TRONGE, JAKE, M.S., May 2022 COMPUTER SCIENCE

ORCHESTRATION OF HPC WORKFLOWS: SCALABILITY TESTING AND CROSS-SYSTEM EXECUTION (58 pages)

Thesis Advisor: Dr. Qiang Guan

HPC and scientific workflows change over time and often require more resources, different parameters and environments. Workflows may eventually have a need for more resources than are available on a single platform. Also, as applications evolve to fit new requirements and design goals, their performance and how well they can scale on existing hardware needs to be measured to ensure optimal application development and design. New workflows, as well as existing workflows, will require next-generation workflow engines that are able to handle multiple platforms, testing of scalability as well as communication and monitoring of applications, all designed to allow for greater portability and reproducibility. In this work I will demonstrate extensions to the Build and Execute Environment (BEE) workflow orchestration system, as well as additional code known as BeeSwarm, that are used for testing scalability and performance of existing HPC applications. This work also encompasses new design choices in BEE that allow for running workflows across multiple underlying systems, thus not limiting workflows to only the resources that are available on a single system.

ORCHESTRATION OF HPC WORKFLOWS: SCALABILITY TESTING AND CROSS-SYSTEM EXECUTION

A thesis submitted

to Kent State University

in partial fulfillment of the requirements

for the degree of Master of Science

by

Jake Tronge

May 2022

Thesis written by

Jacob Tronge

B.S., Kent State University, 2020

M.S., Kent State University, 2022

Approved by

[illegible]

_______________________________________, Advisor
Qiang Guan

_______________________________________, Chair, Department of Computer Science
Javed I. Khan

_______________________________________, Dean, College of Arts and Sciences
Mandy Munro-Stasiuk

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Qiang Guan, for his guidance and expert knowledge for the past two years. Despite not being able to meet in person for more than a year, he introduced me to HPC and was able to help me get started with this work, giving suggestions and possible solutions to problems along the way.

I would also like to thank all those on the BEE team at Los Alamos National Lab who have provided their insight and knowledge of both BEE and HPC in general. Many of the ideas in this work are founded on the discussions that we had in meetings and online.

I'd also like to specially thank Tim Randles and Patricia Grubel, who both took time to give feedback on code, writing and ideas. Without their help, I would not have been able to produce this work.

CHAPTER 1

Introduction

Modern HPC workflows are based on a strong foundation of existing software, existing techniques and models. As Exascale platforms begin to become main stream and existing applications need to take advantage of these powerful resources, tools for environment management, dependency management and others become increasingly important. Applications will also begin to grow and their requirements will change over time. Thus these applications will need to be portable, having the ability to run from a developer's laptop all the way up to the most powerful HPC and cloud systems. New systems for measuring performance and scalability of existing applications are also necessary in order to allow applications to continue to fully utilize systems that they are run on. Workflow orchestration systems are needed here to solve these problems and create a level of abstraction that can be built upon for future work.

Existing HPC applications are typically developed in collaboration with a large number of people from different organizations and backgrounds. Good development practices and testing practices are necessary as projects grow over time. Continuous Integration (CI) is used by many of the top HPC development groups, including many of those in the Next-Generation Code project at Los Alamos National Lab [17]. Current CI pipelines typically work with correctness testing, ensuring that applications produce the expected results. In comparison with other backgrounds, correctness testing with scientific software typically requires domain knowledge and comparison of results with expected values. This becomes complicated by the fact that scientific programs often are designed to produce probabilistic results, rather than the same results for every run.

While correctness testing is one important aspect of testing in HPC programs, another equally important step is that of performance and scalability testing. Testing scalability is a challenge, since HPC applications are designed to run on multi-node systems with specialized hardware and software that developers may not always have consistent or timely access to. Existing CI pipelines are normally launched on a low-end VM in the cloud, which has an extremely limited capability for increasing the number of resources to compare scalablity results, such as with strong and weak scaling. Therefore most existing scalability testing is typically done manually which can be error-prone and also difficult to compare changes in performance between different versions of an application.

The idea behind BeeSwarm, is to extend existing CI pipelines by allowing a tool to connect to external resources, launch scaling tests and then record results for later analysis. This ensures that results are reproducible and that developers are able to note over time how performance has changed with their application. While in this work we focus on launching these scaling tests on external cloud resources, the design of BEE is a plugable architecture which can allow for launching on other systems, including custom-HPC systems. This would ensure that hardware is even better suited to the application tests, while still allowing for automatic testing.

CI scalability testing is but one issue in the current HPC field of research. Scalability testing typically works with one component of a larger workflow; that is to say it is attempting to test how well that smaller component will run when given a certain input size. But performance often involves more than just the performance of a single component. It is, instead, the combined performance of a large number of jobs that together form a workflow. Workflows are sets of jobs with dependencies between them. These could have relations between each other through the production of file data, through external databases or even through in-memory transfers. There are also in-situ workflows, where some processes may be producing data and other processes may be able to simultaneously use that data. Some works use the term pipeline to refer to workflows, but I avoid this here since I am also talking about CI pipelines, which are different in that they run sequences of test cases on an application rather than running multiple applications on some input in order to produce some sort of output.

The total performance of a workflow is limited not only by the amount of resources that are available, but also by the kind that are available, the wait times incurred on particular systems, and the costs needed for running tasks. Workflows may have smaller running High-Throughput Compute (HTC) jobs that are better designed to run on HTC platforms. For example, some set of Python processes or other scripts may be designed to process small chunks of a larger data set, but do not need to communicate between each other in an immediate manner. There may also be more traditional HPC jobs, such as an MPI job that requires a huge amount of nodes with a powerful interconnect in order to solve some problem within a larger workflow. Jobs with different requirements will require different resources in order for workflows to run in an efficient manner. Workflows themselves, when composed of a larger number of these types of jobs, may no longer run efficiently on a single HPC system.

Here I propose additions to the BEE workflow orchestration system that would allow for workflows to run across system boundaries. This means that applications would be able to launch on cloud systems and HPC systems at the same time. The idea is to allow all available resources to be used for a workflow when needed. While this may improve usage of resources and workflow performance over time, it also introduces its own complexities. Data transfer and other mechanisms may become more difficult with this feature. However, in some cases this automation can be extremely useful.

There are a number of existing workflow orchestration systems that attempt to address some of these concerns, but not all of them. These include StreamFlow [14], Cromwell [71], Pegasus [49] and Apache Airflow [27], as well as many others. Both StreamFlow and Cromwell are designed to support the Common Workflow Language (CWL) [15] standard for representing workflows. StreamFlow, similar to BEE, is designed to allow for cross-execution of workflows with some tasks on both HPC systems and cloud systems, but takes a global view of scheduling for running HPC applications. Cromwell also has support for a number of different backends, including both cloud and HPC, but doesn't fully support cross-system execution of workflows. As far as we know, very few of these applications have been demonstrated running in CI environments or for scalability testing purposes of applications. Nevertheless, these applications have each contributed a great deal of research to the field of workflow orchestration.

To wrap up this introduction, I'll list the key contributions of my work:

- Development and design of a CI scalability testing component, BeeSwarm, for HPC and MPI applications, as well as demonstrations of it's use on three different HPC applications;
- Extensions to `BEE` that can allow for workflows to run across system boundaries;
- A cross-system scheduling component for `BEE` that can be used for scheduling more complex workflows across systems and

- Simulation of different workflow scheduling algorithms that can be used within BEE.

This work includes some results and figures that we previously published in collaboration with the BEE team in 2021 [63, 62] as well as additional scheduling work that has not yet been published.

The remaining sections of this thesis are organized as follows. In chapter 2, I will go over the major concepts and existing research upon which BEE is based, as well as the existing BEE code. In chapter 3 I discuss other research that is related to this work. In chapter 4 I go over the proposed work as well as the design of my solution. In chapter 5 I then go over a number of the results produced under different scenarios, including the scheduling simulation results. In chapter 6 I discuss our results, some important design issues as well as future work. Finally in chapter 7 I conclude this work with a short overview of the solution and results.

CHAPTER 2

Background

In this section I will go over some of the many concepts and ideas that are related to this work. This work combines a number of different related fields and research. High-Performance Computing (HPC) is a foundation of this work and recent research in this field has created new areas of study, including dealing with the management of application complexity and portability through the use of containers. Scientific workflows and ensembles are also one of the cornerstones of this work and research has been ongoing into this area for more than 20 years. I will also go over some related areas that are not entirely specific to HPC and scientific applications, but that are needed for current development practices and future software projects. This includes background on scheduling, CI and cloud computing as well.

2.1 Scientific Software Requirements

To start out with, there are a number of key goals and requirements that are needed for modern scientific software. These goals are designed to help make research and communication of that research easier. These concepts are important to workflow orchestration systems and their design.

Reproducibility refers to the ability to rerun certain experiments on the same input data and get the same results back for an experiment. In the past this has been extremely hard for scientists running larger applications because different HPC systems require different configurations that may require a great deal of work and time to produce.

Provenance relates to the management and documentation of data in a manner such that the same result data can be achieved by another run of an experiment or workflow. Provenance is related to reproducibility in the sense that without data provenance there would be no way to reexecute an experiment under the same initial conditions as a previous run. Provenance does not necessarily mean storing every byte of data that was used in an experiment, but rather keeping a careful log of what data was used, either in terms of file hashes or by using some sort of centralized data storage.

Automation in recent years has become increasingly important for scientific applications and workflows. Automation allows both scientists and developers to focus on the key parts of an application while other software handles the mechanisms that are not directly related to what their application does. Automation could relate to, for example, the execution of a program on a cluster, or to the automatic configuration of specific applications that their software depends on. Previously, developers of HPC software often would do many of these tasks manually or with small scripts. As complexities arise over time doing tasks manually becomes burdensome and error prone. The more automation of a process, the less likely there are to be manual errors that could throw off or ruin an experiment.

2.2 High Performance Computing

High Performance Computing is an area of research that encompasses a huge number of concepts and many disciplines, including bioinformatics, computational physics and computational chemistry. The term HPC itself often refers to the field as a whole, but could be defined more specifically as the research and design of highly-parallel and connected programs that often make use of specialized hardware, including specially designed processors, interconnects and accelerators [64]. Furthermore, researchers also have defined the term High Throughput Computing (HTC), as being the scheduling, research and design of systems and applications that are made up of many tasks that typically do not have parallel dependencies linking them together. However these two definitions don't seem to take into account the fact that HPC programs are often much more complex. The typical project may be composed of many more steps and applications that are all linked together with different types of dependencies. In fact, for many HPC programs and frameworks, domain experts and developers work on a large set of programs that are designed to work together to produce results, filter and process those results and then pass data onto other programs for further analysis. This set of applications and dependencies between applications can be grouped together into a workflow, a key term which forms the basis of BEE and this thesis.

2.3 Cloud Computing

Following the explosion of the internet in the 90s and the continual need for businesses, organizations and governments to launch web-based applications and software, cloud computing started as a way for people to easily launch servers to get applications up and running. Cloud computing offers a quick way for people to start up a server running custom software that is connected by way of high-speed connection to the internet. Users don't have to worry about manual upkeep of hardware, and as failures occur the cloud providers usually have mechanisms for moving virtual machines to other systems that are up. **Service-Level Agreements (SLAs)** are contracts between cloud providers and users that set conditions and guarantees under which services are to be provided. For instance, some rules may guarantee that requests will be handled within a certain amount of time, while at the same time users may have a limited number of requests per unit of time. The term **Infrastructure-as-a-Service (IaaS)** encompasses the provisioning and use of these general compute resources and since the start of cloud computing has become useful for more than just web applications [51].

In the past 10 years many researchers have begun to experiment using IaaS for launching and managing complex HPC applications [40, 20, 39]. While IaaS solves some of the problems related to maintaining on-site HPC systems, there are still ongoing problems for running applications in the cloud. Some are more related to the types of hardware offered by cloud providers: while there are often high-end compute systems that can be provisioned there usually isn't a high-speed interconnect option avalable, which is a necessity for many major HPC applications. Another problem with using IaaS for running applications in the cloud is that users often have to deal directly with operating system and other configuration items that are lower-level than their own applications. This can be an error-prone and tedious process, especially for developers who would rather be working on application-level code, rather than the details of the OS.

To combat this configuration problem, which is not only problematic for HPC but for other fields as well, other cloud-based services, such as **Software-as-a-Service (SaaS)** and **Platform-as-a-Service (PaaS** are being used. **Software-as-a-Service (SaaS)** is where certain special types of software are provided to users over the Internet, which could include databases, data storage servers, shared-editing tools, etc [51]. On the other hand, **Platform-as-a-Service (PaaS)** is where cloud providers allow users to run applications on special operating systems or environments that are managed by the cloud providers [51]. The benefits of PaaS is that users can run their applications in the cloud without having to worry about those low-level configuration details, which are instead managed by the provider. In most cases users are still able to make some configuration changes, but these are limited in nature.

Although there are problems with running HPC programs in the cloud and there is great deal of improvements that need to be made, the cloud can still prove to be very useful for certain types of workflows and scenarios. For instance, when systems are overloaded, users may want to launch some jobs in the cloud in order to shorten overall completion time of workflows. In other cases cloud computing can be useful for testing purposes, as I will demonstrate with the BeeSwarm tool.

2.3.1 Autoscaling

Autoscaling is an interesting topic that I mention briefly here because I believe it has relevance to HPC workflows which have different resource requirements over the total execution time of a workflow. Autoscaling denotes the ability of a cloud system or HPC system to dynamically change the amount of compute resources used during runtime. There are many approaches to this and there is a wide field of research. Chen et al. [11] presents a useful survey of different scaling terms, techniques and research, and although it is focused on autoscaling in the cloud, it also has some relevance to HPC.

2.4 Containers

Containers emerged as one way to abstract application dependencies and environment management that can be extremely complex for the average software project, and more so for the average HPC project. Modern containers are based on the namespace feature of the Linux kernel [59]. There are six different namespaces that are available from the Linux kernel. These are used to isolate processes from the others running on a given system. There are a number of example container tools, the most notable one being Docker [2], as well as more recent additions such as Podman [52] and others. Many of these container runtime tools were originally designed for, and still are in most cases, being run as privileged user on a Linux system. For security reasons this is not suitable for existing HPC systems. HPC systems are usually multi-user systems that rely on complex software stacks and other configurations. It is not feasible to run many of the existing container tools on HPC because of the security concerns of requiring users with privileged accounts to be able to launch and run their containers. Because of the sheer number of users that may need to run jobs on a system it is also not feasible to have containers managed by a select few administrators with privileged accounts. Given the proven security of existing HPC tools as well as the infrastructure and the very design of HPC systems, there is a need for container tools that can run jobs at the user-level and thus ensure security and maximum user-level development.

Given these issues with existing container tools, a number of different HPC-specific container run times have been introduces, including Singularity (or Apptainer) and Charliecloud.

2.4.1 Singularity (Apptainer)

Singularity [42] is a major HPC container runtime that is written in Go. Singularity includes their own container definition format, as well as a special container image format, but also still includes support for Dockerfiles, a format that has become ubiquitous in the last 10 years for recording container definitions.

Singularity includes a number of HPC features and is also compliant with the Open Container Initiative (OCI) standard [34]. This allows singularity to work with container scheduling tools such as Kubernetes and makes it easier to control every aspect of container runtime environments. While Singularity has been designed with HPC in mind, there are some issues with the container build process that have not been resolved yet. Singularity's build process typically requires root privileges for the general case. They include a special fakeroot option [33], but this requires configuration of user ID and group ID mappings in the `/etc` director. Both of these require configuration and privileges that are not given to standard users of most HPC clusters and this makes it very difficult to build and use Singularity containers. One possible solution they offer is a container build service: when building a container, users can submit their application to a build service. A build service would need some sort of service installed either in the cloud or within an HPC center.

