METHODS ENABLING PORTABILITY OF SCIENTIFIC WORKFLOWS

A Dissertation

Submitted to the Graduate School

of the University of Notre Dame

in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

by

Nicholas Hazekamp

Douglas Thain, Director

Graduate Program in Computer Science and Engineering

Notre Dame, Indiana

December 2019

METHODS ENABLING PORTABILITY OF SCIENTIFIC WORKFLOWS

Abstract

by

Nicholas Hazekamp

Scientific workflows are common and powerful tools used to elevate small scale analysis to large scale distributed computation. They provide ease of use for domain scientists by supporting the use of applications as they are, partitioning the data for concurrency instead of the application. However, many of these workflows are written in a way that couples the scientific intention with the specificity of the execution environment. This coupling limits the flexibility and portability of the workflow, requiring the workflow to be re-engineered for each new dataset or site.

I propose that workflows can be written for pure scientific intent, with the idiosyncrasies of execution resolved at runtime using workflow abstractions. These abstractions would allow workflows to be quickly transformed for different configurations, specifically handling new datasets, diverse sites, and different configurations. I examine three methods for developing workflow abstraction on static workflows, apply these methods to a dynamic workflow, and propose an approach that separates the user from the distributed environment.

In developing these methods for static workflows I first explored Dynamic Workflow Expansion, which allows workflows to be quickly adapted for new and diverse datasets. Then I describe an algorithm for statically determining a workflow's storage needs, which is used at runtime to prevent storage deadlocks. Finally, I develop an algebra for transforming workflows, which isolates site and configuration specific designs to be applied to workflows as needed. These methods were combined and applied to a dynamic workflow, adapting a site bounds MPI application to a dynamic cloud workflow.

I combine these methods and formulated the Continuously Divisible Jobs abstraction to separate the domain scientist's application from the distributed logic of a dynamic workflow. This abstraction defines an API which applications can implement to allow for dynamic distributed computation, showcasing the flexibility and portability provided through workflow abstractions.

CONTENTS

FIGURES

TABLES

ACKNOWLEDGMENTS

First and foremost, I want to thank Dr. Douglas Thain, my advisor, who always had the time to discuss the big ideas. Dr. Thain has always pushed me ask the right questions, verify my results, and have pride in the solutions we create. I deeply appreciate all of the opportunities that have offered to through time with Dr. Thain, everything from teaching tutorials and classes, traveling for conferences, and having a graduate experience where my time was valued.

I also want to thank:

Dr. Scott Emrich, who has been a mentor and supporter of my research since my graduate career began. Dr. Emrich has always provided helpful insight into the many bioinformatics projects I undertook, and makes an effort to keep in touch and apprised of my progress since moving to UTK.

Dr. Nirav Merchant, who has provided a wealth of work through our collaborations. I have appreciated the enthusiasm you bring to discussing different problem solutions and your willingness to find time for our remote discussions.

Dr. Jarek Nabrzyski, Dr. Paul Brenner, and the Center for Research Computing for supporting all of the computing I could attempt. Even though I have not always been an ideal computing tenant, the CRC finds the time help find a solution or discuss different approaches.

Dr. Aaron Striegel, who helped guide my early graduate career with Operating Systems where I learned how to read and discuss research.

Dr. Jakob Blomer and the CVMFS team who supported my work and collaboration for the Shrinkwrap project, as well as my wonderful summer in Geneva, for which I will always cherish.

Dr. Ben Tovar who has through all projects been a great sounding board for ideas and programmatic approaches. Ben has help mold me into a decent programmer, having spent patient afternoons walking through some large and messy pull-requests.

Nate Kremer-Herman and Tim Shaffer who are always willing to talk through ideas. The honest and upfront approach with which you both discuss ideas, whether related to research or not, has provided helpful feedback and refreshingly candid conversations. I am only sad that our future lunches are limited and I will have no more conferences to experience with you both. Thanks for keeping it interesting.

Charles Zheng, Peter Ivie, Haiyan Meng, Patrick Donnnelly, and the whole Cooperative Computing Lab whose many collaborations helped give me a broader view of distributed computing. Throughout my career, the many member of the CCL, both past and present, have been instrumental in keeping aware of the field and what talents we all bring to the table.

Joyce Yeats and the Computer Science Administrative staff, without whose help I would never have been able to get a degree, buy a house, or find answers to innumerable questions.

My parents, Jeff and Gwen Hazekamp, who have supported me through all of my endeavors, academic and otherwise. Their wonderful support and the occasional 'shouldn't this be done already' has motivated me to finish.

And lastly as a constant source of encouragement, I thank my wife, Laura Hazekamp. Laura has helped and supported through my whole graduate career, motivating me to finally be done. Despite the concerns of my lengthening graduate program, she resisted the urge to press, remaining steadfast in support.

CHAPTER 1

INTRODUCTION

1.1 Scientific Workflows

With the increased demands of scientific computing, there is a constant search for improved ways to harness available resources from clusters, clouds, and grids. A variety of fields are experiencing a boom of data production and there is increased need for user friendly approaches to process the vastly growing data. Workflows are a main avenue scientists are utilizing to exploit these resources for analysis[118], with examples in a number of fields such as bioinformatics[4, 12, 48, 50, 55, 78], high energy physics[35, 124], astronomy[36]. Workflows are used to organize a set of tasks describing the transformation from inputs into outputs. Scientific workflows are advantageous since they can be written once and do not need to be tailored or compiled for the specific resource chosen. Well organized workflows can be scaled to any amount of data produced and leverage resources as required.

Users working to process large scale data find workflows provide a tractable solution. Workflows are designed with a specific application or processing pipeline in mind, based on the developer's experience and data. Workflows are based on many assumptions from the developer such as data, execution site, and configurations. Common cases include assumptions like: adequate partition size for efficient resource usage, target type or number of resources being provisioned, or even options passed to the underlying application. If assumptions do not change, the initial workflow will perform consistently on the desired resources.

Workflows designed for a specific purpose are hard coded. Using workflows designed for other configurations requires updating, but even in easily configurable workflows, changes can be time consuming and resulting effects may be unknown. Users adapting existing scientific workflows need to understand the tunable variables and how these variables relate to their data. The process of finding a good configuration may be difficult, including determining partition granularity, managing several workflows at once, and providing mechanisms to adjust for new sites or configurations.

I will define workflows in two categories: **static** with systems such as Pegasus[38], Kepler[3], and DAGMan[31]; and, **dynamic** with systems such as Parsl[10], Work Queue[16], and SWIFT[121]. Static workflows make one-time bulk partitioning decisions on the data. Data is partitioned and then operated on concurrently. Static workflows enable more complex or strict scheduling and resource handling as their apriori knowledge of tasks and dependencies can be used. Dynamic workflows partition data on demand to process large data sets or iterate over many parameters. The dynamic nature of partitioning allows workflows to converge on results, use available resources, or modify the search space and configurations. Dynamic workflows allow for more complex computational loops, but require a higher level of understanding of the analysis and workflow computing to perform efficiently.

