarded_dir", discarded_dir])
                if step == "segment_foam_data.py":
                    args.extend([
                        "--data_dir", accepted_dir,
                        "--output_dir", segmented_dir
                    ])
                if step == "foam_pore_analysis.py":
                    args.extend([
                        "--data_dir", segmented_dir,
                        "--output_dir", pore_analysis_dir
                    ])

                run_bench(program_path, args)
                steps_since_delay += 1
                time_since_delay = int(time.time()) - delay_timestamp
                if (
                    throttle
                    and steps_since_delay >= steps_per_min
                    and time_since_delay < 60
                ):
                    wait_for = 60 - time_since_delay
                    print("throttle, waiting for: {}".format(wait_for))
                    time.sleep(wait_for)
                    delay_timestamp = int(time.time())
                    steps_since_delay = 0
```

Listing 16: mig_utils_benchmark_runner.py.

Appendix F

MEOW API

Listing 17: The MiG MEOW Workflows API.

Appendix G

Corc APIs

```
class Orchestrator:

    options = None
    _is_ready = False
    _is_reachable = False
    _resource_id = None

    def __init__(self, options):
        self.options = options

    def is_ready(self):
        return self._is_ready

    def is_reachable(self):
        return self._is_reachable

    def endpoint(self, select=None):
        raise NotImplementedError

    def endpoints(self, select=None):
        raise NotImplementedError

    def get_resource(self):
        raise NotImplementedError

    def poll(self):
        raise NotImplementedError

    def setup(self, resource_config=None, credentials=None):
        raise NotImplementedError

    def resource_id(self):
        return self._resource_id

    def tear_down(self):
        raise NotImplementedError
    @classmethod
    def adapt_options(cls, **kwargs):
        """Used to adapt the orchestrators options if required
        before they are passed to the validate_options"""
        return {}

    @classmethod
    def load_config_options(cls, provider="", path=None):
        raise NotImplementedError

    @classmethod
    def make_resource_config(cls, provider, **kwargs):
        return None

    @classmethod
    def make_credentials(cls, **kwargs):
        return None

    @classmethod
    def validate_options(cls, options):
        raise NotImplementedError
```

Listing 18: Corc Orchestrator API.

```
class Scheduler:
    def provision_storage(self, config):
        raise NotImplementedError

    def prepare(self, config):
        raise NotImplementedError

    def submit(self, task):
        raise NotImplementedError

    def list_scheduled(self):
        raise NotImplementedError

    def retrieve(self, job_id):
        raise NotImplementedError

    def remove(self, job_id):
        raise NotImplementedError
```

Listing 19: Corc Scheduler API.

Listing 20: Corc Storage API.

Appendix H

Corc Configurations

```
corc:
  configurers:
    ANSIBLE: {}
  job:
    capture: true
    meta:
      debug: false
      env_override: true
      name: ''
      num_jobs: 1
      num_parallel: 1
    output_path: /tmp/output
    working_dir: ''
  providers:
    ec2:
      instance:
        image: ami-0f18ced0fd6aae5c2
        name: instance
        size: t1.micro
        ssh_authorized_key: ''
      profile:
        config_file: ~/.aws/config
        credentials_file: ~/.aws/credentials
        name: default
    oci:
      cluster:
        domain: ''
        image: nielsbohr/mccode-job-runner:latest
        kubernetes_version: ''
        name: cluster
        node:
          availability_domain: 'lfcb:EU-FRANKFURT-1-AD-1'
          id: ''
          image: Oracle-Linux-7.9-2020.11.10-1
          name: NodePool
          node_shape: VM.Standard2.1
          size: 1
      instance:
        availability_domain: ''
        operating_system: CentOS
        operating_system_version: '7'
        shape: VM.Standard2.1
        ssh_authorized_keys: []
      profile:
        compartment_id: ''
        name: DEFAULT
      vcn:
        cidr_block: 10.0.0.0/16
        display_name: VCN Network
        dns_label: vcn
        id: ''
        internetgateway:
          display_name: default_gateway
          id: ''
          is_enabled: true
        routetable:
          display_name: default_route_table
          id: ''
          routerules:
          - cidr_block: null
            destination: 0.0.0.0/0
            destination_type: CIDR_BLOCK
            id: ''
        subnet:
          cidr_block: 10.0.1.0/24
          display_name: worker_subnet
          dns_label: workers
          id: ''
  storage:
    credentials_path: /mnt/creds
    download_path: ''
    enable: false
    endpoint: ''
    input_path: /tmp/input
    output_path: /tmp/output
    s3:
      bucket_id: ''
      bucket_input_prefix: input
      bucket_name: ''
      bucket_output_prefix: output
      config_file: ~/.aws/config
      credentials_file: ~/.aws/credentials
      name: default
    upload_path: ''
```

Listing 21: Corc Configuration Example.

```
corc:
  configurers:
    ANSIBLE: {}
  job:
    capture: true
    meta:
      debug: false
      env_override: true
      name: ''
      num_jobs: 1
      num_parallel: 1
    output_path: /tmp/output
    working_dir: ''
  providers:
    aws: {}
    oci:
      cluster:
        domain: ''
        image: nielsbohr/mccode-job-runner:latest
        kubernetes_version: ''
        name: cluster
        node:
          availability_domain: lfcb:EU-FRANKFURT-1-AD-1
          id: ''
          image: Oracle-Linux-7.8-2020.09.23-0
          name: NodePool
          node_shape: VM.Standard2.4
          size: 1
      instance:
        availability_domain: ''
        operating_system: CentOS
        operating_system_version: '7'
        shape: VM.Standard2.1
        ssh_authorized_keys: []
      profile:
        compartment_id: 'ocid1.compartment.oci'
        '..aaaaaaaaqbl3z74j5tmii74tgtukhmkv5lx3uwsvwddynaawpyddzsmw27aa'
        name: DEFAULT
      vcn:
        cidr_block: 10.0.0.0/16
        display_name: VCN Patch Network
        dns_label: vcn
        id: ''
        internetgateway:
          display_name: default_gateway
          id: ''
          is_enabled: true
        routetable:
          display_name: default_route_table
          id: ''
          routerules:
          - cidr_block: null
            destination: 0.0.0.0/0
            destination_type: CIDR_BLOCK
            id: ''
        subnet:
          cidr_block: 10.0.1.0/24
          display_name: worker_subnet
          dns_label: workers
          id: ''
  storage:
    credentials_path: /mnt/creds
    download_path: ''
    enable: false
    endpoint: 'https://ku.compat'
    '.objectstorage.eu-frankfurt-1.oraclecloud.com'
    input_path: /tmp/input
    output_path: /tmp/output
    s3:
      bucket_id: ''
      bucket_input_prefix: input
      bucket_name: ''
      bucket_output_prefix: output
      config_file: ~/.aws/config
      credentials_file: ~/.aws/credentials
      name: default
    upload_path: ''
```

Listing 22: Corc McStas Benchmark Configuration.