2.4.2 Charliecloud

Charliecloud is a lightweight container runtime engine that is designed specifically for running on HPC systems [55]. Charliecloud itself is made up of a number of C, Python and shell scripts that are used to build containers, push/pull containers to repositories, manage the container files and various formats and finally execute container binaries in an unprivileged environment.

Charliecloud uses the user namespace feature of the Linux kernel to allow for isolation of container environments and dependencies. A number of directories from the host environment are bind-mounted into the container environment, including, the user's home directory, /dev, /proc and /sys.

Charliecloud also includes an unprivileged build tool for building containers on HPC systems [54]. The tool allows for building containers as an unprivileged user from a Dockerfile using the most common distributions as a base. During the build process, Charliecloud allows privileged commands to be run by installing a version of the `fakeroot` [1] tool into the container.

2.5 Continuous Integration (CI)

Continuous Integration has become a necessity for most modern software. Applications need to be tested with different methods using different configurations. Before CI, tests would be run locally by a developer before or directly after making a commit to a repository. Over time, however, as applications have grown in complexity, it has become more difficult to run tests on a user's local computer, especially if they are long-running or require a large amount of resources. Each user may also have different development environments, such as one developer may work on a Windows machine, while another may work on a Mac. CI is needed to ensure that all facets of a project are tested in all environments that the project is expected to support.

Until recently, CI has largely focused on the correctness of applications, rather than how those applications perform under certain conditions. For HPC, correctness is extremely important, but also just as important is performance and scalability of an application. This is even more difficult to test locally and is also difficult to test in most existing CI environments that are provided by default. Most CI VMs are designed as small, low-power machines that are able to compile code, run tests and then exit quickly. They aren't designed for running scaling jobs or other more complicated tasks with more resources. This is where BeeSwarm comes into play as a CI tool that integrates with the cloud to allow for these types of scaling tests.

2.6 HPC Parallel Programming and Performance Measurement

Measuring performance of HPC applications is usually much different than performance profiling that is done with normal applications. HPC applications are typically launched on multi-node clusters and involve processes that utilize some sort of parallel processing library for performance.

The Message-Passing Interface (MPI) [3] is one of the most-widely used standards for programming parallel applications in HPC centers. A typical MPI application is made up of many processes, or tasks, that are all assigned a unique ID or rank. Processes can send messages to each

[1]https://man.cx/fakeroot(1)

other using functions such as `MPI_Send` and `MPI_Recv`, as well as a collection of other more powerful messaging functions. The individual processes themselves are typically launched and managed by a batch scheduler such as Slurm [56] or IBM LSF [31]. The three applications that I take a look at and test with BeeSwarm in this work all utilize MPI.

Measuring the performance of these types of applications is usually nontrivial and can require expert knowledge of the specific application, but there are a couple key methods that are useful in guiding the process. Most HPC performance measurements will typically take a look at two important types of scaling: strong scaling and weak scaling. **Strong scaling**, which is what I will focus on with my demonstrations in this work, is where a fixed input data size for an application is chosen and the application is then run on increasing numbers of compute resources. Most strong scaling tests increase compute resources with powers of two (i.e. 2 nodes, 4 nodes, 8 nodes, ...). An application is said to show good scalability or scaling if the execution time of the application decreases at the same rate that the compute resources are increased. **Weak scaling** is different in that instead of keeping the problem size fixed, the problem size is increased at the same rate that the number of compute resources are increased. Under weak scaling an application is said to exhibit good scalability or scaling if the total execution time remains about the same as both problem size and resources are increased. Both measures are useful for doing scalability analysis on HPC applications. Weak scaling in particular might be more useful for applications that are inclined to not do well with strong scaling, but are able to handle more data with larger amounts of resources. In this work I only focus on doing strong scaling, but BeeSwarm can easily do weak scaling as well.

2.7 Workflows

Workflows are the basic building blocks of scientific and HPC software. Workflows are designed to automate processes of data management, execution and runtime environments and dependencies between jobs [24]. Workflows are designed to abstract underlying infrastructure, dependencies and other lower-level computing issues. Abstracting these lower-level interfaces allow scientists and other domain experts to focus on developing and utilizing software for their specific use-cases. In the past scientists often had to write complicated submission scripts and other scripts on top of these to manage job execution and monitoring. While this method worked in the past, these scripts are typically not portable between systems and can become a hassle to maintain over time as underlying infrastructure is updated. These older script-based workflows are both difficult to update and do not easily show the data relationships between jobs. Workflow management systems attempt to avoid these issues and to make the design and dependencies between jobs in a workflow explicit and easy to manage.

2.7.1 Workflow Classifications

Workflows can be classified using a number of different methods. Da Silva et al. [24] characterize a number of different workflow types using execution methods, dependencies, data exchange and storage as classification methods. They also enumerate some of the many workflow management systems, which by now has only increased in number and type. Workflows can typically be categorized in terms of their execution models [24]:

- **sequential** workflows process one task at a time, produce some data and then hand this data off to another task;
- **concurrent** workflows typically have multiple tasks that are dependent on other tasks, for instance one task may produce some input for another task to consume;
- **iterative** workflows typically involve a kind of execution loop where one task will execute and the results of that task will determine if another task ”loop” is run;
- **tightly coupled** workflows involve tasks that may send results periodically back and forth to each other and is in some ways similar to concurrent workflows;
- **external steering** workflows require some sort of user interaction in order to determine the next steps to be executed in a workflow.

Some other works also use the term **in-situ** to refer to workflows that can run several tasks at the same time, producing or analyzing data [19].

2.7.2 Common Workflow Language (CWL)

How we can represent workflows is also another key area of research for HPC workflow orchestration. The Common Workflow Language (CWL) [15] is a workflow standard based on YAML that many workflow orchestration engines have begun to converge on in recent years. Early workflow orchestration systems were designed with custom workflow specification languages. Each language would be specific to a given workflow engine, and this made it extremely difficult to port workflows from one system to another. Having a standard format is key to maintaining and using existing workflows and also is extremely important to support for modern workflow orchestration systems.

2.8 Scheduling

There are a wide range of mechanisms for submitting and managing jobs on different types of clusters and cloud systems.

Many existing schedulers have been built using heuristics, such as the Backfill algorithm, First-Come-First-Serve (FCFS) and Shortest-Job First (SJF). Backfill, in practice, has proved to be extremely useful in many HPC systems. For example, the batch scheduler Slurm uses Backfill as its default scheduling algorithm [57]. Backfill is designed to allocate resources to short running lower-priority jobs if their execution will not affect already scheduled jobs. There is also other research which has used modified versions of FCFS and SJF for different scheduling tasks. These heuristic algorithms often work well in practice because they typically don't require a long time to make a scheduling decision and can provide good allocations for single-task applications. However, when workflows are used, which often consist of many interconnected job, these types of algorithms may not always provide the best allocations.

For workflow scheduling, where we must schedule many interdependent tasks on given computing resources, there is typically some sort of optimization algorithm involved which requires optimization of a number of competing parameters, including cost, resource types and amounts. This type of algorithm has shown by some to be NP-hard [9]. Thus for more complex workflows, some sort of algorithm that is an approximation, or that can be shown to be on average close to the true optimal schedule, need to be used.

Liu et al. [44] present a survey of workflow execution. They categorize workflow scheduling algorithms into four major categories:

- **task** scheduling is where scheduling is done task-by-task without looking at more than one task at a time

- **path** scheduling uses individual paths or chains of dependent tasks within a workflow to schedule tasks
- **bag-of-task** scheduling schedules sets of independent tasks within a workflow
- **workflow** scheduling attempts to schedule whole workflows at a time

Workflow scheduling algorithms can also be further classified based on the optimization metrics that they use. Some algorithms may attempt to optimize on cost, while others may attempt to optimize based on a deadline. Other algorithms may also be constrained to a set deadline or a set budget that must be fulfilled.

Many of these algorithms are based on solving some sort of optimization problem with multiple values that need to be minimized or maximized. Within the last 10-15 years, some interesting algorithms have been proposed for solving these types of multi-objective optimization problems. Some are based on the idea of the pareto set, which refers to a set of solutions where the results cannot be made better without making other components of the solution worse. Once can visualize this well with a graph, where objective values are along the axes, and the pareto set refers to those data points that lie along the upper edge or the lower edge, depending on whether the problem is a minimization or a maximization problem. Vachhani et al. [65] present a survey of a number of algorithms that can be used for determining Pareto set solutions or approximations.

One distinction that I also try to make in this work, is the distinction between user-level scheduling and system-level scheduling. Much of the scheduling research in the field of cloud computing, as well as in scheduling specific to supercomputers within HPC centers, deals with a number of major issues. Some of these are more concerns of the management and system administrators. These include issues such as fairness, fulfillment of Service-Level Agreements (SLAs) and load balancing. On the other hand scheduling can also take on a user-perspective. In this case, from the user's perspective, they care more about execution time (or makespans) of workflows, total cost of execution and whether or not they meet certain deadlines within a particular amount of time. All of these issues together make up the scheduling field in general, however, typically algorithms are geared towards one side or the other. Tradeoffs have to be made in designing a scheduling algorithm to fulfill a set of SLAs within a datacenter or cloud computing system, such that some users may not be able to always fulfill workflows in the exact time that they want, if those goals are outside of given SLAs. On the other hand schedules that are completely geared towards user needs will often cause problems for operators of data centers and HPC sites. Fulfilling all of a user's needs for their programs may lead to load imbalance or blocking of users who also need to use resources. All of these issues have been around since the time that multi-user computer systems first existed. I attempt to differentiate specifically between user-level scheduling and system-level scheduling because each side comes at the scheduling problem from different angles and each offer solutions that may not be compatible with each other.

2.9 Build and Execute Environment (BEE)

The Build and Execute Environment (BEE) [10] is an HPC workflow orchestration system built specifically for running on LANL's HPC infrastructure as well as for utilizing cloud and other available systems for running applications. BEE is based on the Common Workflow Language (CWL) standard to allow workflow executions to be standardized. BEE includes a number of components for monitoring workflows, launching workflows and handling workflow dependencies. A diagram of the main components of BEE is shown in Figure 1.

One of the key design pieces of BEE, is the use of an HTTP REST interface to allow for communication between components. Most of the major components of BEE present a well-defined REST interface. This ensures that over time as new components are added to BEE that they will easily be able to "fit in" with the existing components. An HTTP-based REST interface can be easily used with a number of different frameworks and languages, making it easy for extensions to be put together.

The main component of BEE is the Workflow Manager. This component loads and parses workflows into a graph database (GDB). Workflows are stored in the Common Workflow Language (CWL). Once parsed and loaded, workflows can be started by the user, at which point the workflow manager will use the graph database perform a DAG operation to determine the "READY" tasks that are able to launch. The workflow manager can then send individual tasks to the task manager for execution. When tasks complete execution, this causes the workflow manager to begin another DAG operation to determine what the next "READY" tasks are that can then be sent to the Task Manager. This process repeats until all tasks have been sent to the Task Manager and run. If tasks fail, then the workflow will be placed in a failed state, allowing users to notice this and examine

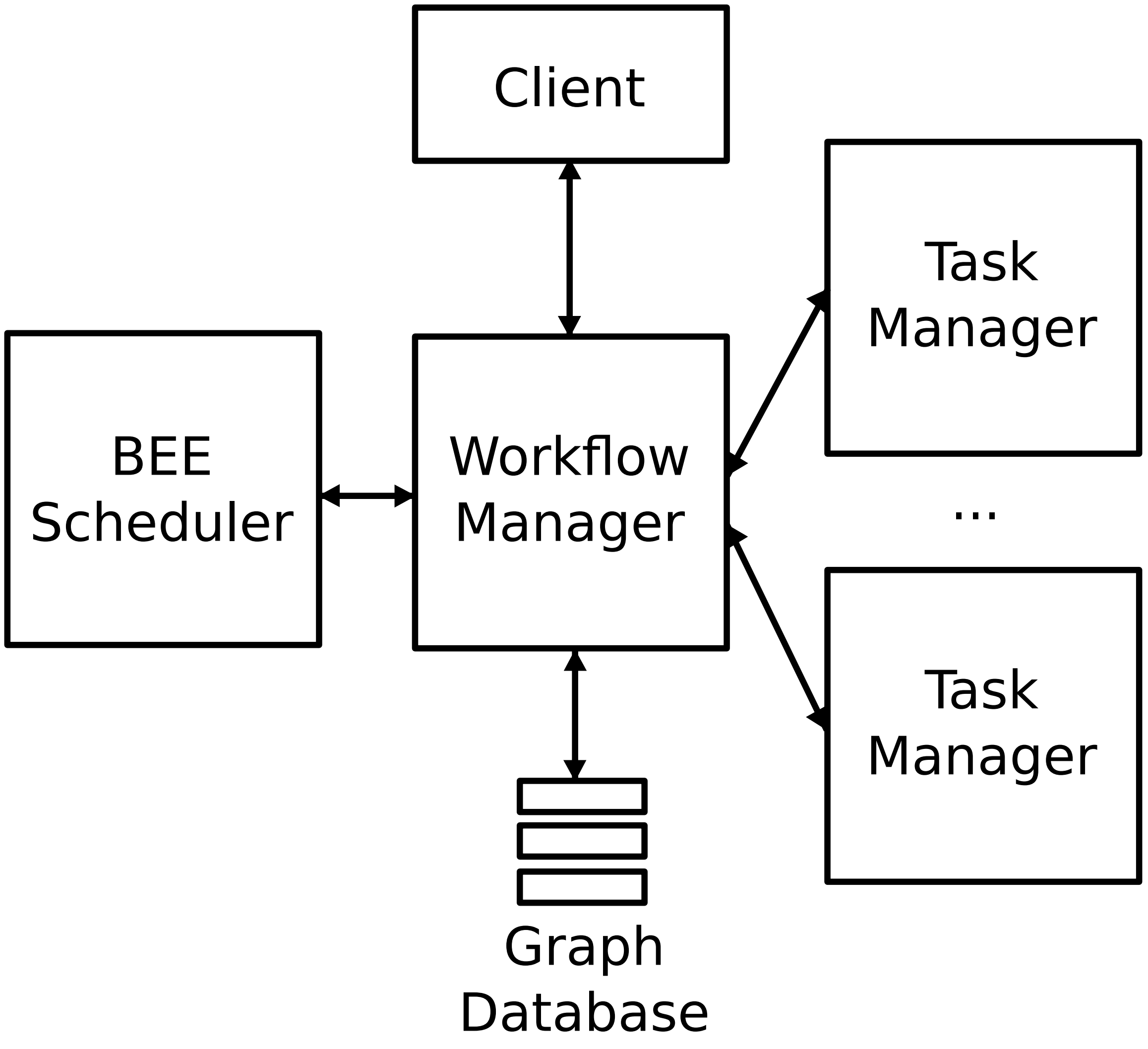

Figure 1: BEE Component Diagram.

the logs for runtime failures.

The Task Manager component of BEE is the daemon that actually interfaces with the underlying resource management system (such as IBM LSF [31], Slurm [56], or cloud system schedulers). Its role is to launch tasks, monitor them and then relay this information back to the workflow manager. Configuration of the task manager is designed per system. Thus this should allow us to have multiple task managers, if necessary. For example, a user could launch one task manager for submission on one HPC system that utilizes Slurm, while another task manager for a system that uses LSF. Then, the workflow manager could pick between launching tasks on one system versus the other, based on scheduling requirements.