In static workflows, the partitioning of the data set is determined by a static partition size or number of partitions desired. Cases using static sizes, changes to data structure or density affect performance. This is exacerbated when the number of partitions is static as concurrency does not track with increased size. Using bioinformatics as an example, differences between genome contigs and large scaffolds, hundreds of base pairs versus thousands respectively, introduces an order of magnitude difference on execution and resource needs. Inappropriate partitioning can result in two scenario: under- and over-sized. Under-sized partitions leads to

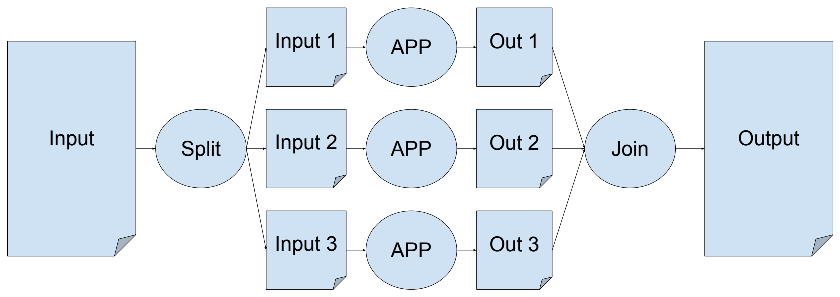


Figure 1.1. Diagram showing Example static workflow. Typical static workflow which makes an initial static partitioning decision, executes an application on each partition, and joins the results.

increased overhead, limited concurrency (result of shorter tasks), and lower utilization of resources (from limited bandwidth of the coordinator). Over-sized partitions leads to limited resource capacity (from a lack of available partitions), more expensive fault-tolerance (from reduced granularity), and larger resource requirements to accommodate execution.

To show a simple illustration of this problem I varied the partition size of a BWA workflow, a bioinformatics alignment application. The BWA workflow takes a query and reference genome, partitions the query genome, and performs alignments between the query partitions and the reference. This BWA workflow maps cleanly to the example workflow shown in Figure 1.1. Figure 1.2 shows the affect partitioning has on the execution time of BWA. This result measures execution time for different numbers of partitions of the query genome sequence. The log scale graph shows a valley where performance is reasonable, with many sizes yielding acceptable performance. At the edges of the graph we see orders of magnitude longer execution as a result of under- and over-sizing.

As a result of partitioning decisions resources are acquired (or intermittently needed) beyond the scope of the initial workflow state and used concurrently with

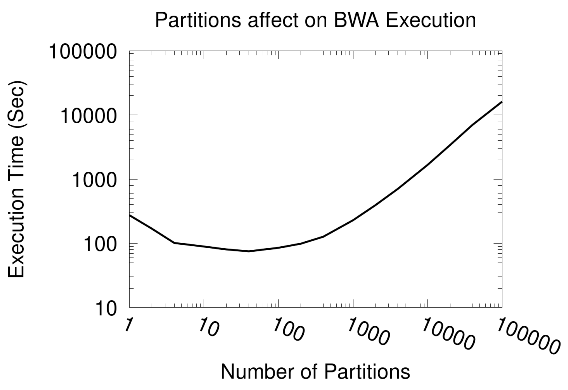


Figure 1.2. Partitioning's affect on BWA execution. This is a sample bioinformatics(BWA) workflow's performance with the input partition size varied. The number of partitions was varied from 10 to 100,000, using a statically size query of 1,000,000 sequences.

the resources needed to execute the workflow. This means that additional storage is needed for intermediate and partially completed tasks, resources for managing and retrieving external data, and management of out-of-workflow resources such as databases and servers. Persistent usage of storage provides a clear illustration where additional storage accrued in execution then leads to execution limitations and failures (i.e. deadlocking the workflow in need of storage space). This problem is exacerbated as users run multiple workflows concurrently with no control or coordination mechanism.

Workflows that are properly sized and have resources appropriately allocated indicates the workflow was designed with a specific site, resource, and configuration in mind. As a user looks to change these expectations, they are left to change the underlying workflow to reflect the desired environment. Containers serve as a premier example as they are both commonly used in scientific computing and are delivered in a number ways such as Docker[89] and Singularity[74]. Containers are used in scientific workflows to replace the underlying operating system and environment configuration. Existing workflows can be modified to build the container into the existing tasks, but this creates new dependencies and does not pertain to the original intent of the scientific workflow. If the workflow is modified for a specific container, the new workflow may only be suitable for sites using that container engine, requiring updates for future or different technologies. The difficulty is further extended for additional changes needed, resulting in tasks with undefinable or indefinite definitions.

I will demonstrate several methods for increasing the flexibility and portability of workflows. Methods include how to cleanly expand static workflows, track resources required, and transform tasks and workflows for new configurations. These methods are applied to dynamic workflow as proof of wider applicability. Finally, I will define the Continuously Divisible Jobs abstraction for bridging the simplicity of static definition with the flexibility of dynamic execution.

Most workflows are designed with the intention of pure scientific analysis, then cluttered with site and configuration specific implementations. Incorporating methods for abstracting the execution specifics from the scientific intention allows sound scientific workflows to be used more flexibly; adapting to new data, sites, and configurations without changing the underlying analysis.

1.2 Methods for Improving Workflow Flexibility

I will be examining methods for improving the flexibility and portability of workflows, first by developing methods for static workflows and then by applying them to dynamic workflows. Three main areas of focus will be partitioning and expanding workflows, analysis and management of sustained resources needs, and methods for transforming existing workflows to accommodate new requirements. Though these are not exhaustive when looking at improving flexibility, they tackle several of the main concerns encountered by scientific users. The culmination of these methods will be presented through the Continuously Divisible Job abstraction, as a novel approach to workflow design.

Dynamic Workflow Expansion allows for a template workflow to be defined and expanded at runtime. This uses static partition sizes and known application behavior to expand applications to workflows in target systems with limited user interaction. Dynamic Workflow Expansion has three objectives for expanding the workflow behind a user interface to reduce the user's required knowledge and make partitioning decisions that target the software and site of deployment while delivering results comparable to normal execution. This allows the user to seamlessly employ the application workflow in a larger analysis pipeline.

Workflow Resource Management uses static analysis of the workflow to determine the expected size, then employs this knowledge for runtime management of the workflow. This concept relies on the static nature of directed acyclic graph (DAG) workflows to analyze the full workflow, estimating the need and lifetime of resources. One method uses strict scheduling of the DAG to limit overall size and overlapping resource usage. This method requires knowledge of resources requirements and accurate execution time to effectively execute concurrently. While difficult to know apriori, an incorrect estimations reduces the overall performance significantly. The method developed relaxes the expected knowledge of execution time, relying solely on resources and allowing logical constraints to be placed on the DAG without strict scheduling. The workflow can utilize the maximum concurrency without over committing resources that will be needed later. The underlying design statically analyzes task footprints to determine the allocated resources and their commitment length. This approach is used at runtime to determine if an execution branch can reserve the requisite resources, allowing concurrency within specified limits.

Workflow Transformations allow workflows to be dynamically adapted for new requirements without changing the underlying workflow. Changing the size and parity of the data while running concurrently requires recreating the necessary environment for task execution. As new resources or execution sites are adapted the original expected environment needs to be distributed. This environment extends beyond the simple definition of task inputs and includes information not pertinent to the scientific analysis but necessary for execution. This approach uses a strict task sandbox to outline a task definition and how tasks are transformed for new environments. This is demonstrated using the base case of a single task: capturing the tasks behavior and then applying several task transformations to adapt to new configurations.

As outlined above, these three methods are used to increase flexibility of static workflows, but can also dynamic workflows. This is explored by apply the lessons learned to dynamic workflows by examining how to transform tasks to: accommodate complex software configurations, manage dynamic task sizing, and provide clear tools with feedback to the user to make informed decisions about resource allocation.

Continuously Divisible Jobs abstraction is based on the methods and lessons learned from both static and dynamic workflows. The importance of separating the application mechanism, which the user understands, from the execution policy, which relies on distributed computing knowledge, is emphasized. An abstraction for creating static applications that enable dynamic partitioning and execution is needed. The proposed solution, Continuously Divisible Jobs, defines a simple API that allows an application to encode how data is manipulated and the analysis executed. This API is then used by job coordinators to enable different partitioning models, scheduling methods, and execution systems. Utilizing Continuously Divisible Jobs decisions to handling data can be determined based on the underlying systems and abstracted away from the application level allowing users to focus on writing applications, and using job coordinators for allocating available resources.