The graph database is another important component of BEE. This is a Neo4j-based [46] graph database that is launched by the workflow manager as needed for parsing and storing workflow metadata. The database is launched from within a container as to ensure that it is easy to run on a variety of different platforms. The internal design of BEE ensures also that different types of database interfaces can also be added in the future. Storage of workflow metadata and other useful properties of parsed workflows is currently done through a special workflow interface that acts as an abstraction on top of the graph database code. Within the graph database code, we utilize the Cypher [47] query language to update and manage database entries.

BEE forms the base of my work, and the extensions that I will discuss in the coming chapters.

CHAPTER 3

Related Work

There has been a large amount of research into workflow orchestration, as well as with performance measurement and the use of CI tools for HPC and other fields. BEE [10], the basis for this work, is one of many workflow orchestration systems that are designed to run portable workflows.

3.1 Workflow Management Systems

There are a massive amount of Workflow Management systems that have been developed in the last 10-20 years. In fact there are so many that the applications have developed fragmented communities among which it can be difficult to share and reproduce workflows [16]. When attempting to solve a problem with existing workflow systems, many researchers have instead taken to creating entirely new systems instead of basing work on existing software. In this section I don't attempt to list every single available system, but instead to list a number of the major systems that have contributed significant research to the field or that have gathered a large userbase and community.

StreamFlow [14] is a recent workflow orchestration system that is designed for running multiple processes within one container and also for allowing hybrid execution across systems. It's written in Python and is under active development at the University of Torino. StreamFlow represents execution as three different components: a **model** which represents the deployment of the system, a **service** which links tasks of a workflow to the model, and a **resource** which is an active instance of a deployed model or system. StreamFlow is designed to run both HPC applications using MPI on Slurm-based clusters, as well as cloud-related tasks with Kuberenetes in the cloud.

Pegasus [49] is another workflow management system with support for a variety of scientific workflows. Pegasus has a much longer development history, starting around 2001, than many of the other systems listed here. The project offers good stability and a well-tested environment, as well as an active development community.

Cromwell [71] is another workflow orchestration system for scientific workflows. Written in

Scala, Cromwell has support for multiple backends, including AWS, Slurm, HTCondor, etc.. This tool also uses a REST API with support for querying and examining existing workflows. and a number of other features that make it suitable for HPC workflows. Cromwell is developed by the Broad Institute [35], supported by researchers from MIT and Harvard.

Apache Airflow [27] is a workflow or pipeline management system developed in Python. Airflow has a huge community as well as a massive amount of plugins and extensions that allow all kinds of workflows and integrations to be done. As with most other workflow systems, Airflow represents workflows as DAGs. However, for representing dependencies between tasks, Airflow includes two types of dependencies: the typical file-based method and then what is referred to as XComs or cross-communications where tasks can gather metadata about other tasks. Airflow also includes a GUI user interface and a number of other sub-components that make it easier for users to work with.

Arvados [8] is another existing workflow engine that provides support for a variety of workflows written in CWL. It is mainly designed for running workflows in the cloud, and it has support for AWS, Azure and Google Compute Engine. Arvados allows for decentralized execution and for multi-cluster workflow execution.

Toil [70] is another CWL-compliant workflow engine that is mostly designed for running cloud-based workflows. Toil also works with HPC batch schedulers, such as Slurm and LSF, but this functionality is considered community supported rather than a part of the core application. Toil works by storing workflow file information at a specified file path at runtime. It also has support for running service jobs, such as databases, that are designed to provide some sort of interface to other jobs within a workflow. As for data management, Toil integrates support for using rsync [18] to copy files from a local machine to a cluster and vice versa.

As stated before, there are other workflow managers that I do not list here, but these represent a number of the key actors in this wide field.

3.2 HPC Performance and Continuous Integration (CI)

There is extensive research and many tools for doing testing and performance analysis of HPC applications. However there are few CI tools designed for this kind of work, and I have not come across any that can directly test HPC workflows with CI.

Many tools focus on testing particular communication libraries, such as MPI. Vetter and Chambreau [69] introduce mpiP for profiling MPI applications. Their tool collects metadata during runtime and attempts to add as little overhead as possible. Results are stored into a single profile at the end of execution.

Many researchers have proposed different CI-based tools for testing performance of non-HPC applications. BlazeMeter [50] is a more recent CI tool for measuring application performance, but is mostly designed for web applications and for testing performance of different types of APIs. It however doesn't have support for HPC applications and the complexities that are introduced by testing scalability across multiple nodes. Jenkins [1], the widely used open-source CI tool, has a performance plugin [37]. It allows for running performance tests and generating charts for developers to analyze. There are even a number of language-dependent solutions for running different types of benchmarks and performance profiles. PerCI [36] is one such tool that has been proposed for performance analysis of Python projects in CI. There is also some more research for microbenchmarking of Java and Go-based applications [41]. They focus on microbenchmarking where they attempt to obtain performance results in as short a time as possible. Due to the complex nature of HPC applications and the amount of resources that may be required, it seems unlikely that microbenchmarking can be utilized; however this idea may still be useful for certain types of HPC applications. As for language-dependent profiling, in most cases HPC applications are multilingual, or are built with a number of different libraries that can play a major role in performance. So, while language-dependent solutions might be useful for components of some projects, they are not applicable to larger projects.

3.3 Container Tools

Docker [2] is one of the most-well known container tools and is used for many cloud and large-scale applications. While it is ubiquitous in most non-HPC environments, it has a number of design flaws that make it difficult to use for HPC programs; for one, it requires a root-privileged daemon to start and monitor containers. In order to start containers, users either need to be root, or must be added to a special group. Given that HPC systems can have hundreds and even thousands of users that need to access the systems this presents a huge amount of risk. In response to this, different institutions have created HPC-specific container management tools. The main two that

I note here are Charliecloud [55] and Singularity [42]. Charliecloud, designed by a team at Los Alamos, is meant to be run completely at the user-level. It includes a number of tools written in C, Python and shell scripts, including a container build tool that is able to build containers as an unprivileged user. Singularity, on the other hand, is a larger go-based application that is also designed for running HPC containers. Singularity includes support for a special SIF container file format that is designed to be portable from system to system.

Podman [52] is another container runtime system that was not originally designed for HPC, but still has some features that could make it useful for HPC. Podman is OCI compliant and thus follows existing container standards. Podman, while typically run as a root user, also offers a rootless mode which could be useful for running user-level programs in an HPC environment.

Since the BeeSwarm tool is designed to measure performance with applications in containers, I think it is important to note that containers have been shown to have little to no overhead versus running applications on bare-metal. Torrez et al. [61], for example, run three different benchmarks using Singularity and Charliecloud and find little to no overhead for these tools, except for a modest increase in memory on some tests.

3.4 HPC in the Cloud

Cloud providers typically provide support for setting up virtual machines in the cloud. Some of these companies offer more HPC-specific resource types as well as networking solutions that can give performance akin to that of an HPC system.

Google Compute Engine (GCE) [28] offers different VM types and classes. They classify their VMs into machine families, starting at lower-end machines that are designed for hosting small websites or databases, all the way up to machines that are optimized for memory, compute and for accelerators. Their different options offer a huge amount of possibilities for running custom cloud-based clusters. AWS [6], like google, also offers a large number of cloud services that can be useful for HPC, as well as AWS Batch. AWS Batch [5] is a service provided by Amazon for running various types of batch jobs. The system offers a Platform-as-a-Service (PaaS) option for developers who need to run batch jobs but don't want to set up and manage the virtual machines and operating system-level configuration details. AWS Batch manages all compute nodes for executing jobs without having to worry users about those details.

There are also open source solutions for providing and supporting Cloud infrastructure. OpenStack [26] is one such option that has been widely used for different public and private cloud systems. Being open-source it provides a Python-based API and is supported by BEE for launching and configuring clouds.

Another key part of cloud computing, especially with complex configurations, is the use of software to manage and deploy VM instances. Terraform [29] is one of the key players in this market. It's designed as a tool to manage cloud instances using the HashiCorp Configuration Language (HCL). Different providers and modules allow for extensions to the base configuration. Being able to store instance configuration as text, allows for a greater deal of control over cloud configuration and makes it easier to manage future changes. BEE offers similar functionality, using a templating language and configuration that is specific to each provider.

3.5 Resource Managers

Resource Managers represent a step below the workflow orchestration tools in terms of abstraction. They are used for management of jobs and node within a single cluster. Some of these tools have support for handling job dependencies, but they are not easy to work with for running workflows. Some of the key players are Slurm [56], IBM LSF [31] and HTCondor [30].

Slurm [56], or the Simple Linux Utility for Resource Management, is by now one of the standard cluster-management tools for HPC. Slurm is designed to schedule jobs using a special Backfilling algorithm. It even has support for dependencies between jobs, but does not handle more complex workflows. Slurm supports managing MPI jobs as well as non-parallel jobs.

IBM Spectrum LSF [31] is another resource manager designed to manage HPC systems. IBM explains that it designed for both high-performance and high-throughput computing. They also purport that it has features for GPU usage, containerization and big data analytics. LSF is used for many of the HPC systems that are based on IBM's power architecture [32], including some of those at Oak Ridge National Laboratory, such as the Summit supercomputer [66].

HTCondor [30] is a cluster management tool for High-Throughput Computing which can connect together distributed systems for job launching. It offers support for restarting failed applications and also includes mechanisms for file transfer and file management. HTCondor is also designed such that users do not have to have accounts on the compute nodes where applications will be run. Resource management and allocation to jobs are done through an exchange and comparison of ”Ads” that list the available resources on the compute node’s side, and the required resources for a job on the submission side.

Borg [67] is Google’s internal resource manager. Borg jobs are split between daemon-like jobs, that may need to theoretically run forever, and short-running batch jobs. Job submissions are done with a special BCL file format and they also use containers to execute applications. The design of Borg was the main inspiration for Kubernetes [60] which has become the open-source version of Borg.

Borg and Kubernetes, while using similar techniques for scheduling and job management, are not designed for HPC resource management. For one thing, they both were not designed for running highly-parallel and tightly coupled jobs. There does not seem to be any notion of gang scheduling, which is necessary for scheduling of MPI jobs on HPC systems. Nonetheless, the existing research and work done on both systems includes many good solutions and well-tested techniques for scheduling and management of clusters as a whole.

3.6 Scheduling

Benoit et al. [9] gives an extensive survey of workflow scheduling algorithms, theories and models. They note that the majority of existing workflow algorithms attempt to minimize the *makespan* or the total execution time of a workflow and note the representation of workflows as Directed-Acyclic Graphs (DAGs). They outline a number of different types of sub problems within workflow scheduling. These subproblems include the issues of scheduling chain-of-task workflows, structured application graphs, and various models of task replication. They also go over some more general methods for optimizing scheduling latency and other factors.

Fard et al. [22] present a new method for scheduling microservices using the Knapsack problem as a basis. Their system has a profit function which measures memory and CPU usage. They name their algorithm Least Waste, Fast First (LWFF) which is based upon the classic FCFS algorithm. While their solution was designed for private and public clouds, the resulting algorithm could have possible applications to HPC and HTC scheduling under certain workflow scenarios.

Duplyakin et al. [20] offer an interesting method for integrating multiple computing resources, while still keeping control of individual partitions of resources under the management of different systems. They focus on how to efficiently migrate resources, rather than jobs or tasks, between these systems when necessary. While their work is not completely related to scheduling of workflows, it can be extremely useful for the coordination of multiple systems, each of which having their own internal scheduling framework, which is very similar to what BEE attempts to do.

There are also some interesting related works from the field of grid computing. Ye et al. [72] present a method for scheduling workflows with multiple scheduling objectives. They present a number of evolutionary and population-based algorithms that attempt to find possible schedules that dominate other possible schedules. Their specific configuration, however, focuses on workflow planning, where scheduling of a workflow is completely done before the workflow even starts executing. While for small workflows this may be feasible, for those that are much more complex it can be very difficult to schedule workflows that may run for long periods of time, during which resources can go down and resource properties can change. Thus, full workflow scheduling is not really a useful algorithm for scheduling workflows.

Scheduling with Kubernetes [60] has also gathered a ton of research. Kubernetes uses a two-step scheduling process for scheduling Pods, or processes made up of a number of containers: first filter the list of available resources to determine what can be used to run a Pod, and second score the resources based on certain properties. Kubernetes then picks the resource with the highest score, schedules it and moves on. Stratus [13] is a cloud scheduling algorithm that utilizes bin packing with runtime estimation of tasks for minimizing execution cost. Stratus is separated into a packing component and a scaling component: the first deals with how jobs are allocated and the second deals with how many cloud resources are needed. Their design is meant to work with a system like Kuberenetes which would run in a private or public cloud scenario. However, their scheduling ideas have possible uses with workflow orchestration systems such as BEE. They could also be applied to help with autoscaling in some of these situations. Other research into Kubernetes-based scheduling includes adding I/O and CPU usage information to the scheduling process [43]. They assume that there is some way to know about current resource usage and then use this information along with a Balance Disk IO priority (BDI) algorithm to do the actual scheduling.

CHAPTER 4

Design of BeeSwarm and BEE Extensions

In this chapter I will go over the design of the extensions that I've added to BEE, as well as the design of BeeSwarm which uses BEE internally.

4.1 BeeSwarm

BeeSwarm is a set of wrapping shell and Python scripts meant to extend BEE for running in a CI environment. BeeSwarm installs all dependencies of BEE, initializes the environment and then executes BEE with a specified set of workflows using BEE's REST interface for communication.

4.1.1 Components

BeeSwarm adds a number of additional components on top of BEE to allow for execution of workflows on a CI virtual machine. These are composed of a number of configuration shell scripts and a main Python script that interfaces directly with BEE. The configuration scripts are designed to install all dependencies in the environment, including Charliecloud, the proper Python interpreter and package manager (Poetry [53]). Finally BEE's Python dependencies are installed, allowing the BEE code to run.

The CI script starts by running the `beeswarm/start.sh` script from the root of the repository. This script first installs Charliecloud and then pulls down a container containing most of the BEE dependencies [2]. This container contains a version of Python compatible with BEE (greater than or equal to 3.8) and includes a number of initial Python dependencies including Jinja2, YAML and others that are all required initially by the BeeSwarm Python scripts.

Once the container is set up, a special `beeswarm/beeswarm.sh` script is started from within the container environment. This in turn calls a number of environment and other initialization scripts that configure required environment variables and other required tools. Upgrades are done to some of the existing tools to ensure that no installation errors occur and special credentials are pulled

[2]https://hub.docker.com/r/jtronge/bee

from secrets that are stored in the environment of the CI system.

The `beeswarm.py` script itself is designed as a command-line tool with multiple subcommands that handle extra functionality. For instance, the beeswarm script has a `cfg` option which allows scripts and user tools to pull configuration variables out of the beeswarm.yml file. This is useful for checking environment settings, generating set up scripts and also to gather metadata from the workflow runs. The example workflows tested with BeeSwarm all used this feature to capture workflow output profiles by committing them to a special results branch of the BeeSwarm repository [3]. The beeswarm config (or `cfg`) option was used to pull the email and name configuration for making commits with git.

Once all dependencies have been installed, the environment properly configured, BeeSwarm then launches the desired cloud configuration with a call to the `beeflow-cloud` command which is a part of the base BEE package. This is based on the cloud configuration which is stored in the `beeswarm.yml` configuration file, which can be updated and modified for each test that needs to be run. This set up step can take around 10-20 minutes depending on the cloud providers response time and their internal network bandwidth. This step involves launching all VMs, installing software, including Slurm, Charliecloud, NFS servers, and creating users that will be used for running jobs. BEE is also installed on the cloud system and a single Task Manager is launched at the end of set up. This Task Manager will listen to HTTP REST requests coming from the workflow manager which runs on the CI system.

Communication between the cloud cluster and the CI system is done through an SSH tunnel which allows for secure HTTP communication. In order to facilitate the connection set up I added a connect option to the `beeflow-cloud` script which handles this configuration and waits until the Task manager comes up and responds to requests, or dies after a certain max number of retries if something went wrong with the cloud configuration.

Once the cloud set up is complete, the beeswarm script is invoked again, this time with another subcommand, the `scale-tests` subcommand, which initiates the scale-tests based on the beeswarm.yml configuration. This step encapsulates a number of substeps, including that of starting the BEE components that will need to run on the CI system, such as the BEE Workflow Manager and the scheduling component. The script waits for these to start up and then will enter

[3]https://www.github.com/jtronge/BeeSwarm/

a loop for submitting and starting the scaling/performance tests.

The scaling/performance tests are based on a number of options within the beeswarm.yml configuration file. A templated workflow file will be generated and saved to disk. The BeeSwarm client then uses the REST API to submit the workflow to BEE along with a second request to start the workflow. Then BeeSwarm will wait for BEE to complete the run of the scalability test. To ensure stability of results, the configuration also includes a `count` value that will cause the workflow to be executed multiple times. During execution BEE generates a workflow profile in a JSON-format that can be used for later analysis and generation of graphs. This includes timing information and state changes of each task in a submitted workflow.

Finally, once all tests have been run to completion, a simple git command is used to commit all generated results to a special branch, and these results are then pushed up to the remote repository. To demonstrate the BeeSwarm code, I've used a simple `results` branch as to differentiate from the main branch that is used for testing. In some cases developers may wish to add more CI actions that will automatically analyze produced results. In this way developers can easily open up a repository front-end and view current result graphs and quickly compare the current version of the code with the previous

4.1.2 Management and Communication

Communication between the existing BEE components is defined through a REST interface. This makes it easy to add new components for new features, since all that is required is that they are able to interface with the existing REST interface code. For BeeSwarm, I've design a simple Python script that is able to communicate the BEE Workflow Manager through the existing client REST API. BEE includes an existing client program that already works with this API, however this script is not suitable for CI environments since it is designed for user-interaction, rather than operation in a headless-environment as with CI testing. BeeSwarm includes all of the same options as the existing client script and also includes code to monitor existing workflows without user interaction. This makes it well-suited to testing purposes, where some tests may take a long time to complete.

4.1.3 Configuration

BeeSwarm's configuration is based on a YAML-file. This file includes a whole range of options, specifying everything from the repository that BeeSwarm is installed in to the configuration of the

```yaml
scale_tests:
  - name: 'nwchem-pspw-f29685d'
    wfl_dir: './workflows/nwchem-mpi'
    # Workflow params
    params:
      container: '/home/bee/nwchem_f29685d.tar.gz'
      ntasks_per_node: 8
      nodes: 1
    template_files: ['nwchem-beeswarm.cwl']
    main_cwl: 'nwchem-beeswarm.cwl'
    inputs:
      nw_file: "/nwchem/QA/tests/pspw_scan_h2o/pspw_scan_h2o.nw"
    count: 2
```

Figure 2: Section of an example beeswarm.yml config file. This particular section shows the `scale_tests` configuration section that tells BeeSwarm the tests to run, containers to use, number of tasks, etc.

cloud that will be used for testing. Each test to be run corresponds to an entry in a test list. Each test entry contains options related to a specific workflow file that is to be run. There are also supporting options, such as the container, the number of nodes required and metadata such as the hash of the commit used to build the code that is being tested. Container information is also listed in order to allow containers to be built on the fly, since new code changes may have been made that need to be properly tested by BeeSwarm. An excerpt of an example configuration file is shown in Figure 2.

4.2 Cloud Launching

For my work with BEE, I needed to implement a number of extensions, including code for launching jobs in the cloud. For running jobs, there are a number of different mechanisms for managing cloud resources. Some providers, such as AWS [6], offer batch job submission as a service. For BEE, our cloud launching systems is based on setting up a configurable template-based cluster. This allows one to easily install and configure an HPC-cluster with the exact dependencies and resource management code needed.

For configuration, Jinja [48] templating is used, which has been used in other cloud management systems as well, such as with Google Compute Engine [28]. This allows for injection of a number of test-specific parameters that can make it easy to launch varying amounts of nodes. For instance, one test of an application may require eight compute optimized cloud nodes, while another may require only a single GPU-optimized node for a GPU application test. See Figure 3 for an excerpt of an example configuration.

Due to the different options and features that each provider has, BEE's cloud configuration mechanism is designed to take input files that are different for each provider. This, of course, can cause problems with porting one cloud configuration from one provider to another. However trying to make a single configuration work with multiple providers is more complex than one might initially think. For instance, each provider typically has their own list of instance names and each instance may have a certain amount of RAM, number of cores, and options for accelerators. Even among providers that use the same API, they may have different resource types that they provide. This can make configuration extremely difficult if some parts of the set up rely on certain node features or specific quantities that are related to the instance configuration. Thus, BEE doesn't attempt to abstract these problems away, but allows the user to define their configuration for each platform. This does require different configurations for each provider, but ensures that configurations will work well with each platform.

I should note here that some of the other solutions do allow for instance configurations to work with multiple providers. One of these is Terraform [29], which supports using multiple providers in their own special configuration language. Different providers from different regions can be used for similar configuration files. It attempts to abstract the providers and the configuration, but because of differences between providers there are still some issues with the configuration process when needing to launch with different providers.

4.3 Multiple Task Managers

Having the ability to run BEE with multiple Task Managers is an important design decision. Previous versions of BEE were designed to work with a single task manager that could submit tasks to an underlying HPC resource manager. The task manager would then monitor tasks and

```
{% for node_name in compute_nodes %}
  - name: {{ node_name }}
    machineType: {{ compute_nodes[node_name]['machine_str'] }}

    disks:
      # Set the boot disk
      - boot: True
        autoDelete: True
        initializeParams:
          # Set the source image and disk size
          sourceImage: {{ compute_nodes[node_name]['src_image'] }}
          diskSizeGb: {{ compute_nodes[node_name]['disk_size_gb'] }}

    networkInterfaces:
      - network: 'global/networks/default'
        # External IP address
        accessConfigs:
         - type: 'ONE_TO_ONE_NAT'
           name: 'External NAT'
    metadata:
      items:
        # Start up script for the node
        - key: 'startup-script'
          value: |
{{ startup_script(wireguard_conf_compute(compute_nodes[node_name]['vpn_key'],
                                          compute_nodes[node_name]['psk']),
                  compute_nodes[node_name]['ip'], False, slurmconf())|indent(12) }}
{% endfor %}
```

Figure 3: Example subset of a Google Compute Engine (GCE) configuration for BEE. This produces a YAML configuration file that will be input to the GCE. GCE also offers similar options for generating templated files like this.

then report any changes back to the Workflow Manager. Using a single task manager works for general workflows, but can be limiting since a task manager is limited to only a single system. What if a user has access to multiple HPC systems, or multiple cloud systems for that matter? It is possible that a workflow may be able to span across these multiple systems. These are issues that have only been studied in a limited number of recent works, including StreamFlow [14].

Therefore, to allow for multiple systems to be utilized for workflow execution, multiple task manager can be used. I created a prototype implementation of this, which works by launching a task manager on each available system and connecting the workflow manager to each task manager that is up and running. Then, with the scheduler component, the Workflow Manager can choose which Task Manager to send tasks to as a workflow executes. The Task Manager can then launch and monitor tasks as before.

This design does have its issues that need to be addressed. For instance, how do we deal with data being stored on one system, and not on another? One way to deal with this is to ensure that the scheduler understands where data is and what tasks depend on that data. Then there is also the question of how data should be transferred, or whether it should be transferred, from one system to another. This may be desired in some cases because of the particular computational requirements of a task versus the network latency required to transfer the data. I talk about how data transfer is done with this prototype implementation in Section 4.5.

4.4 Containers and MPI

BEE is designed to work with both Singularity [42] and Charliecloud [55]. BEE's configuration contains a section for container configuration, allowing the user to choose which container run time system will run within the BEE Task Manager component. Different configuration options, such as the parameters to be passed to each command line tool, container storage and build directories are also contained within the configuration file. BEE includes an abstract container runtime interface which modularizes the code and ensures that new container runtimes can be added in the future. BEE includes support for the generic `DockerRequirement` hint which is used for specifying container information. Note that even though the requirement begins with "Docker", no support for Docker is built into BEE due to its inherent security issues for HPC. Rather the name Docker is used since it is one of the most common container tools and is used for other non-HPC workflow orchestration systems.

In order to support running MPI applications with BEE, I added support for an extension requirement, `beeflow:MPIRequirement`, that allows specification of the MPI version, amount of nodes and other MPI runner information. BEE uses this information to pass specific command line arguments to the underlying batch scheduler on the system where an MPI task is to run. For example, on a Slurm-based HPC system, BEE will pass extra arguments to the `srun` binary to ensure that the task runs with the proper MPI environment set up on the cluster. An example of this requirement for a task is shown in Figure 4. The CWL community has also produced another MPI requirement [45], however its design is not suitable for our needs because of the extra parameters and configuration required for HPC.

4.5 Data Transfer

Data transfer is another major issue when working with Workflow Orchestration software. BEE was originally designed to completely rely on a shared file system, which is a part of most existing HPC systems. This works well within a single managed cluster, but no longer works when one wants to run workflows across systems. In order to handle multi-system execution, some method of data management, other than shared file systems is needed to push and pull data between systems if necessary.

For a number of experiments with BEE, I've implemented a simple HTTP-based file transfer method. Basically this adds another endpoint within the Workflow Manager component which the Task Manager's can pull data from if a given task requires some sort of input data. This design is a simple placeholder, however, as workflows with higher data requirements will easily overwhelm the workflow manager. I will discuss possible future work in this area in a later chapter.

4.6 Scheduler Design

To start out with, I will define a couple of the terms used here more clearly. Other research may use different terms or may even use some of these same terms for completely different concepts. A single scheduling resource, as I define it here, is a given platform or system, such as a supercomputer or a cloud system. I try to highlight this here, because in many other works a resource is defined at a finer level, into nodes or partitions of homogeneous resources. BEE is designed, instead, to schedule jobs at a much higher level, choosing between whole systems, rather than single nodes or

```
class: Workflow
cwlVersion: v1.0
inputs: ...omitted...
outputs: ...omitted...
steps:
  lulesh:
    run:
      class: CommandLineTool
      baseCommand: [/lulesh2.0]
      stdout: lulesh_stdout.txt
      inputs:
        size: ...omitted...
        iterations: ...omitted...
      outputs:
        lulesh_stdout:
          type: stdout
    in:
      size: size
      iterations: iterations
    out: [lulesh_stdout]
    hints:
      DockerRequirement:
        dockerPull: "...registry location of container..."
      beeflow:MPIRequirement:
        ntasks: 27
```

Figure 4: Example BEE workflow which includes several hints (which we also refer to as requirements here) including `DockerRequirement` specifying the container and `beeflow:MPIRequirement` which specifies MPI-specific runtime information.

partitions of a scheduler. Of course, this is not to say that users may only have access to certain parts of an underlying system, such as a specific partition dedicated to the research that they are doing. Here this simply is referring to what BEE sees as a resource. BEE is designed to make a scheduling decision and then pass job information onto a lower-level system- or batch-scheduler such as Slurm or LSF, which will then handle execution and further allocation at the node and partition level.

When putting together the design of the scheduler for BEE, I decided to make some simplifications of the process. Within BEE, workflows are stored as a DAG, similar to most other workflow orchestration systems. Tasks are executed as they become ready, which happens as other tasks complete and fulfill dependencies. Some works attempt to schedule entire workflow DAGs before the workflow ever starts, or may attempt to schedule large dependent partitions of the workflow after initial tasks run. The problem with this design is that tasks of a workflow can take an inordinate amount of time to complete. In most cases there is simply not enough information available to schedule tasks that will not run until far in the future. Depending on how long a workflow is expected to last, resources could go down and there may even be system and configuration updates that could break a schedule. There also may be special types of tasks that are designed to branch out into many more tasks based on data produced in a previous step. The scheduler has no way of knowing how many subtasks will be produced for such a task and therefore could not possibly produce a good schedule for that until each task that needs to execute is known and ready to run. Thus, taking all these issues into consideration I decided to design the scheduler to make decisions based on tasks in independent sets of ready tasks. In other words, these are all the sets of tasks that will become ready at the same time and will thus be able to run concurrently. Scheduling in this way is designed to be able to make use of accurate and current resource information, environment conditions, as well as data dependencies and costs for a given set of tasks. When compared with full-workflow scheduling, this method is designed to make more accurate decisions under real world conditions. Liu et al. [44] categorizes this type of scheduling as bag-of-task scheduling in workflows.

The scheduling process that I work on here produces a simple map from task ID to resource ID. This does not include timing decisions or other submission details. Rather this type of information is left up to the underlying system scheduler to handle. I believe this makes scheduling within BEE more flexible since it ensures that it doesn't have to deal with exact timing issues and submission information that are specific to underlying schedulers.

For the algorithms that I've put together, I've designed them to follow a similar technique to that used by Kuberenetes and Google's Borg [67, 60]. There are two steps, one for filtering the available resources and another for scoring/picking from among those resources. The implementation of the scheduler allows for use of any class that includes a `schedule()` function, thus allowing for other scheduling algorithms to easily be added.

I implement two algorithms in this work: **First-Choice FCFS** and **Sampling**. While I show in the next chapter that these produce good results under simulation, there are many other algorithms out there that may produce more optimal results, especially for highly specialized workflows that need to take into account specific considerations.

First-Choice FCFS is based on the classic First-Come First-Serve algorithm, in the sense that when given a set of independent tasks to schedule, for each task it filters the resources based on task requirements and simply chooses the first resource that is available in the list. This is not designed to produce an optimal schedule, but rather to give a quick valid schedule. For smaller, simpler workflows this should work quite well. For larger and longer running workflows, however, this will likely give bad results.

The **N-Sampling** algorithm, is designed to give good results for larger, more complex workflows. This algorithm determines the valid resources for each task, as in the previous algorithm. Then the algorithm computes an MxK table where M is the number tasks, and K is the number of resources, filling each entry (i, j) with a score for running task i on resource j. This, of course, ignores those entries for which task i cannot run on resource j. The Sampling algorithm also requires an input count parameter N that determines the number of possible schedules to generate. The algorithm will generate N random samples, corresponding to valid schedules. It then chooses the schedule from the randomly generated mappings for which the pre-computed scores produce the best summed score.

The score produced for each task T and resource R is a tuple (α, β), where α gives the total estimated runtime, including transfer times and β gives the total estimated cost. These values are computed as follows:

$\alpha = \text{load(R)} + \text{transfer\_time(T, R)} + \text{runtime(T)}$

$\beta = \text{cost\_per\_core\_second(R)} * \text{runtime(T)}$

The transfer_time(T, R) function is based on the amount of estimated data that will need to be transferred between resources and the connection speed between the resources. To simplify this I've included a matrix as input to the scheduler where each entry (i, j) gives the estimated transfer speed between each resource i and resource j.

At the end of the next chapter I will demonstrate use of these two algorithms with a custom simulator.

CHAPTER 5

Results

In this section I will detail a number of experiments that I have done with the BeeSwarm code that wraps around BEE. I will also show a demonstration of running a workflow across systems, revealing some of the key scheduling issues. Finally I will show results of running some of the scheduling algorithms for BEE and how well they work under simulation.