1.3 Overview of this Dissertation

Each chapter is briefly outlined, including an outline of the conclusion. A pictorial overview is shown in Figure 1.3. Chapters are labeled numerically on the overview, with a corresponding description in the caption.

Chapter 2 focuses on **background and related work**, exploring existing technologies and related work. Specifically this section looks at existing work in scientific workflows, differentiating between static and dynamic workflows. It explores existing work in data partitioning, resource provisioning, and storage analysis. Next, highlighting research related to different approaches for capturing, replicating, and providing environments. Then I delve more in depth on the software used throughout this research, including examples of static and dynamic workflow systems.

Chapter 3 focuses on the **dynamic workflow expansion**. Dynamic workflow expansion is explored to create BWA and GATK workflow that are expanded behind the Galaxy user-interface. The goal of dynamic job expansion is to capture a logical workflow created by the user and expand each step into a workflow of concurrent tasks. Relying on a model of the software behavior, dynamic job expansion accelerates performance with minimal user interaction. This chapter focuses on alleviating the upfront cost from a user's perspective, while avoiding under- and over-sizing of partitions in a target system.

Chapter 4 focuses on the **static and dynamic resource management** which analyzes the entire workflow to prevent deadlock from storage exhaustion. The algorithm developed analyzes a static DAG and reports the estimated minimum and maximum storage needed by a workflow. This data is then used for dynamic workflow management, which accounts for allocated storage space. This technique is able to consistently prevent deadlock and safely execute a workflow within a user specified limit. This chapter focuses on analysis and management of intermittent and accumulated resource needs of scientific workflows.

Chapter 5 focuses on **workflow transformations**. Workflow Transformations aim to address the problems that arise in accurately executing partitioned work on different resources. I explore how to model and then transform workflows to provide a consistent, correct task execution environment. This is accomplished by defining a model of isolation using task sandboxes. Using strictly defined task sandboxes, tasks and transformations can be arbitrarily nested allowing for flexible task and workflow handling.

Chapter 6 focuses on **applying static methods to dynamic workflows**. Key challenges in this work looked at: providing the correct environment for a dynamic partition, handling complex input and output setups, and scaling using both external concurrency and internal parallelism. Using WQ-MAKER, the local parallel performance using MPI was compared with the remote concurrent performance using Work Queue, a master-worker framework. Following this, the Work Queue implementation was used to analyze several larger datasets to show the continued scalability of the framework and flexibility of applied methods.

Chapter 7 focuses on **an abstraction for bridging static and dynamic workflows**. Introducing the Continuously Divisible Jobs model, which defines an API for applications that is used by job coordinators for execution. The goal is to separate the mechanism of the application and the execution on a platform, using the API and job coordinators respectively. Leveraging the flexibility of dynamic workflows to avoid bad configurations, while using static application definitions for easier creation. This work illustrates the need for Virtual Files to abstract the logical data partitioning from physical data movement. I show that by using Continuously Divisible Jobs in conjunction with Virtual Files, execution consistently outperforms static workflows.

Chapter 8 **Conclusions** focus on evaluating the overall effectiveness and applicability of the different methods defined in this dissertation. Reiterating the contributions as they apply to workflows and looking more broadly at how these methods apply to scientific computing in general. I then briefly discuss the software that was created in the process of completing this work, peer-reviewed publications that resulted, and the broader impacts of this work.

1.4 Example Applications

Now I will outline several of the consistently used applications throughout this dissertation's research. The applications will be briefly introduced, as well as an outline of their typical behavior. These applications provide examples of behavior seen in a number of fields, such as bioinformatics and high-energy physics.

BWA[77] employs the Burrows Wheeler Transform algorithm to align genome queries. BWA is a light-weight alignment tool that supports paired-end mapping, gapped alignment, and various file formats like ILLUMINA [30] and ABI SOLiD [53]. The output format is SAM (Sequence Alignment Map), which can be analyzed using a number of tools such as GATK and the SAMtools package [79]. In related work[27] we observed that the runtime of BWA is roughly proportional to the product of the reference and the input size. Partitioning the reference is possible, but increases the complexity of joining and negatively impact the benefit of cached files. The joining phase of the BWA workflow separates the header and content of each of the output, generating a single result using a single header and concatenated results.

Genome Analysis Toolkit(GATK)[94] employs a Bayesian algorithm to compare aligned sequences with the reference. GATK provides a number of functions, such as HaplotypeCaller and UnifiedGenotyper, used for variety analysis in genomics. HaplotypeCaller functions by indexing the input set and creating walkers to locate variations between the query and reference. Once a difference is detected, HaplotypeCaller performs local assemblies to fill gaps or correct mistakes. This produces a output that expresses how closely alignments match, the match confidence, and other statistics of the analysis. GATK is generally used to prune alignment information for later analysis. The execution time of GATK is dominated by the size of the reference file, attempting to map the full reference in memory. Thus, partitioning both the query file and reference allows for performance that outweighs the additional data handling, allowing consistent performance on smaller machines. There is added complexity to join results from correcting quality scores, but the effect on execution time is less than serial execution.

MAKER[18] is a bioinformatic pipeline used to annotate genomic information. MAKER utilizes standard programs in bioinformatics to customize the processing and preparation of the raw data. This includes processes to identify repeats, align ESTs and proteins to a target genomes, predict genes, and quantify the quality of the results based on the provided evidence. MAKER focuses on automating the entire annotation process to create an easy and consistent initial annotation. MAKER is still under active development and is used in many areas of organism modeling. MAKER is interesting because as a pipeline there are a large number dependencies and hurtles that need to be overcome for setup and execution. These include a number of restrictions such as database handling, hardcoded installation paths, and specific data locations and names. Additionally, MAKER supports MPI internally which provides opportunities for increased concurrency on workers.

Dimuon Detection is a high energy physics application that analyzes events from the Large Hadron Collider(LHC) for dimuon activity. The input to this detection relies on the event data presented in ROOT[14] format. This format is large and can be cumbersome when analyzing events concurrently. This leads to interesting opportunities for rearranging data and handling in scientific workflows.