5.1 BeeSwarm

For testing BeeSwarm, I ran experiments using three different HPC applications: CoMD [21], LULESH [38] and NWChem [7]. CoMD is a molecular dynamics code that can be used for running simulations on different types of materials. LULESH is a simplified hydrodynamics code that is designed to represent common scientific application design practices. NWChem is a powerful tool used for running computational chemistry experiments using many different types of models; it is also under active development with numerous commits being made each day. Each of these applications have unique characteristics that require slightly different scalability test cases. The results presented here are a good indicator of how BeeSwarm can work with different HPC applications which often have extremely diverse set up procedures, input files and runtime requirements. For the VM and instance types that I use for the experiments, please see Table 1 which lists the configuration details.

5.1.1 CoMD

For the first test, I've used the CoMD HPC application. CoMD is a molecular dynamics proxy application and is a mini-app, in the sense that it is designed to run with a fixed example problem, while still allowing for various parameters to test particular HPC platforms and to demonstrate tools such as BeeSwarm.

For the first test, I ran the CoMD MPI version on a single n1-standard-16 node for 1-16 MPI tasks. Figure 5 shows the scalability graph for running CoMD under this configuration. As you

Instance Name	**CPUs**	**Memory (GB)**
n1-standard-1	1	3.75
n1-standard-2	2	7.50
n1-standard-4	4	15
n1-standard-8	8	30
n1-standard-16	16	60
n1-standard-32	32	120

Table 1: List of Google Compute Engine instance types with the number of available CPUs and memory.

can see, BeeSwarm is able to test CoMD under a number of different configurations and the results show relatively good scalability.

For the second test of CoMD, I ran the application on 1-4 nodes to test multi-node scalability. Figure 6 shows the results of executing this test. As you can see, once again the results show relatively good scalablity and performance.

5.1.2 LULESH

For LULESH I ran two major tests. LULESH requires the number of MPI tasks to be the cube of an integer, thus it cannot use the typical power of two scaling for these tests.

The first test of LULESH used a container with MPI-only and OpenMP disabled. Here two n1-standard-32 nodes were provisioned with Google Compute Engine and tests were launched with 1, 8, 27 and then 64 MPI tasks. Figure 7 shows the results of running this configuration of LULESH. LULESH increases the problem size automatically with each task added and produces its own specialized performance value in zones per second. Thus for this example we show how this metric increases with the number of tasks.

The second test of LULESH was done with a different container where LULESH was compiled with OpenMP support. Then I designed test cases to run LULESH on multiple cloud instances with increasing numbers of cores. This result is shown in Figure 8 and gives a pretty good scaling result for the OpenMP version of LULESH.

Both CoMD and LULESH result in relatively good performance, since they are both well-tested programs that have been refined for the purposes of HPC scalabilty testing and other research. Their

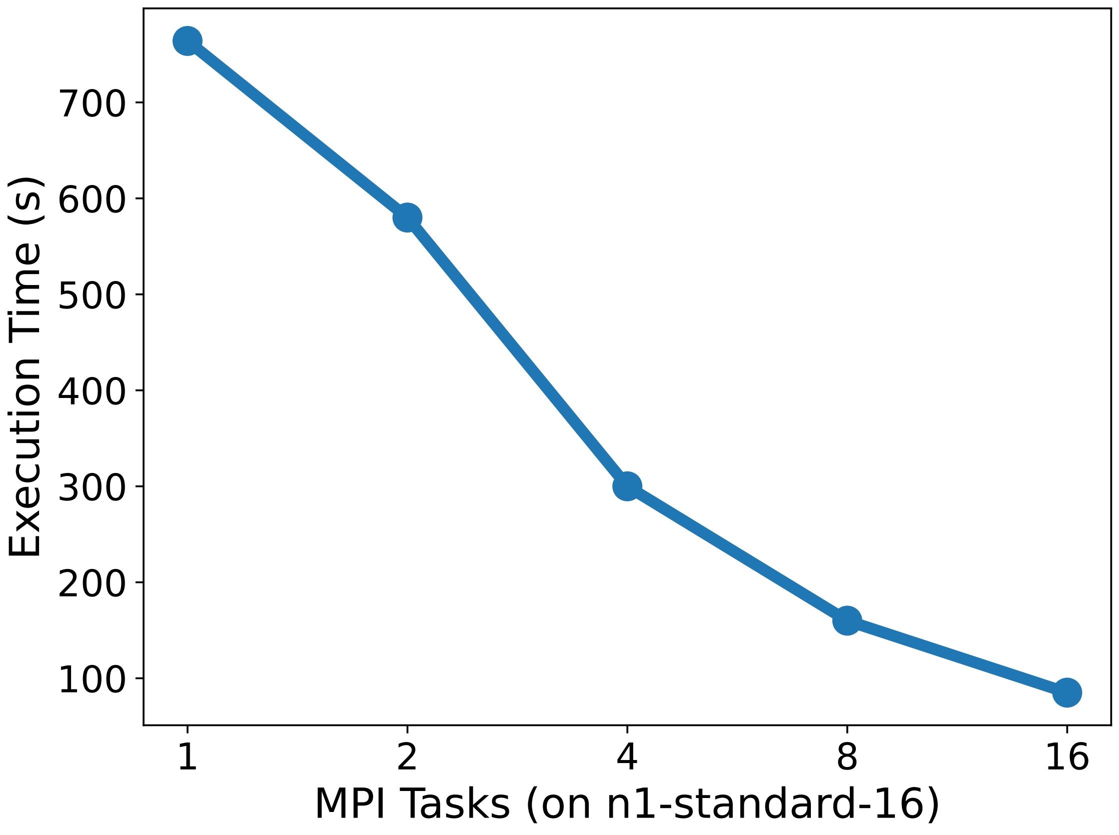

Figure 5: Strong scaling test of CoMD on an n1-standard-16 node with 1 through 16 MPI tasks.

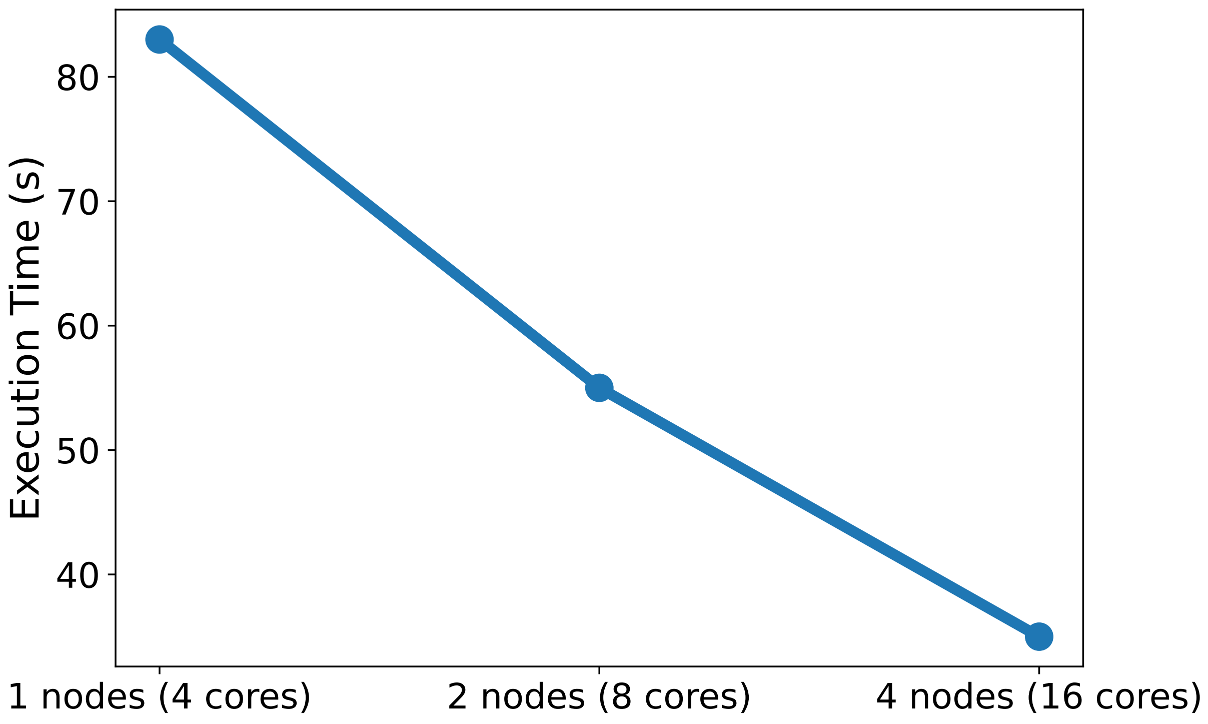

Figure 6: Multinode scaling test of the CoMD application. This test made use of four n1-standard-4 VM instances.

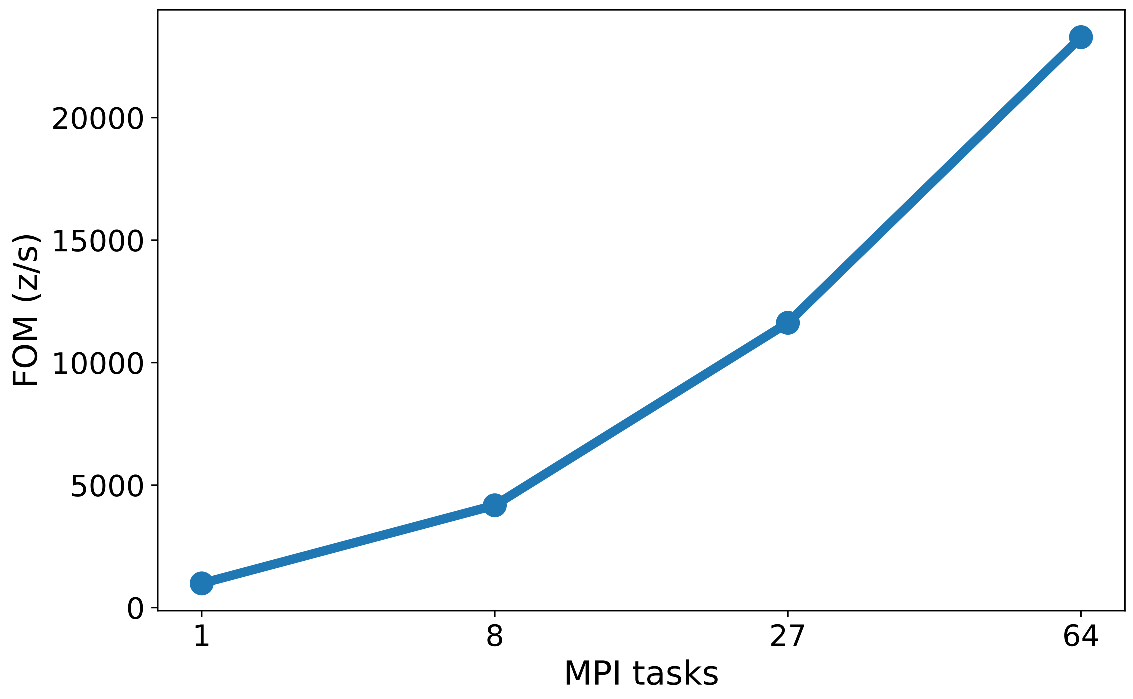

Figure 7: Scaling test of LULESH on two n1-standard-32 nodes with 1, 8, 27 and 64 MPI tasks. LULESH is designed to run with MPI task counts that are the cube of an integer, thus a typical power of two scaling test cannot be done. The y-value given here is in zones per second, which is output by the application as a performance measurement.

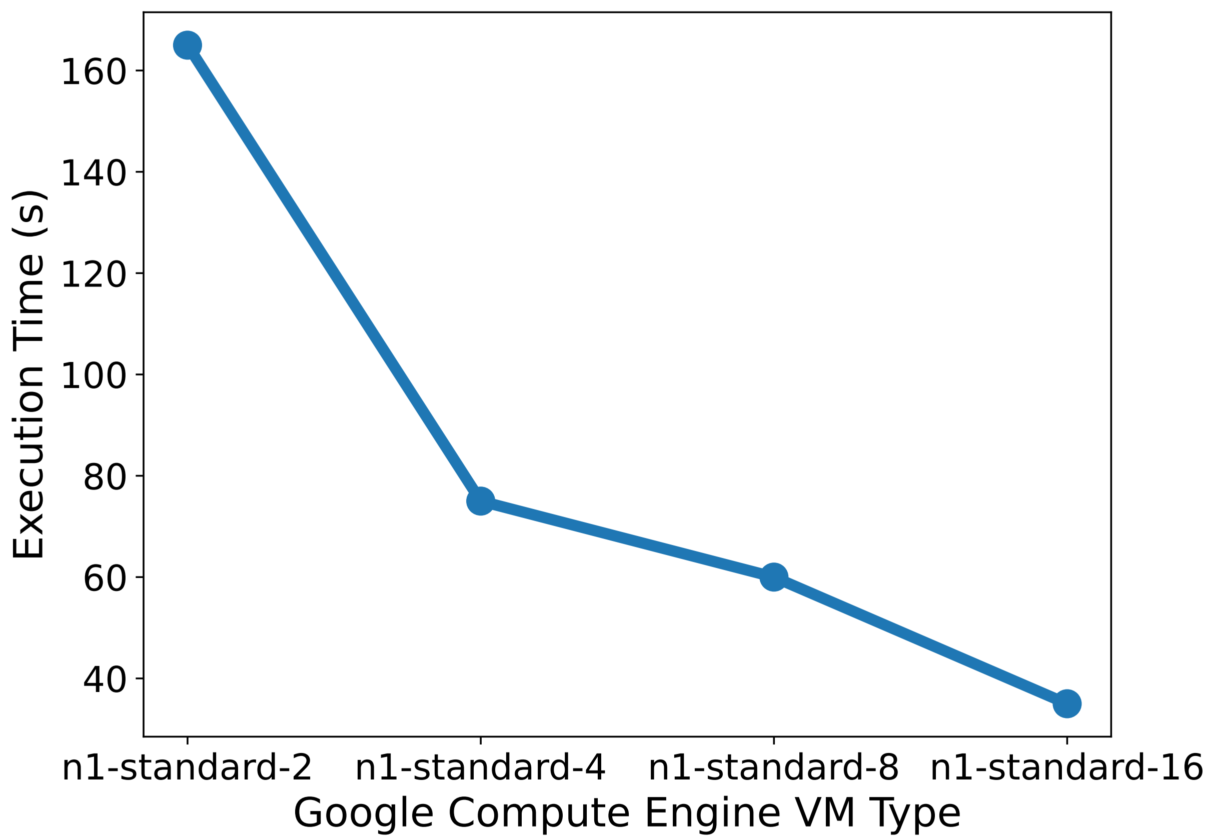

Figure 8: A scaling test of LULESH compiled with OpenMP support. LULESH is run on Google Compute Engine VMs with increasing core counts.

main goal is to be well-suited to running experiments such as these, as to demonstrate how HPC applications should work under optimal conditions. Real-world HPC applications will never be as easy to test, however, given that they may have much larger codebases and more areas that can affect the performance. So, to highlight this, next I'll take a look at a widely used HPC application: NWChem.

5.1.3 NWChem

Using NWChem with BeeSwarm, I executed tests using a number of different commits from the recent commit history. After building a container for each commit I then used BeeSwarm to run scaling tests for each version of the code on different input data sets. These results highlight where BeeSwarm can be extremely useful for projects with long commit histories and a large number of developers working on different components. Tests such as these can help pinpoint problem areas in terms of performance and help developers to determine how to solve these performance issues that could result from a number of different modules within a project.

The four commits that I used, as made by different NWChem developers, are listed below:

- f29685d (Nov 29 2021): *"change for singularity [ci skip]"*
- 05aafc8 (Dec 9 2021): *"Update build_simint.sh"*
- adab52a (Jan 3 2022): *"removed mention of preload script [ci skip]"*
- 519b710 (Jan 13 2022): *"default memory increased to 1G"*

Note that between each commit above there were a number of other commits, contributing to a larger total number of changes.

In Figure 9, a single-node scaling test was done with the commits of NWChem on an n1-standard-16 node. This used a file which makes use of the SCF module of NWChem [4]. This test has one interesting result in showing that commit `f29685d` seems to perform better than all of the other commit hashes. In a full-application test, this could be extremely useful in determining which commits contain the most performant code.

Pinpointing the exact cause of the performance in this case is definitely the job of an active developer on the project, but for now I attempt to note some of the possible causes. Comparing

[4] nwchem/QA/test/scf_feco5/scf_feco5.nw

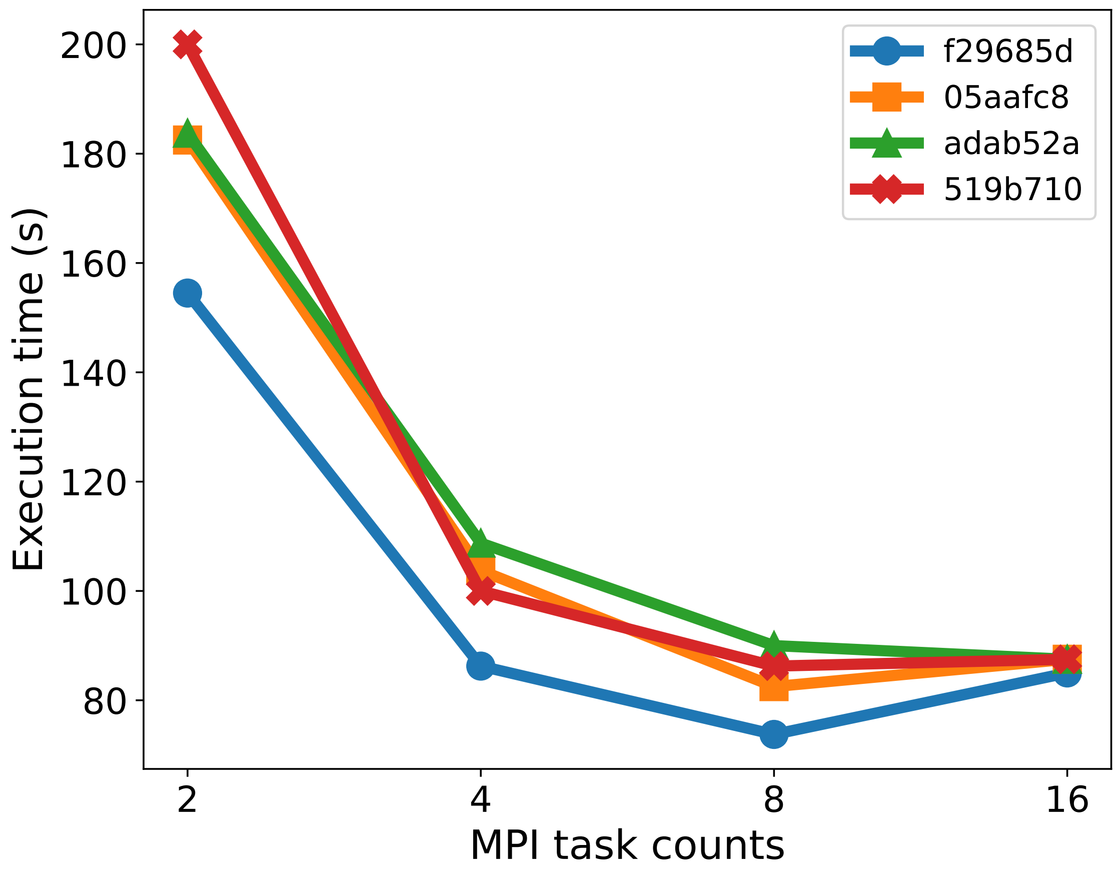

Figure 9: First multi-commit scalability test of NWChem using the `nwchem/QA/test/scf_feco5/scf_feco5.nw` input file.

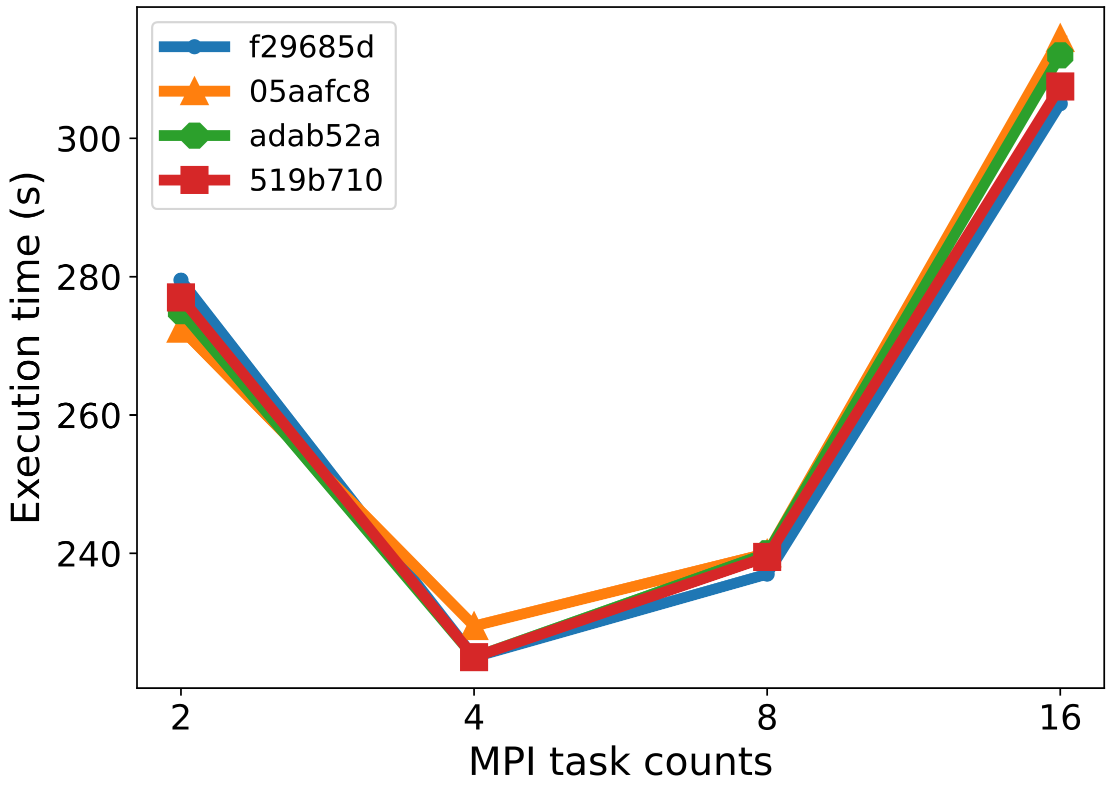

Figure 10: Second multi-commit scalability test of NWChem using the `nwchem/QA/tests/oniom2/oniom2.nw` input file. This test begins to show poor performance with 8 and 16 MPI tasks.

changes between the commits `f29685d` and `05aafc8` you can see changes in a number of files including `src/tce/tce_energy.F` and in `src/nwpw/band/lib/electron/c_electron.F`. These files include code that seems to be related to the SCF module, but the changes shown do not directly involve it. There is a possibility that a series of function calls could cause some of these changed lines to be run, causing the results of the performance analysis.

In Figure 10 I ran a test with a different data set [5], but with the same node configuration. Good scaling is shown for 2 to 4 MPI tasks, however the performance degrades for 8 and 16 tasks on the same node. In these cases something is causing a performance hit for larger numbers of tasks on a single node. As for pinpointing the exact cause, there is a possibility that the particular configuration does not have enough memory; each MPI task may require a certain amount of

[5]nwchem/QA/tests/oniom2/oniom2.nw

memory, and as all cores are used, there is not enough memory to support each task. It could also be that this particular module is not designed for running on a single node.

These results for NWChem obtained with BeeSwarm show how CI scalability tools can be used to help find critical performance issues in HPC applications. As applications change over time and are updated, the correctness tests will continue to show passing results, but performance issues can be much more difficult to analyze without a proper tool. Being able to test HPC application performance with CI is invaluable for larger HPC projects.

5.2 Cross-System Execution

While BeeSwarm utilizes BEE for testing performance of individual HPC applications, larger HPC projects and workflows are typically made up of many more applications and dependencies. One of the key design decisions for BEE is the ability to schedule tasks of these types of workflows across multiple systems. This design allows for workflows to grow beyond the size of a single system and also allows for BEE to support workflows with components that require cloud resources or hardware that is specific to HPC. There are a number of useful consequences of this design. For instance, some workflows may be designed to produce data on an HPC system and then analyze it on a cloud system. Another possible use case is for when specific systems are overloaded and users still need to run workflows on a deadline. In some ways, this use of BEE can be thought of almost as autoscaling at the user-level.

5.2.1 Blast Workflow Example

For one demonstration of running a BEE workflow across multiple systems I decided to use a workflow based on the BLAST [4] bioinformatics application. This workflow includes a first task that builds a database file from an input FASTA sequence, which encodes a protein sequence in a simple text file. After this task runs and completes, there are 25 different tasks that depend on the created database. These tasks each take a different input file and are designed to search and compare the input file with the sequence in the database. These tasks are highly parallel and can all run at the same time. The data sets used were an open Goldfish genome sequence [12] and another set of fish gene sequences annotated by Fischer et al. [25].

In Figure 11 I show the execution of this workflow under three different resource configurations. Since in my case I do not have access to a wide range of cloud services and HPC platforms, I chose

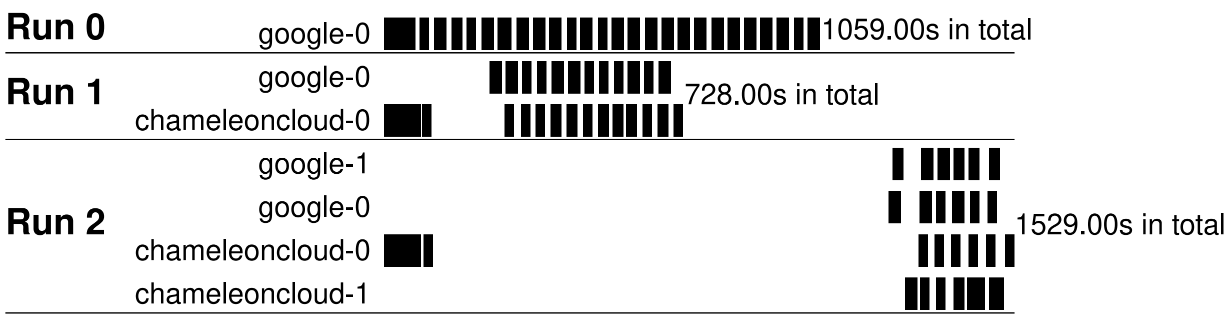

Figure 11: Example run of the BLAST workflow on three different resource configurations with BEE. The last run begins to degrade because of poor scheduling and the increased data transfer time.

to simulate having more resources than I actually do by partitioning the resources that I did have access to on Google Compute Engine (GCE) and ChameleonCloud into multiple resources. In this case each resources has a single node, but in reality, the typical resource would usually have many more available nodes and options. In the figure, I first run the BLAST workflow on a single GCE node, labelled *Run 0*, giving a makespan time of 1059 seconds. In the second run I executed the workflow with two resources, labelled *Run 1*, one on GCE and one with ChameleonCloud, giving a total decreased makespan time of 728 seconds. Finally in the last run, *Run 2*, I executed the workflow on two GCE resources and two Chameleoncloud resources, giving a makespan time of 1529 seconds in total. This last run doesn't give good results because of poor scheduling. Here the scheduler is designed to spread the resources out to each resource available, which decreases execution time in *Run 1*, but increases execution time in *Run 2*.

5.2.2 Scheduling

To run experiments with the scheduler, due to the lack of many existing open workflows, as well as the set up and configuration time required for more complex workflows, to properly evaluate the scheduler on a wide-range of data sets, I've decided to use a number of simulations using different workflow traces. Here the algorithms used are designed to take into account data transfer, as well as cost and total makespan time, some of which have been shown to be problematic in the last section.

I attempted first to utilize a number of workflows traces from the Workflow Trace Archive (WTA) [68], however many of the workflows did not contain exact node count or memory infor-

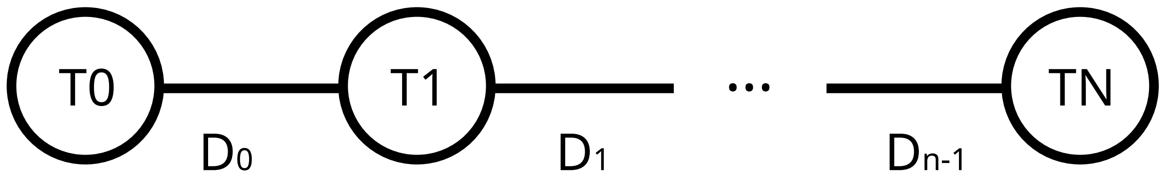

Figure 12: An example Linear Chain workflow. Task T0 produces some output data of size D_0 which is needed by task T1. One task executes at a time producing data for the next task to consume.

mation. Instead I found more useful job data in the Parallel Workloads Archive [23] which lacks workflow dependency information but includes extensive trace data about different types of jobs. Using this input data, I created synthetic workflows where each task of the workflow is a job from a trace, and dependencies are added randomly between tasks based on the type and shape of the workflow to run under simulation. I based the types of generated workflows on the types listed by Benoit et al. [9]. In particular I create synthetic workflows that are **linear chains**, or a sequence of dependent tasks where one task is able to run at a time, and **general** graphs with random amounts of tasks that can run at once. The exact traces used for the synthetic workflows are based on traces produced by the ForHLR II system in Germany [58], which include more recent data with larger node requirements common to modern workflows.

For the simulation data, I assign data amounts to the dependencies between tasks based on the total runtime of each task. I assume here that each task can produce a limited amount of data that can be used as input to a dependent task. Dependencies are then added between tasks randomly. For the **linear chain** workflows, each task will have a data dependency with the task immediately preceding it in execution. An example of the linear chain workflow is shown in Figure 12.