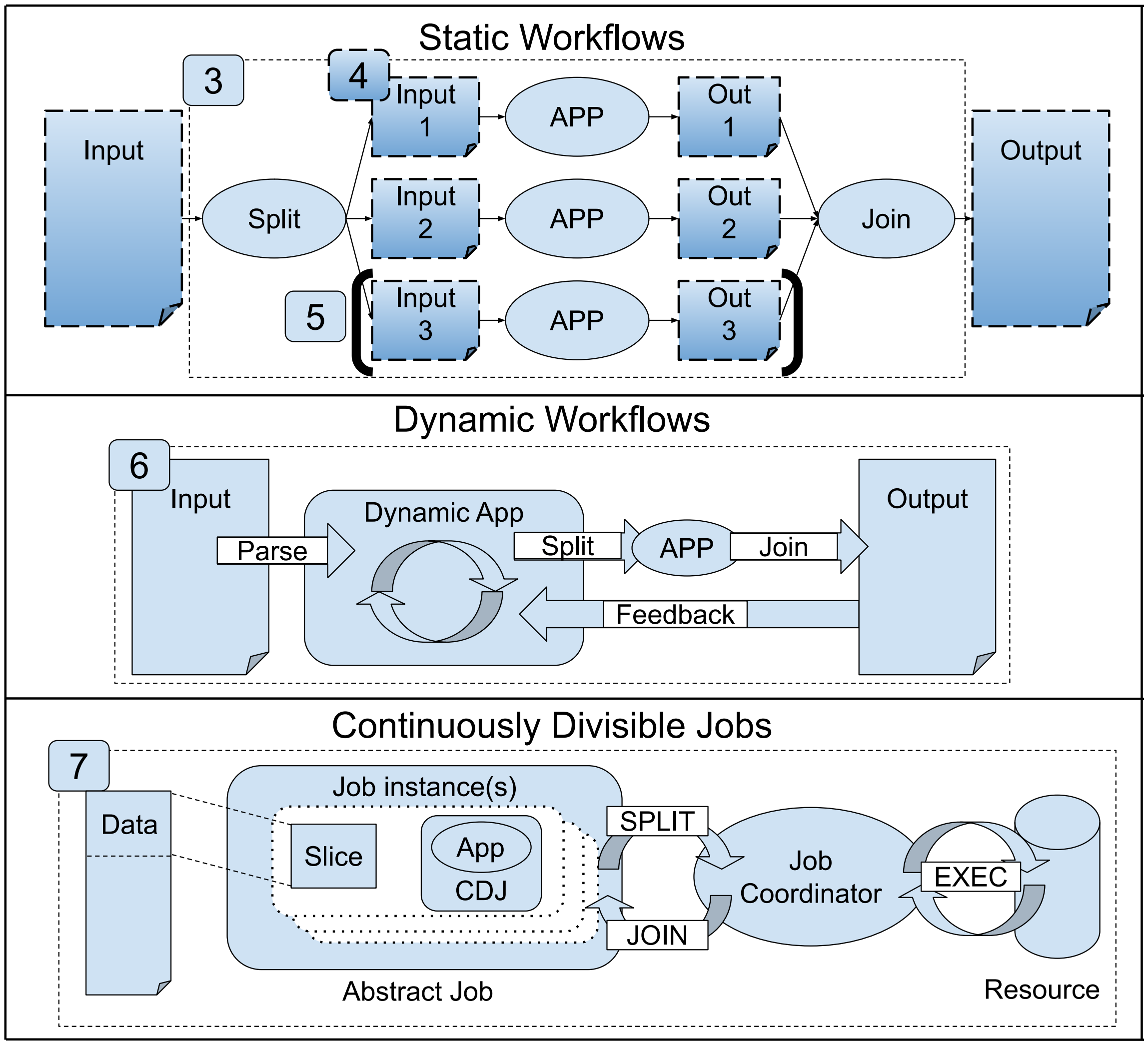


Figure 1.3. Map of dissertation. The top graphic illustrates a static DAG, which partitions a dataset, runs an pipeline of applications, and the concatenates the results. Chapter 3 looks at dynamically expanding a dataset and pipeline to create a workflow, as well as handling data partitioning sizes. Chapter 4 then looks at statically analyzing storage needs of a workflow and dynamically managing it. Chapter 5 follows, exploring how to address environment needs using workflow transformations. The middle graphic shows a dynamic workflow, whose partitions are determined during execution. The lessons learned for static workflows are applied and discussed in Chapter 6. Finally, bridging the two categories, Continuously Divisible Jobs are introduced in Chapter 7 and enable a static application definition to use dynamic runtime behavior, with the goal of avoiding bad partitioning decisions.

CHAPTER 2

RELATED WORK

In this chapter, I will outline and discuss existing work related to this dissertation. The primary focus is on existing workflow systems, where I will separate static (Section 2.1) and dynamic workflows (Section 2.2). I will also briefly highlight some related batch execution systems(Section 2.3). Following discussion of workflow systems, I will review research related to the problems tackled and solutions used throughout task and workflow resource provisioning (Section 2.4), workflow storage analysis, and management (Section 2.4.1), and environment creation tools such as containers (Section 2.5). Finally, I conclude with a more in depth look at the tools used for this research, specifically Makeflow and Work Queue (Section 2.6).

2.1 Static Workflow Systems

Pegasus [38, 40] is a workflow management system which utilizes directed acyclic graphs (DAG) similar to Makeflow. It makes data flow and movement a priority and attempts to optimize these in scheduling. Pegasus focuses on grouping and managing the individual tasks of the workflow to partition the work while minimizing communication overhead[37]. This was explored further to combine results of the workflow partitioning with the resource provisioning[22] and then refined for storage constrained conditions[21]. Pegasus explored a number of different workflow optimizations such as task clustering, job throttling to limit traffic, and pre-staging of data.[20] Pegasus[37, 38] is Workflow Management System(WMS) that focuses on static planning of workflows on remote sites. It utilizes a multi-stage method to plan, execute, monitor, and re-use workflows across multiple platforms. Pegasus's similar DAG structure would allow methods like dynamic-job expansion and workflow transformations to be applied in very similar ways, though with Pegasus's more verbose task structure care would need to be taken when translating transformations. Additionally, Pegasus's emphasis on static planning may limit the interface with Continuously Divisible Jobs, where task are dynamically created.

DAGMan[31] is a DAG manager built as a meta-level job manager on top of HTCondor. DAGMan controls the order and speed tasks are submitted to HTCondor based on workflow dependencies. DAGMan relies on HTCondor to allocate and control the resources needed for a task, trickling jobs in as dependencies are met. Much like Pegasus and Makeflow, DAGMan operates on DAG workflows, allowing the dynamic workflow expansion and workflow transformations to apply cleanly to the defined abstraction. DAGMan supports a similar role in job throttling to the job coordinator described in Continuously Divisible Jobs, but the static DAG nature limits the flexibility. Unlike Pegasus and Makeflow, the dependecies for DAGMan are describe purely at the job level, rather than through files. This limits the clear mapping of the static and dynamic storage manage as DAGMan does not define a global view of files and their lifetimes.

Snakemake[76] allows for static workflow definitions that exploit file wildcards. It is written in Python and allows data scaling without the need to rewrite workflows, functioning similar to what was achieved with dynamic workflow expansion. Snakemake is growing in popularity as it supports a number of bioinformatic tools out of the box, is easy to learn and write, and support Python, shell, and R scripts directly. Snakemake assumes a similar task structure to what is used in other workflow systems, which would allow workflow transformations to be applied. The flexibility and ease of use are tempered by the limited support for batch systems; targeting newer cloud platforms but supporting only Sun Grid Engine(SGE) for clusters. Snakemake determines dependencies using files and generates the workflows as jobs are executed. This limits the amount of verification that can be done on the workflow prior to execution, as well as limiting the ability to statically analyze the storage needs.

Dryad[66, 72] is a distributed execution engine geared toward data-parallel applications. It's workflows are defined as DAGs, where the vertices describe computation and the edges consist of data communications. This description is vague as Dryad supports a number of different communication pathways, such as files, TCP pipes, and shared-memory queues. The underlying design of Dryad is to allow scaling from a single machine up to large scale distributed environments. The inherent complexity of Dyrad's supported communications pathways limits the direct applicability of methods used within this papers. Dynamic workflow expansion assumes there is a partition point at which the work can be split into multiple files. This is possible with these other communication methods, but complicates workflow creation, validation, and management. Similarly, in Chapter 5 a task is defined using only files as they are a stable way to pass data through several layers. Each different method used complicates the transformation method and increases the fragility of the process based on the shell used.

PaRSEC[13] is a task scheduling system used for hybrid distributed systems. PaRSEC is a dataflow system that builds a directed acyclic graph based on the implied data connection. This graph is then traversed, launching parallel task available resources. It's focuses on leveraging available hardware accelerators for computation[125]. Additionally, in moving toward a dynamic approach, the most recent implementation provides dynamic task discovery[65] which allows tasks to be injected into an existing runtime as they are created. PaRSEC operates on sequential code, transforming the code into a DAG workflow, producing tasks as function calls in a parallel environment. For this to work, the environment is assumed to be highly connected and consistent, much akin to an MPI application. This assumption limits the effectiveness of solutions such as dynamic workflow expansion and static workflow analysis, as the workflow are derived from the application and not around a workflow structure. Additionally, workflow transformations use a strict task definition and nesting using processes, which eliminates the advantages of PaRSEC's *close to the hardware* approach.

Some workflow management systems operate as high-level applications that focus on the development and organization of workflows for a User. These workflow management systems are GUI-based, providing a rich editing environment for users and are often the first point of entry for users into the realm of workflows. These systems facilitate easy workflow design. The high-level nature of these systems lack expressiveness for dynamic concurrency when compared with the above solutions. In practice this is remedied by partitioning the data prior to submitting but this is additional burden on the users.

Galaxy[12, 48] is a web-based workflow management system focused on bioinformatics. Galaxy is built with similar goals as Kepler, with an emphasis on low barrier to entry and simple workflow development, using drag-and-drop components. To facilitate this ease of development, users are provided a curated set of applications and tools for analysis. However, tools can not be added by normal users and must be managed by system administrators. Workflows in Galaxy are comprised of applications/tools that are linked together graphically by the user to connect outputs with the tools that require them. In Galaxy concurrency is limited as workflows lack any means of dynamic partitioning. As is discussed in Chapter 3, Galaxy provide an opportunity to abstract the concurrency from the user using dynamic job expansion during execution. Additionally, job clean-up may be inconsistent, so high-level workflow storage management using static analysis and dynamic management is crucial to safely deploying dynamic workflows.

Kepler [3] is workflow management system that provides a GUI interface for user to build and refined workflows. The core design targets ease of use for users with little background in workflows or distributed computing. Kepler aims to provide tools for creating generic scientific workflows where portability and power are important. Unlike other Workflow Management systems, Kepler allows the workflow design to be separated from the computational model, more akin to typical workflow execution systems. It also offers support in tracking and managing provenance. Combining the provenance and workflow design aspects, Kepler, facilitates workflow sharing for collaboration and reproducibility. The similarities between Galaxy and Kepler allow the methods described in Chapter 3 and Chapter 4 to apply.

Apache Taverna[114, 122] allows the user to create a static workflow that is used as a template, but the degree of partitioning is static within the workflow unless other means are provided. It supports a wide range of tools and domains for scientific workflow, with three core elements. Taverna: Engine executes tasks of workflows concurrently across resources, Workbench is the desktop client application targeted for the end user, and Server empowers the user to execute workflows remotely. Like Galaxy, Taverna is comprised of applications/tools that are linked together graphically by the user to connect outputs with the tools that require them. The underlying complexity Taverna modifies the approach needed to leverage available resources. Dynamical job expansion would need to configure how to submit new jobs to the Taverna Engine for new task execution, as opposed to Galaxy where the resources were out of Galaxy's control.

Uintah[32] is a parallel design environment, where workflows are constructed of components that create and consume data to other components or into data warehouses. Uintah strives to provide a clear interface for scientists to develop and steer large scale computation, and stands apart from several of these workflow management systems in that it builds parallelism into is design. This assumption of flexible concurrency allows for powerful scaling, without having to re-engineer the entire system.

Uintah is actively continuing development targeting next-generation exascale system. The approach Uintah provides for concurrency pushes the work and complexity of defining concurrency points onto the user, which differs from the more abstracted approach examined using dynamic job expansion in Galaxy.

Prune[67] focuses on preserving and providing provenance for scientific computation and workflows. Though not specifically a workflow management system, Prune tracks the organization of workflows which allows data to be changed or updated triggering fresh computation. Prune uses Makeflow as a target back-end, interfacing to allow for complex workflows to be executed and preserved. As a result of the provenance information, these workflows and computations can be shared for collaboration, similar to the goals of Kepler and Galaxy. Storage management is crucial benefit of Prune, as files can be removed and regenerated as needed across several workflows. This helps to avoid storage deadlocks entirely, as the space is actively managed by Prune. Additionally, Prunes support for Makeflow workflows and tasks allows both dynamic job expansion and workflow transformations to be applied as needed, with the caveat that each transformation or expansion may define new paths in Prune, limiting the reuse of previously executed tasks.

2.1.1 Workflow Specification Languages

As a side-effect of creating a new and unique workflow system, a specification is created. Many of these specifications have unique design considerations such as: implicit concurrency, parameter mapping, or system specific properties. For most workflows, the core components remain the same: inputs, outputs, and analysis.

There have been many attempts to create a universal specification, but many do not reach a wide level of usage due to a lack of consistency. **Common Workflow Language**[6] strives to unify workflow specification languages. Though the Common Workflow Language does not support its own implementation, it boast support from a large number of popular workflow systems. In a similar approach, the Broad Institute has developed **Workflow Description Language**[117], which is coupled with their Cromwell implementation. Both approaches aim to be a common interface or intermediate representation of workflows, though in many cases design decisions limit their overall applicability in specific workflow systems.

Though there are minor differences between the specifications, both define tasks similar to existing workflow systems, particularly Makeflow, which allows dynamic job expansion and workflow transformations to apply, with the restriction that dynamic file names are not compatible as used in the Common Workflow Language. The Common Workflow Language allows for inputs and outputs to defined as both static file names and file name blobs which may be resolved using wildcards or Javascript functions. This dynamic and possibly non-deterministic file mapping limits the proper handling of partitioned tasks in a dynamically expanded job.

2.2 Dynamic Workflow Systems

As opposed to static workflows, dynamic workflows allow for flexible execution patterns. To achieve this flexibility, the work of partitioning the data, defining tasks, and controlling dependencies is left to the user. Once the user has defined this level of detail, they are provided a robust set of execution tools to migrate and scale work as needed. With all of these systems, the inherent dynamic behavior leads to incompatibilities with the dynamic job expansion and static storage analysis at the master level. Dynamic job expansion would be possible if applied to the resulting task submitted by the workflow system, at which point the tasks is static in definition. In the same way, workflow and task transformations assume a static definition, and would need to be applied after the task is submitted.

RADICAL-Pilot[90] is a pilot-based system that aims to provide scalable approaches on a variety of platforms. RADICAL-Pilot is written in Python, allowing easy adaption from the currently flourishing scientific python community. It is built on the SAGA[51] platform, providing support for common execution platforms such as Portable Batch System and Sun Grid Engine. RADICAL-Pilot operates very similar to Work Queue from a user's perspective by providing a clear task definition and pilot jobs for execution. These similarities would allow a job coordinator for Continuously Divisible Jobs to be created similarly to what was designed using Work Queue.

Swift[121] is dynamic workflow system built on a unique scripting language[129]. Swift's scripting language is a implicitly parallel functional language whose goal was to provide a simple language to define parallel workflows. It has even been used to investigate parallel approaches integrated with Galaxy[85], similar to what is explored in Chapter 3. The difference between these methods is how the concurrency in Chapter 3 is abstracted from the user. Swift uses compiler techniques to collapse and expand tasks within the implicit task parallelism[7]. Swift operates most similar to Continuously Divisible Jobs in that Swift applications define how a workflow operates and what operations are allowed for partitioning, and then allows the underlying engine to evaluate and execute the workflow.

Parsl[10] is a python based workflow system that separates the user from the execution engine. Applications are written in python with functions tagged as Parsl tasks, which are parsed and distributed with little user interference. The decisions on how to execute remote tasks are defined by the user specified execution engine. This allows for a number of execution techniques and systems to be combined for an application specific configuration.