For the simulator itself, I created a simple simulator in Python [6]. The simulation is based on a number of different objects including a `Resource` object and a `Workflow` object. `Resource` objects each have their own properties, such as the amount of processors available and the estimated compute ability, etc.. I also include a simple `Workflow` object which is constructed with all of the dependencies between tasks as well as information about the individual tasks. Workflows are separated into sets of independent tasks which can all run at the same time. To start a simulation

[6]https://github.com/jtronge/bee-scheduler

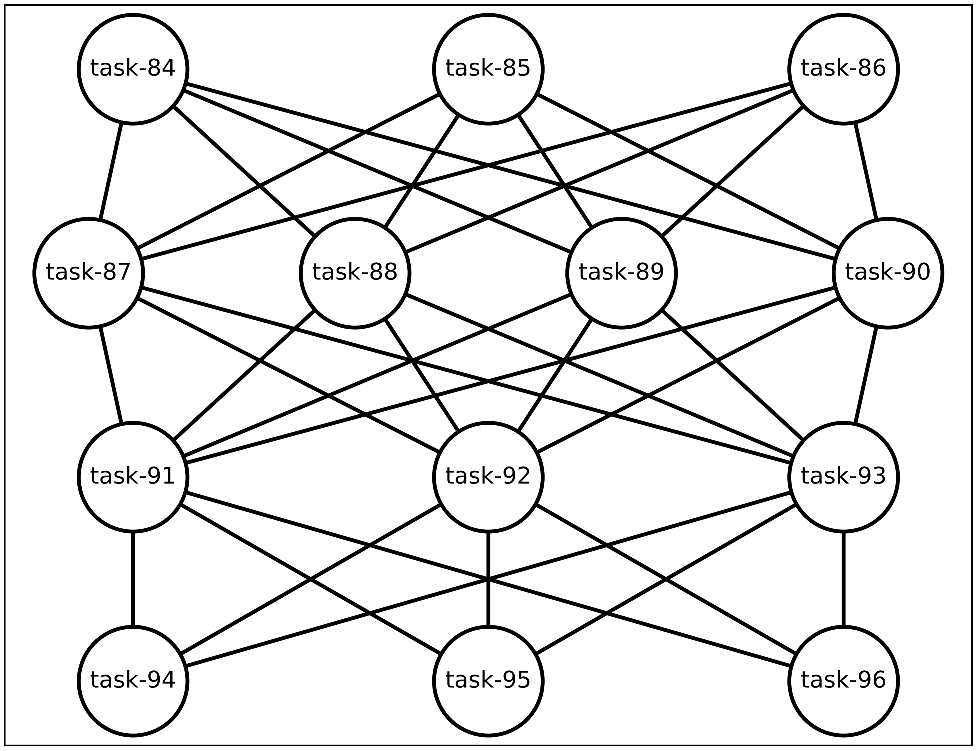

Figure 13: Display of one of the general workflows with synthetic dependencies added.

for a workflow and scheduling algorithm, the first set of independent tasks are queried from a workflow. The scheduling component is then invoked with the task metadata, resource metadata, and a communication matrix specifying data transfer rates between each resource, which then produces a schedule for each task. These tasks are then allocated to each available resource, which is configurable within the simulation. For each resource object, the resource is allowed to "execute" each individual task. In reality the resource object performs a calculation based on the resource computing speed, available processors and the properties of the individual tasks allocated. Once all "execution" has completed, the loop will start again with the next independent set of tasks and continue until all tasks have run and completed. The total execution times, scheduling results and other metadata are all returned back to the caller of the simulation for analysis. Pseudo code for this simulation algorithm is shown in Figure 14.

For the simulations that I ran I used four different resources. The key properties of each resource are its speed (`speed`), total number of cores or processors (`cores`), the load factor (`load`) and the cost per core second (`cost_per_core_second`). Each of these cause different changes in the simulated execution of tasks. As the `load` property increases a random minimum wait time value will increase, simulating the load of an HPC system. The `core` value helps determine how many tasks can run at once on a resource, given that each task also has a required number of cores. The `speed` property is a real value that estimates how fast a given system is. The total execution time is based on a very simple calculation where `speed` is multiplied by the task's runtime. This will also factor in a wait time that is given by the `load` value. Table 2 shows the exact resource parameters that I used. Another important part of scheduling is the communication matrix M, where entry $M_{i,j}$ gives the communication or data transfer rate between resources i and j. When tasks have a dependency with another task that has run on another system, the simulation adds in a transfer time based on this matrix. For the simulation I generate this matrix by giving a 0 value for each entry (i, j) where $i = j$, which holds the communication latency for a resource and itself, while giving a random value from 1 to 10 for every entry where $i \neq j$.

For the evaluation of the scheduling algorithms, the key factors that I look at are total makespan time, or the time from the start of the first task to the finish time of the last task of a workflow, as well as the total cost of running the workflow. Here, of course, the goal is to minimize both cost and makespan time in order to get the best scheduling results. I use the same resource configuration

```
def simulation(workflow, resources, comm_matrix, scheduler):
    full_schedule = initial empty schedule
    profile = profile of task runs
    while there are still more tasks to run:
        tasks = get ready tasks from the workflow
        allocations = scheduler.schedule(
            tasks,
            resource_metadata,
            comm_matrix,
            full_schedule,
        )
        for resource in resources:
            scheduled_tasks = get scheduled tasks for this resource
            result = resource.execute(schedule_tasks, comm_matrix)
            add result to profile
        update full_schedule with scheduling results
    return profile
```

Figure 14: Pseudo-Python code for performing a scheduling simulation.

resource ID	**cores**	**load**	**speed**	**cost_per_core_second**
res-0	1024	10	1	0.00003
res-1	65536	30	2	0.0001
res-2	2048	20	1.2	0.00008
res-3	512	10	1.1	0.0001

Table 2: Configured resources and their properties for the scheduler simulations.

as listed above. Figure 15 shows the results for running 256 linear-chain workflows with synthetic dependencies and Figure 16 shows the same experiment for 256 general workflows with synthetic dependencies added. The graphs use a statistical box-plot to help show the distribution of workflow makespan and cost values for the 256 different workflows run with the different algorithms.

The First Choice FCFS algorithm is the simple FCFS algorithm explained in the previous chapter. The algorithms labelled "Sampler n" give the sampling algorithm run with n samples; I experimented with four different versions of this algorithm with increasing samples of 2, 8, 16 and 32. Finally an absolute optimal value is calculated and added beside the algorithms. The optimal values for both cost and makespan are calculated assuming complete optimal conditions: data transfer time is assumed to be 0, as if the workflow data was automatically managed by a shared filesystem, and the speed factor is assumed to be 2.0, which is the best value that I've used for the resource configurations. Cost is assumed to be 0.00001 per core second, which is the same as the cheapest resource. The optimal value is rather unrealistic in the sense that to achieve a minimum value in either the cost or the makespan time, a tradeoff must be made with the other factor. The sampling algorithm is designed to minimize both cost and makespan. As the number of samples are increased you can see that the average makespan and cost decreases for both sets of workflows. For the linear chain workflows, it appears that the optimimum cost calculation may have more room for improvement, since there seems to be many outliers when compared with each other and with the optimal value. For the general workflows, the makespan for the sampling algorithms seems to level out around 16 and 32 samples, and I don't increase the sample count beyond 32, since further improvements will not be achieved by increasing the sample count.

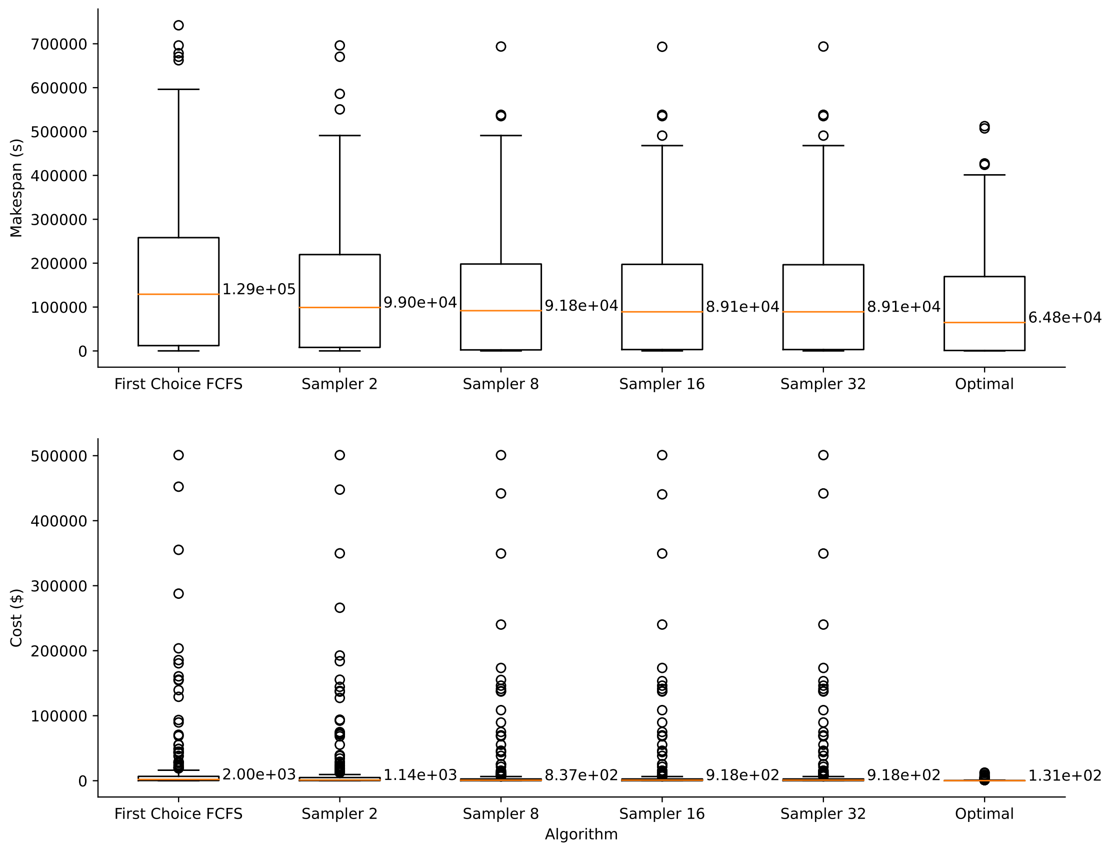

Figure 15: Results of running the simulator with the linear chain workflows.

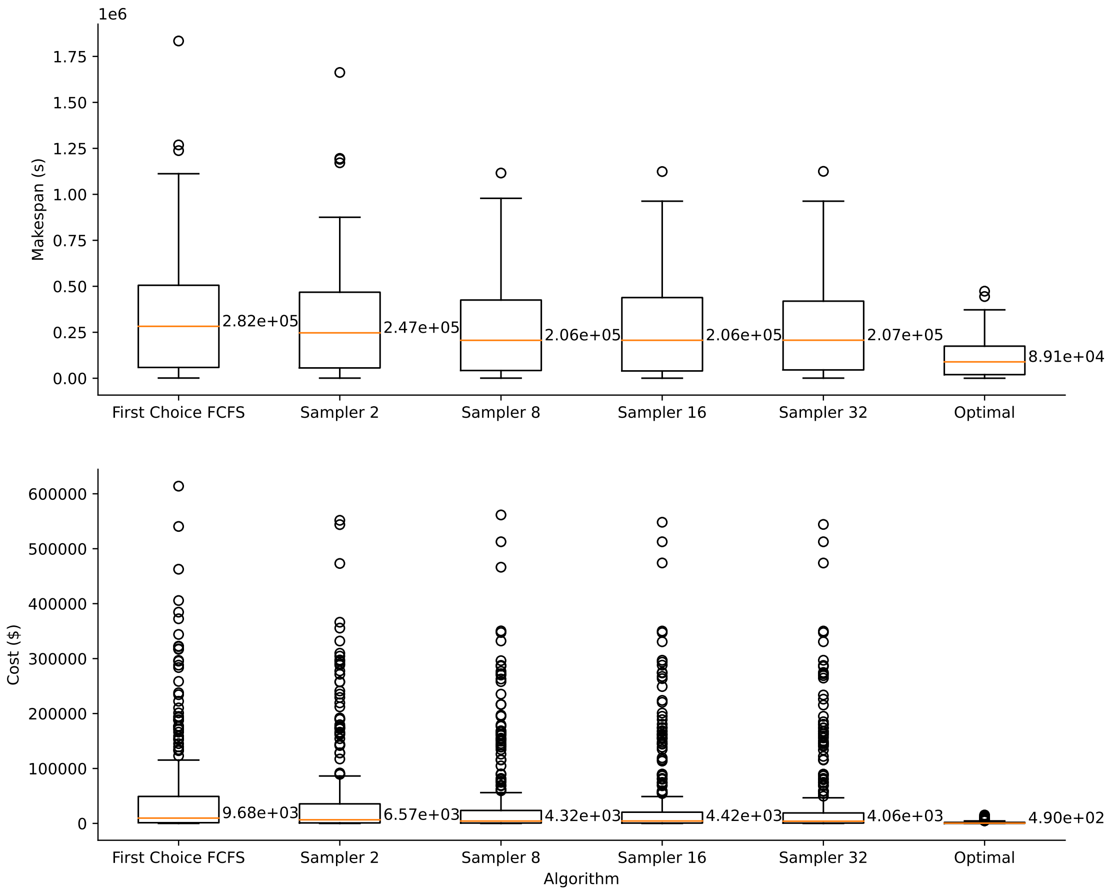

Figure 16: Results of running the simulator with the general workflows.

CHAPTER 6

Discussion

In this chapter I will discuss the results of this work and possible improvements that could be made with extensions and other modifications. I will also try to detail some of the challenges that I and others have faced when implementing and designing some of the components and interfaces within BEE.

I demonstrate BeeSwarm here with a number of different HPC applications and cloud configurations. The results given only represent those configurations that I was able to test with. For fuller application scalability tests, it will likely be necessary for scientists and developers to choose very specific configurations in order to stress-test applications or ensure that applications have access to proper hardware and networking to ensure optimal performance. While I perform my tests in the cloud, it may also be possible for this type of testing to be done internally, within HPC centers or private clouds. For example, these types of tests could be done on a testbed cluster or on a system with a light load. For some HPC applications those configurations may show more realistic results. What is important to note here, is that it is possible to set up BeeSwarm in this scenario, and other similar applications may attempt to connect CI and HPC platforms in order to do scalability testing like this.

Turning to the other aspect of my work with scheduling, my implementation attempts to take into a number of different factors that can affect efficiency, cost and makespan time. But there are many other factors that can all play a role in scheduling and efficiency. These include other values such as reliability, which could give a measure of how likely a resource is to produce correct results or whether it will be able to complete execution of a task. For some workflows users may want to run multiple versions of the same task on different resources to ensure stability of results or for a certain degree of fault tolerance given that some resources could fail during the process. Working with a number of different HPC applications and workflows, I've realized that each one is extremely unique and may require special configurations in order to function properly. Specialized schedulers, tuned to a single workflow, may be extremely useful for this purpose.

6.1 Future Work

There are many possible directions for future work. The extensions that I have worked on for BEE are also applicable to other workflow orchestration tools. Other tools that are designed for different fields may find use for some of the research that BEE is founded on and some of the research that led to BeeSwarm.

Originally, within BEE data was expected to reside on a shared file system which all tasks have access to. When running BEE with multiple Task Managers, this design is no longer feasible and some method of data transfer management is needed. A prototype implementation using the HTTP REST API can work for small examples, but will quickly overwhelm BEE with larger data requirements. Instead I believe that some sort of new data management component is needed for BEE. This component would be in charge of transferring data across system boundaries when they do not both utilize the same shared file system. I believe this tool could also offer new research opportunities for improving the provenance of workflow data and could also help in sharing data among different research groups. Perhaps this tool could make use of some sort of cloud-based data back end, such as an object store, as well as in-house storage servers, to allow for maximum flexibility.

BEE's cloud interface and cloud support could also be improved by allowing for PaaS usage and also for better configuration management mechanisms. There are not a huge amount of HPC PaaS and SaaS services right now, but as time progresses there will likely be more providers who are willing to offer similar software. I believe that allowing BEE to launch tasks with these types of cloud services would be extremely useful for further easing the launch and management of HPC applications. Further work could also be done to make configuration more general as well. Right now OpenStack [26] and Google Compute Engine [28] are both supported by BEE. Addition of more providers would be very useful. There is also a need for configuration to work with multiple providers at the same time. For this, it may be useful to invest in other existing projects like Terraform [29].

There is also a need for supporting more complex workflow dependencies within workflow orchestration systems such as BEE. In the past, some initial support was added to BEE for more complex dependencies, such as in-situ dependencies. For in-situ workflows, tasks often depend on other tasks that must be running at the same time. They may be pulling chunks of data from another task that is continuously producing the data. This introduces more complexity, since data transfer might have to be managed here and also the scheduling of these dependent tasks need to note this. There are also other complexities that may arise, such as the need for tasks to utilize some constantly running resource, such as a database that must be stopped and started. Some may argue that this is outside the scope of a workflow orchestration system, or that a database such as this should not be used within a workflow, but nevertheless in some cases there are configurations that require this.