Pydron[92] performs semi-automatic parallelization of python code to support multi-core and cloud execution. Pydron has a similar philosophy to Parsl, aiming to provide a simple interface for users to achieve parallelism in a language they are already using. Pydron adds two decorators to Python that allow functions to be marked as either *schedule*, saying the function should be parallel, and *functional*, saying the function is free of side-effects. *Schedule* runs the execution on the same machine, while *functional* can be run on other machines.

Both Parsl and Pydron focus on providing an easy interface for users and less on creating a distributed execution platform, favoring existing solutions configured for python tasks. From this perspective, Parsl and Pydron operate to create jobs and submits to existing job coordinators similar to abstract jobs in the Continuously Divisible Job definition. Both systems rely on the user to make concurrency decisions, whereas Continuously Divisible Jobs passes a concurrency definition to the job coordinator.

From here on, the dynamic workflow systems are similar in principle to the ones already discussed, but the underlying task structure and definitions vary highly. As a result workflow transformation is applicable in concept, but in practice the difference in task structure would need to be resolved. As an example, Spark defines tasks using inputs, outputs, and the code to execute, but environment and resource definitions are relegated to the executing platform such as Hadoop. Charm++ and Legion on the other hand define lower level tasks that may not be tied to files, but data more generally making clean task encapsulation more loosely tied to an single process execution. As a result, transformations such as applying a container don't make sense in these environments.

Spark [126] is another powerful dynamic workflow implementation, that focuses on utilizing the RAM to achieve higher performance, and could be leveraged to further boost performance. It relies on the concept of Resilient Distributed Datasets[127] to address both a performance issues as well as the storage issue created by holding the data in memory, and performs well with sufficient memory for the analysis. Spark functions similar in execution to Continuously Divisible Jobs by defining how the a type of analysis is done, relegating the concurrency to the underlying batch execution

(Hadoop). Hadoop does the data partitioning and distribution, allowing Spark to define execution.

Charm++[71] is not directly a dynamic workflow system, but relies on task-based concurrency similar to most workflows. Charm++ is a parallel programming system that interfaces with applications using C++. A C++ interface allows for applications to create Charm++ objects which are used as the base concurrent unit, define data, coupled functions, and relationship to other objects.

Legion[11] is a programming model that organizes applications into logical regions. These regions define both locality of computation, independence of data, and analysis from other logical regions. Applications that have defined their logical regions then leverage the Legion runtime system that identifies independent tasks and available parallelism. To simplify the creation of these logical regions, **Regent**[103] allows the user to write high level programs that define tasks and logical regions, and are compiled to Legion.

The last several approaches are all similar because they provide a mechanism for describing or translating concurrency into a form that can be executed in a distributed manner. Spark does this using Hadoop, Charm++ and Legion do this using threads and MPI. This approach has similar high-level goals as Continuously Divisible Jobs, mainly allowing the user to define analysis, while an automated execution system performs parallelism. However, a key difference in these approaches is the limited task execution platforms, which are assumed to be closely bound to the host node. Continuously Divisible Jobs are defined such that communication and task execution can be delegated to a number of difference services when defined as job coordinators.

2.3 Batch Computing Systems

One of the key advantages of using a workflow system is that it allows the user to leverage large scale resources for computation. To achieve this workflow system in turn rely on the underlying batch system to allocate resources and schedule tasks.

Generally, Batch systems operate as a centralized coordinator of all available resources. These systems provide an interface for submitting jobs and specifying resources. The centralized scheduler then finds available resources, places the job on the resources, and starts the job. For most large scale compute sites this is the norm, with systems such as SLURM[69], Sun Grid Engine (SGE)[45], and the Portable Batch System (PBS)[63].

HTCondor[80] is a cycle-scavenging batch system, that provides large scale computation scheduling for resources. Jobs are submitted to HTCondor labeled with resource needs (using ClassAds[97]), and when a suitable resource becomes available the job is placed. HTCondor differs from many standard batch system as it coordinates resources and submits, rather than having a single centralize decision engine.

MapReduce[33], as implemented in Hadoop, is a parallel execution engine which strives to move the computation to the data. This is achieved by partitioning the data when it enters the system, and creating replicas as specified. The combination of partitions and replicas allows the data to be operated on concurrently for each piece with no data movement. This is leverage in a number of systems, such as Spark, to provide the underlying concurrency for execution.

ATLAS PanDA[83, 84] is a large scale workload management system deployed for organizing the LHC ATLAS experiment analysis, primarily the US ATLAS collaboration. Rather than creating and organizing individual workflows, PanDA operations on all jobs submitted for analysis, and coordinates their execution on available resources.

When considering the applicability of methods explored throughout this dissertation, workflow transformations and Continuously Divisible Jobs have clear implications of possible value for any batch system. Though workflow transformations look at apply transformations to full workflows, it builds off of the single task base case. This base case is directly applicable to batch systems, where each job generally defines a single task of class of similar tasks. If a transformation evaluator was written for any of these batch systems, the lessons and techniques used could be applied quickly.

In all of these cases, each batch system defines a potential job coordinator. The consistency and similarities between how most of these systems define tasks would allow job coordinators be easily written. This in term could provide a consistent base case on which system administrators could tune or refine the job coordinator for their specific site. One except to this is MapReduce/Hadoop, which generally assumes control and partitioning of the data internally, which prevents Continuously Divisible Jobs from dynamic reactive partitioning of data.

Liu et al.[81] proposed the idea of elastic job bundling, which defines a method for implicitly decomposing large jobs into smaller jobs to reduce queue wait time. This approach is positioned between the parallel application and the underlying batch system to break up monolithic jobs. This approach has similarity to the work discussion in Chapter 3 and Chapter 7, in that the further partitioning of work is done without user input. A significant difference here is that jobs are tightly bound to a single batch system, relying on the assumed parallel environment to accommodate data movement and consistency. Additionally, partioning and join jobs acts only on virtual data reference, not physically partitioning the data as may be needed for non-shared distributed computing.

2.4 Resource Provisioning and Management

Resource provisioning can be viewed from either the resources needed by the workflow or more specifically the resources used by each individual task. Statically planned workflows employ workflow-wide provisioning to ensure there are ample resources or to minimize the needed resources through scheduling [28, 39, 70, 86, 91, 102]. Many of these approaches take into account the resources provisioned at the task level but require information at the planning stages that may not be known in a dynamic workflow. The more fine-tuned task level specification is the most useful as it allows different categories of tasks to be specified with their required resources.

Workflow systems utilize one of three approaches:

- Naive approach - Assumes task will fit on any resource. Each task is scheduled to the whole resource, and resources are harder to adjust at the master.
- Static solution - Tasks are labeled with resource needs and scheduled only on machines that can satisfy the needs. This is often backed by systems such as Classads in HTCondor[97] or resource advertising in Mesos[64]. This is done on the workflow side by setting static attributes to the task, as evidenced in the workflow systems that statically determine resource before hand.
- Dynamic solution - Task resource specification is adjusted to meet the actual performance of the workflow. An example of this is the Resource Monitor[49].

Using these approaches, a wide variety of works [43, 62, 75, 99, 110] have proposed different approaches for solving the classic DAG scheduling problem. These approaches typically focus on minimizing the overall execution time, while considering utilization of various resources as a means of determining good candidates. However, in a storage constrained environment these algorithms may over commit the system with limited calculations on the persistent storage needed to hold files produced. As we show in Chapter 4, naive accounting of storage without consideration for dependencies can lead to deadlock. When operating with more dynamic execution, as with Continuously Divisible Jobs, approaches like the resource monitor[49] could prove to be a powerful tool for capturing resource usage and relating that used to a model of the data and computation.

2.4.1 Storage Management

The persistent nature of storage use changes the nature of the problem beyond what much of this above resource aims to address. For a better understanding, there needs to be a discussion the workflow data management and how to manage data lifetimes[34]. An example of this is Overflow[113], which provides a uniform data management system over several sites based on the needs of a scientific workflow. Overflow aims to be a plug-able solution into workflow to coordinate and manage data usage, being cognizant of cost of storage and transfer.

Pegasus's strict planning allows for more strict adherence to quota limits and attempts to leverage the maximal parallelism, even across multiple sites[19]. This strict planning also allows Pegasus to pre-stage data to limit task lifetimes when storage is available[26]. It also allows for changes to a workflow, doing so internally by clustering tasks[23–25], spawning clean-up tasks to remove files[104], and restructuring the workflow[52]. Pegasus has also explored using a data store for long term storage and staging data in as needed for execution[115]. However, in none of these cases does the workflow analyze the expected storage needs of the workflow, limiting the effectiveness of planning in storage constrained environments. The similar structure and expectations of Pegasus means that the static analysis and runtime management used in Chapter 4 could also apply. The case of managing storage across several sites, as is often done in Pegasus, is more complex, but the core file lifetimes and footprints still apply.

2.5 Environment Re-Creation

As part of making workflows more flexible and mobile it is crucial to be able to capture, preserve, and replicate a task or workflow execution environment[47, 82, 106, 130]. A variety of solutions exist within this space, and understanding these available technologies is key to providing adequate abstractions for any solution.

A common consideration when looking for a way to build a reproducible environment is to use container technologies such as Docker [89], Singularity [73, 74], CharlieCloud[95], NERSC's Shifter[46], or Slacker[54]. Containers provide a means of creating reproducible execution environments. They can differ from the execution platform even at the operating system level. Similar to containers, Umbrella[88] captures the state of execution and data, so that it can be used for reproduction. Umbrella expands on the expectations of containers to capture every aspect of an analysis for preservation, as opposed to the more constrained container builds.

Similarly there are several automation systems [44, 98] that rely on system configuration to provide consistent environments. Richer formats such as RPMs and DEBs provide version requirements for software. However, both solutions are installed with administrative privileges.

Virtual Cluster Builder[111], does not replicate the original operating system but builds the tools on the current system, allowing Virtual Cluster Builder to operate at the user-level. The goal of this system is not to provide the exact execution environment, build the required tools in the new environment. This allows the Builder to provide a consistent software stack on a flexible platform.

Each of these approaches have different benefits and restrictions, but may be the correct solution needed depending on the application and sites configurations. These different approaches are discussed in Chapter 5, where applying transformations that update or recreate an environment are some of the most common, powerful, and complex transformations. The methods explored in here can also be seen in use for Chapter 6, where the Virtual Cluster Builder was used to compile and configure a complex MAKER software stack. This software stack was shown to be flexible and consistent across hundreds of cores. Lastly, these methods provide a solid basis, for quickly and consistently deploying job coordinators for Continuously Divisible

Jobs, where execution needs to be consistent between a number of heterogeneous distributed nodes, as seen in Chapter 7.

2.6 Makeflow and Work Queue

Makeflow[2] is a system built to create and run workflows using a syntax that is very similar to that of classic Make, and now expanded to support a JSON like syntax called JX[100]. Each rule in the workflow must explicitly state input and output dependencies along with the command to run. Makeflow can dispatch jobs to a variety of execution systems, including Condor, SGE, and Work Queue.

The simple syntax supports DAG-structured workflows, which are ideal for transformation of input to output over a number of predefined steps. This syntax also makes it easy to write or have a script write rules for a workflow based on the inputs which allow for the dynamic creation of performance workflows based on the provided input, and a means to execute them. Makeflow is great for running workflows on a distributed platform as it supports several different execution engines. For improved flexibility, Makeflow assumes there is no shared file system and provides the execution engines with the information about file dependencies. Since many workflows are structured as a series of defined tasks that can be mapped across the input; scripts can readily describe the workflow as a Makeflow. This architecture also performs well as a lightweight workflow manager that can express resources needed by tasks and schedule when these requirements are met.

Work Queue [16] is a lightweight master-worker platform. Workers consist of a process that is started at an execution site and communicates with the master process to retrieve, execute, and return tasks. While preparing the task, the required dependencies described by Makeflow or the dynamic workflow application are staged and a sandbox is established. To utilize a worker, the site runs the worker, allowing workers to be created on any supported platform or through a batch system. Work

Queue provides several benefits that help performance workflows. Workers are processes that persist outside of the execution of single tasks. This allows the master to cache files on the worker and reuse them to limit multiple transfers to a single worker. Caching helps to limit the execution time as well as the number of workers needed to impact performance. This benefit can be extended by utilizing ”multi-slot” workers on multi-core machines. If a task is labeled with resource requirements and the worker is larger than the task's requirements, multiple tasks can be scheduled on the same worker. When a worker performs several tasks, they share a cache that limits transfers and total disk usage. This also prevents over-subscription of workers, which is crucial when running MPI or multi-threaded tasks.

Worker persistence also enables workers to be given tasks as soon as they are available, without the overhead of waiting for the task to be scheduled through the batch system. Side-stepping the batch scheduler allows short tasks to be rapidly scheduled, with less overhead. The uninstantiated workers still need to go through the scheduling process, but once at an execution site they can be utilized. Further, a Work Queue pool can be created that scales active workers up and down as need arises. As more workers are needed, the pool will submit them to the specified resource, and lets them time out when more exist than are needed.

CHAPTER 3

DYNAMIC WORKFLOW EXPANSION

3.1 Introduction

In the previous chapter, I reviewed a large selection of systems available for writing, managing, and executing applications concurrently using workflows. A core conceit of workflows is the leverage they provide users to scale applications with minimal modifications is a concurrent manner. Many workflow systems are expressed as easy to use, but any system has a learning curve that must be overcome for the results to be worth the work. Once a workflow system is selected the user is locked into the specifics of the system they have chosen with only a cursory knowledge of workflows and how they would need to use them in their execution. The question becomes, how do we provide the performance and flexibility of workflows in a opaque way, such that the user see no difference in behavior of their data pipeline, only an improvement in performance.

A great test bed for answering this question is in the context of workflow management systems such as Galaxy[12, 48, 50], Taverna[123], and Kepler[3]. Workflow management systems provide an high-level view of workflow design, linking files and tools to define the **logical workflow** for data processing. Abstracting the execution environment from the organization of the workflows allows workflow management systems to provide flexible and resilient experience for users. However, the simplicity of expression make specifying dynamic concurrency difficult. As data becomes larger there is either limited or no mechanism to increase concurrency. Enabling additional parallelism must then come as either a change to the workflow management system's job handling, or an extension of the job structure already in place.

Dynamic job expansion provides a solution where each job of a logical workflow is expanded at runtime into a **performance workflow** in which the majority of the work is done in parallel, accelerating the performance of the original process. The purpose is to create tools that are indistinguishable from sequential tools, while supporting greater concurrency. Abstracting concurrency from the user allows dynamic job expansion to be utilized by workflow management system without user involvement and enables the parallelization to be specialized to the local environment. Ideally this method could apply to any computation that consists of data-independent sub-groups, which we find to be prevalent in bioinformatics.

3.2 Galaxy

Galaxy is a web portal that was created to provide biologists with ready access to a number of bioinformatic tools. It creates an environment where the user only supplies inputs and selects the tool to perform computation. This abstraction provides biologists with a generic analysis framework without the need for specifying resources. The administrator of a Galaxy instance configures the underlying infrastructure and provides the tools available to the users. The portal records how and when the tools are run, so that reproduction of an experiment is consistent and easy. Each tool consists of a predefined web page that serves as a launch for the tool, as well as scripts in the background to provide the correct setup.