BEE currently has support for storing hashes of workflows and DAG metadata for a given system. I believe there may be some opportunities to add more support for data and code provenance. For instance, there is the possibility of adding support for hashing input data as well as intermediary files for dependencies between tasks. In some special cases whole files or smaller files specifying particular parameters could be saved during BEE's workflow archival process. However for larger files, saving the file data is simply not feasible and hashing seems the only route here at the moment. There are future possibilities for making provenance easier, such as integrating BEE into special shared data systems and object storage systems that can allow scientists to share results easily across platforms and security domains. Management of provenance data is definitely a big area for future research and design decisions, not only for BEE, but for other workflow systems and scientific automation software.

CHAPTER 7

Conclusion

In conclusion, this work builds on top of existing HPC orchestration and workflow research, using the Build and Execute Environment (BEE) as the basis. An HPC scheduler implementation, code for executing workflows across systems is developed and tools for testing HPC application scalability are designed and analyzed.

BeeSwarm introduces a wrapper around BEE that enhances CI pipelines, allowing scalability testing to be done in pipelines which were previously only designed for correctness testing. I also introduce methods for extending BEE to run workflows across systems and I demonstrate the use of a scheduler component to schedule workflows that take account of cost, data transfer and total makespan time. This will allow for more complex HPC workflows and also ensures that workflows are not limited to the resources that are only available within a single HPC platform. Future workflows written with CWL will be able to support all platforms that a user has access to.

Both BEE and its extension BeeSwarm are designed to abstract underlying issues of software portability and execution. As HPC systems grow, so too will the workflows and applications that run on them. New software is needed to manage the complexities of HPC software to allow for those applications to be manageable and usable on these systems. As an orchestration system, BEE is designed to add to this field and help users of HPC software to manage and improve software by abstracting away the underlying systems and runtime dependencies that are used for systems.

As future workflow systems and the underlying platforms continue to grow, so too will the workflows that need to run on those systems. Workflow orchestration systems still have a need for more research and development, both from a more theoretical perspective, as well as from a practical development perspective, in order to design systems and applications that will last.

BIBLIOGRAPHY

[1] Jenkins. `https://www.jenkins.io/`, 2022.

[2] The moby project. `https://github.com/moby/moby`, 2022.

[3] Mpi forum. `https://www.mpi-forum.org/`, 2022.

[4] S F Altschul, W Gish, W Miller, E W Myers, and D J Lipman. Basic local alignment search tool. *J Mol Biol*, 215(3):403–410, October 1990.

[5] Inc. Amazon Web Services. Aws batch. `https://aws.amazon.com/batch/`, 2022.

[6] Amazon Web Services, Inc. Cloud services - amazon web services (aws). `https://aws.amazon.com/`, 2022.

[7] Edoardo Apra, Karol Kowalski, Jeff R. Hammond, and Michael Klemm. Nwchem: Quantum chemistry simulations at scale. 1 2015.

[8] Arvados Project. Arvados. `https://arvados.org/`, 2020.

[9] Anne Benoit, Ümit V. Çatalyürek, Yves Robert, and Erik Saule. A survey of pipelined workflow scheduling: Models and algorithms. *ACM Comput. Surv.*, 45(4), aug 2013.

[10] Jieyang Chen, Qiang Guan, Xin Liang, Louis James Vernon, Allen McPherson, Li-Ta Lo, Zizhong Chen, and James Paul Ahrens. Docker-enabled build and execution environment (bee): an encapsulated environment enabling hpc applications running everywhere, 2017.

[11] Tao Chen, Rami Bahsoon, and Xin Yao. A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems. *ACM Comput. Surv.*, 51(3), jun 2018.

[12] Zelin Chen, Yoshihiro Omori, Sergey Koren, Takuya Shirokiya, Takuo Kuroda, Atsushi Miyamoto, Hironori Wada, Asao Fujiyama, Atsushi Toyoda, Suiyuan Zhang, Tyra G. Wolfsberg, Koichi Kawakami, Adam M. Phillippy, , James C. Mullikin, and Shawn M. Burgess. De novo assembly of the goldfish (carassius auratus) genome and the evolution of genes after whole-genome duplication. *Science Advances*, 5(6), 2019.

[13] Andrew Chung, Jun Woo Park, and Gregory R. Ganger. Stratus: Cost-aware container scheduling in the public cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 121–134, New York, NY, USA, 2018. Association for Computing Machinery.

[14] Iacopo Colonnelli, Barbara Cantalupo, Ivan Merelli, and Marco Aldinucci. StreamFlow: cross-breeding cloud with HPC. *IEEE Transactions on Emerging Topics in Computing*, 9(4):1723–1737, 2021.

[15] Michael R. Crusoe, Sanne Abeln, Alexandru Iosup, Peter Amstutz, John Chilton, Nebojsa Tijanic, Hervé Ménager, Stian Soiland-Reyes, and Carole A. Goble. Methods included: Standardizing computational reuse and portability with the common workflow language. *CoRR*, abs/2105.07028, 2021.

[16] Rafael Ferreira da Silva, Kyle Chard, Henri Casanova, Dan Laney, Dong H. Ahn, Shantenu Jha, William E. Allcock, Gregory Bauer, Dmitry Duplyakin, Bjoern Enders, Todd M. Heer, Eric Lancon, Sergiu Sanielevici, and Kevin Sayers. Workflows community summit: Tightening the integration between computing facilities and scientific workflows. *CoRR*, abs/2201.07435, 2022.

[17] David John Daniel and Aimee L Hungerford. Lanl asc advanced technology development and mitigation: Next-generation code project (ngc). Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2016.

[18] Wayne Davison. rsync. `https://rsync.samba.org/`, 2022.

[19] Tu Mai Anh Do, Loïc Pottier, Stephen Thomas, Rafael Ferreira da Silva, Michel A. Cuendet, Harel Weinstein, Trilce Estrada, Michela Taufer, and Ewa Deelman. A novel metric to evaluate in situ workflows. In *Computational Science – ICCS 2020: 20th International Conference, Amsterdam, The Netherlands, June 3–5, 2020, Proceedings, Part I*, page 538–553, Berlin, Heidelberg, 2020. Springer-Verlag.

[20] Dmitry Duplyakin, David Johnson, and Robert Ricci. The part-time cloud: Enabling balanced elasticity between diverse computing environments. In *Proceedings of the 8th Workshop on Scientific Cloud Computing*, ScienceCloud '17, page 1–8, New York, NY, USA, 2017. Association for Computing Machinery.

[21] ExMatEx. Comd proxy application. `http://www.exmatex.org/comd.html`, 2012.

[22] Hamid Mohammadi Fard, Radu Prodan, and Felix Wolf. Dynamic multi-objective scheduling of microservices in the cloud. In *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, pages 386–393, 2020.

[23] Dror G. Feitelson, Dan Tsafrir, and David Krakov. Experience with using the parallel workloads archive. *Journal of Parallel and Distributed Computing*, 74(10):2967–2982, 2014.

[24] Rafael Ferreira da Silva, Rosa Filgueira, Ilia Pietri, Ming Jiang, Rizos Sakellariou, and Ewa Deelman. A characterization of workflow management systems for extreme-scale applications. *Future Generation Computer Systems*, 75:228–238, 2017.

[25] Christoph Fischer, Stephan Koblmüller, Christian Gülly, Christian Schlötterer, Christian Sturmbauer, and Gerhard G. Thallinger. Complete mitochondrial dna sequences of the threadfin cichlid (petrochromis trewavasae) and the blunthead cichlid (tropheus moorii) and patterns of mitochondrial genome evolution in cichlid fishes. *PLOS ONE*, 8(6):1–14, 06 2013.

[26] Open Infrastructure Foundation. Open source cloud computing infrastructure - openstack. `https://www.openstack.org/`, 2022.

[27] The Apache Software Foundation. Apache Airflow. `https://airflow.apache.org/`, 2022.

[28] Google. Compute Engine. `https://cloud.google.com/compute`, 2022.

[29] HashiCorp. Terraform by hashicorp. `https://www.terraform.io/`, 2022.

[30] HTCondor Software Suite (HTCSS). HTCondor Overview. `https://htcondor.org/htcondor/overview/`, 2022.

[31] IBM. IBM Spectrum LSF Suites. https://www.ibm.com/products/hpc-workload-management.

[32] IBM. Power systems servers — ibm. https://www.ibm.com/it-infrastructure/power, 2022.

[33] Sylabs Inc. Fakeroot feature. https://sylabs.io/guides/3.5/user-guide/fakeroot.html, 2019.

[34] Sylabs Inc. Oci runtime support. https://sylabs.io/guides/3.1/user-guide/oci_runtime.html, 2019.

[35] Broad Institute. Broad institute. https://www.broadinstitute.org/, 2022.

[36] Omar Javed, Joshua Heneage Dawes, Marta Han, Giovanni Franzoni, Andreas Pfeiffer, Giles Reger, and Walter Binder. Perfci: A toolchain for automated performance testing during continuous integration of python projects. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1344–1348, 2020.

[37] Jenkins. Performance. https://plugins.jenkins.io/performance/, 2021.

[38] Ian Karlin, Jeff Keasler, and Rob Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, August 2013.

[39] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. Lessons learned from the chameleon testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[40] Dalibor Klusáček, Boris Parák, Gabriela Podolníková, and András Ürge. Scheduling scientific workloads in private cloud: Problems and approaches. In *Proceedings of The10th International Conference on Utility and Cloud Computing*, UCC '17, page 9–18, New York, NY, USA, 2017. Association for Computing Machinery.

[41] Christoph Laaber and Philipp Leitner. An evaluation of open-source software microbenchmark suites for continuous performance assessment. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 119–130, New York, NY, USA, 2018. Association for Computing Machinery.

[42] LF Projects LLC. Apptainer. https://apptainer.org/, 2022.

[43] Dong Li, Yi Wei, and Bing Zeng. A dynamic i/o sensing scheduling scheme in kubernetes. In *Proceedings of the 2020 4th International Conference on High Performance Compilation, Computing and Communications*, HP3C 2020, page 14–19, New York, NY, USA, 2020. Association for Computing Machinery.

[44] Junwen Liu, Shiyong Lu, and Dunren Che. A survey of modern scientific workflow scheduling algorithms and systems in the era of big data. In *2020 IEEE International Conference on Services Computing (SCC)*, pages 132–141, 2020.

[45] Rupert W. Nash, Michael R. Crusoe, Max Kontak, and Nick Brown. Supercomputing with mpi meets the common workflow language standards: an experience report. *2020 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, Nov 2020.

[46] Neo4j, Inc. Graph data platform — graph database management system — neo4j. `https://neo4j.com/`, 2022.

[47] Neo4j, Inc. The neo4j cypher manual v4.4. `https://neo4j.com/docs/cypher-manual/current/`, 2022.

[48] Pallets. Jinja. `https://jinja.palletsprojects.com/en/3.0.x/`, 2007.

[49] Pegasus WMS. Pegasus wms – automate, recover, and debug scientific computations. `https://pegasus.isi.edu/`, 2022.

[50] Inc. Perforce Software. The complete continuous testing platform — blazemeter. `https://www.blazemeter.com/`, 2022.

[51] Tim Grance Peter Mell. The nist definition of cloud computing. `https://csrc.nist.gov/publications/detail/sp/800-145/final`, 2011.

[52] Podman. Podman. `https://podman.io/`, 2022.

[53] Poetry. Poetry - Python dependency management and packaging made easy. `https://python-poetry.org/`, 2022.

[54] Reid Priedhorsky, R. Shane Canon, Timothy Randles, and Andrew J. Younge. Minimizing privilege for building hpc containers. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2021.

[55] Reid Priedhorsky and Tim Randles. Charliecloud: Unprivileged containers for user-defined software stacks in hpc. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[56] SchedMD. Slurm workload manager - documentation. `https://slurm.schedmd.com/`, 2021.

[57] SchedMD. Scheduling configuration guide. `https://slurm.schedmd.com/sched_config.html`, 2022.

[58] Mehmet Soysal. The kit forhlr ii log. `https://www.cs.huji.ac.il/labs/parallel/workload/l_kit_fh2/index.html`, 2019.

[59] The kernel development community. Namespaces. `https://www.kernel.org/doc/html/latest/admin-guide/namespaces/index.html`.

[60] The Linux Foundation. Kubernetes. `https://kubernetes.io/`, 2022.

[61] Alfred Torrez, Timothy Randles, and Reid Priedhorsky. Hpc container runtimes have minimal or no performance impact. In *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, pages 37–42, 2019.

[62] Jacob Tronge, Patricia Grubel, Timothy Randles, Quincy Wofford, Rusty Davis, Steven Anaya, and Qiang Guan. Bee orchestrator: Running complex scientific workflows on multiple systems. In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 376–381, 2021.

[63] Jake Tronge, Jieyang Chen, Patricia Grubel, Tim Randles, Rusty Davis, Quincy Wofford, Steven Anaya, and Qiang Guan. Beeswarm: Enabling parallel scaling performance measurement in continuous integration for hpc applications. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1136–1140, 2021.

[64] University of Wisconsin–Madison. What is high throughput computing? `https://chtc.cs.wisc.edu/htc.html`, 2017.

[65] Vimal L. Vachhani, Vipul K. Dabhi, and Harshadkumar B. Prajapati. Survey of multi objective evolutionary algorithms. In *2015 International Conference on Circuits, Power and Computing Technologies [ICCPCT-2015]*, pages 1–9, 2015.

[66] S S Vazhkudai, B R de Supinski, A S Bland, A Geist, J Sexton, J Kahle, C J Zimmer, S Atchley, S H Oral, D E Maxwell, V G Vergara Larrea, A Bertsch, R Goldstone, W Joubert, C Chambreau, D Appelhans, R Blackmore, B Casses, G Chochia, G Davison, M A Ezell, E Gonsiorowski, L Grinberg, B Hanson, B Hartner, I Karlin, M L Leininger, D Leverman, C Marroquin, A Moody, M Ohmacht, R Pankajakshan, F Pizzano, J H Rogers, B Rosenburg, D Schmidt, M Shankar, F Wang, P Watson, B Walkup, L D Weems, and J Yin. The design, deployment, and evaluation of the coral pre-exascale systems.

[67] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[68] Laurens Versluis, Roland Mathá, Sacheendra Talluri, Tim Hegeman, Radu Prodan, Ewa Deelman, and Alexandru Iosup. The workflow trace archive: Open-access data from public and private computing infrastructures - technical report. *CoRR*, abs/1906.07471, 2019.

[69] Jeffrey Vetter and Chris Chambreau. mpip: Lightweight, scalable mpi profiling. 2005.

[70] John Vivian, Arjun Arkal Rao, Frank Austin Nothaft, Christopher Ketchum, Joel Armstrong, Adam Novak, Jacob Pfeil, Jake Narkizian, Alden D. Deran, Audrey Musselman-Brown, Hannes Schmidt, Peter Amstutz, Brian Craft, Mary Goldman, Kate Rosenbloom, Melissa Cline, Brian O'Connor, Megan Hanna, Chet Birger, W. James Kent, David A. Patterson, Anthony D. Joseph, Jingchun Zhu, Sasha Zaranek, Gad Getz, David Haussler, and Benedict Paten. Toil enables reproducible, open source, big biomedical data analyses. *Nature Biotechnology*, 35(4):314–316, Apr 2017.

[71] Kate Voss, Geraldine Van Der Auwera, and Jeff Gentry. Full-stack genomics pipelining with gatk4 + wdl + cromwell [version 1; not peer reviewed], 2017.

[72] Jia Yu, Michael Kirley, and Rajkumar Buyya. Multi-objective planning for workflow execution on grids. In *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, GRID '07, page 10–17, USA, 2007. IEEE Computer Society.