Galaxy also provides the user with a means of creating DAG-based workflows. Workflows use the data dependencies between tools to determine order, and provide a means of creating a logical flow for processing inputs in a consistent way. Workflows created may be configured with predefined parameter values and can be shared so that analysis can be reproduced easily by numerous parties.

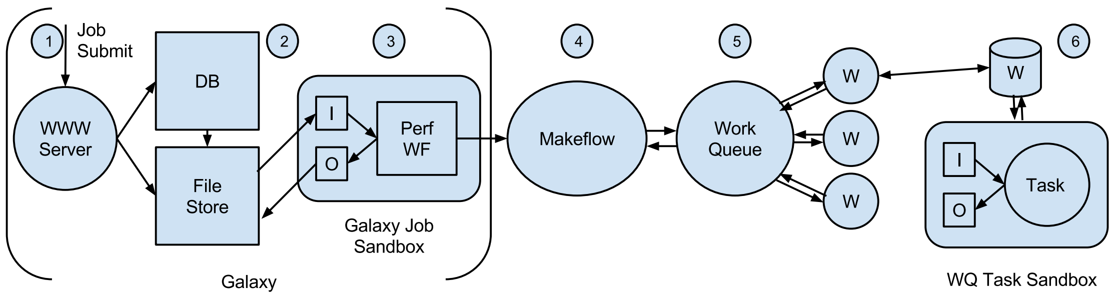


Figure 3.1. This diagram shows the dynamic job expansion process. In Stage 1, the job has been created by the user from the tool launch page. Once Galaxy gets the launch, the job is given an id, a working directory is created, and the job is added to the history. Stage 2, the files selected at launch are located via the file database, and the location is communicated to the job. Stage 3, inputs are collected, either directly or linked, in the job sandbox. Following setup, a script creates the performance workflow. Stage 4, Makeflow is launched with the performance workflow in the job sandbox, and the workflow begins processing. Stage 5, a Makeflow creates a Work Queue master that communicates with workers to create execution locations. Stage 6, the worker receives task, retrieving the inputs and task information. The task is computed and the output delivered back to Work Queue, who relays this to Makeflow. The performance workflow will move through stages 4, 5, and 6, until the workflow is complete. After completion, stage 4 will finalize the outputs and copy it to the output location defined by Galaxy. If successful, stage 3 is cleaned up, and the wrapper process concluded. At stage 1 Galaxy will change the job status and the user will be informed.

The portal creates an environment through which users can analyze data without the hassle of data and resource management. Galaxy's abstraction makes using tools quick and easy, while offering a platform to create meaningful results. It also provides a means of sharing results and the process required to achieve them. Work can then be verified and analyzed by many people at once, allowing for easy collaboration.

3.3 Dynamic Job Expansion

Dynamic job expansion is the run-time transformation of a single job in a **log-**

ical workflow into a **performance workflow**. The resulting performance workflow must be logically indistinguishable from the original job by accepting the same input files and generating the same output files, while hiding the complexity from the user. Figure 3.1 gives an overview of dynamic job expansion.

For each type of logical job to be expanded, we must write a **job expansion tool**. This is a command-line tool that is invoked in an identical manner to the underlying application. Instead of running the application directly, it writes out the desired performance workflow, invokes the performance workflow manager, and waits for it to complete. From the perspective of the logical workflow manager, the job expansion tool *is* the job to be run.

Naturally, job expansion must be specialized to a given application and must take advantage of details of the application's structure and performance. For many applications, a split-process-join pattern is effective. The initial step of the generated performance workflow evaluates the size of the input and split the inputs into appropriately sized pieces. Then, the primary application runs on each split of the input, generating multiple outputs. The final step merges the results into a single output file. In simple cases, this might be concatenation of the outputs, while in more complex cases, it could require recomputing statistical results. The more complex cases rely on knowledge of properly dividing the work, possibly in stages, to maintain equivalence with the original tool.

Using Galaxy, Makeflow, and Work Queue, the process works as follows. For each tool in the logical workflow, Galaxy assigns the job an identifier, specifies input locations, and generates output file-handles. Galaxy creates a working directory for the job and job execution is moved to the directory where the job expansion tool can create the performance workflow. The Galaxy wrapper script links inputs to the directory and copies down the necessary execution files. The job expansion tool writes a performance Makeflow based on the input's characteristics.

After the performance workflow is created, the job expander invokes Makeflow with Work Queue as its execution environment. Makeflow verifies that the structure of the performance workflow is correct, and confirms the presence of required input files. The split and join processes, which require most of the existing files, run locally to limit the data transfer. The remaining tasks utilize a subset of the files and are run in parallel. Makeflow sends each task through the Work Queue master to workers as required files become available.

The Work Queue master communicates with workers to send tasks and inputs. When the task is ready at the worker, the process is executed in a task sandbox. The completed task returns the output to the master. Having verified that the task produced a successful return value, the output is collected and returned. Makeflow continues to submit and collect tasks until all the tasks have been completed.

Upon completing the tasks, Makeflow joins the result and the wrapper copies outputs to the specified file-handles. The Galaxy wrapper script completes and Galaxy verifies that the output files have been created. After the verification, Galaxy changes the state of the job to either successful or failed. The results are then delivered to the user, and the Galaxy job is complete.

Dynamic job expansion benefits from several characteristics of this process.

- **Hidden Complexity** Using an expansion tool to write and run a performance workflow alleviates the user need to understand the inter-process complexities and expand within the logical workflow. This allows lay users to quickly pick up a tool and use it interchangeably with the original tool, without needing to know the background process.

- **Environment Aware Decisions** Expanding the workflow at the execution environment provides several benefits. Inputs are defined and can be used to make intelligent partitioning decisions [29]. Intermediate files are managed locally and do not fill the data store and history. Processes utilizing many inputs, such as split and join, can be run locally to prevent large amounts of data traffic.

- **Flexible Execution Resources** The ability to utilize resources that are not primarily dedicated provides a flexibility in scaling. This flexibility also provides a means of users coming to a portal with their own resources.

3.4 Example Application

To demonstrate job expansion, we show a logical workflow that transforms general genomic data to aligned genotyped data. The working data is a query of 32GB of reads against a reference dataset of 36MB, which are the reference loci of Red Oak. This is done using BWA alignment, SAMtools sort, Picard AddOrReplaceReadGroups, and GATK HaplotypeCaller. When run on the working data, the BWA alignment and GATK genotyping steps take the longest to execute: 19 hours and 12 days, respectively. These two steps were used to demonstrate dynamic job expansion, and remaining intermediate steps were left as sequential jobs.

Figure 3.2 details how this particular logical workflow was expanded at runtime. The expansion strategy is slightly different for each of the two tools:

BWA implements the Burrows Wheeler Transform algorithm to align genome queries. BWA is a light-weight alignment tool that supports paired-end mapping, gapped alignment, and various file formats like ILLUMINA [30] and ABI SOLiD [53]. The output format is SAM (Sequence Alignment Map), which can be analyzed using a number of tools such as GATK and the SAMtools package [79].

In previous work[27] we observed that the runtime of BWA is roughly proportional to the product of the reference and the input size. The input dataset was partitioned into splits of approximately 50K reads each. Partitioning the reference is also possible, but would increase the complexity of joining and negatively impact the use of cached files. We kept the reference whole and distribute it to all nodes of the performance workflow. The joining phase of the BWA workflow separated the header and content of each of the output splits, then generate a single output with one header and concatenated contents.

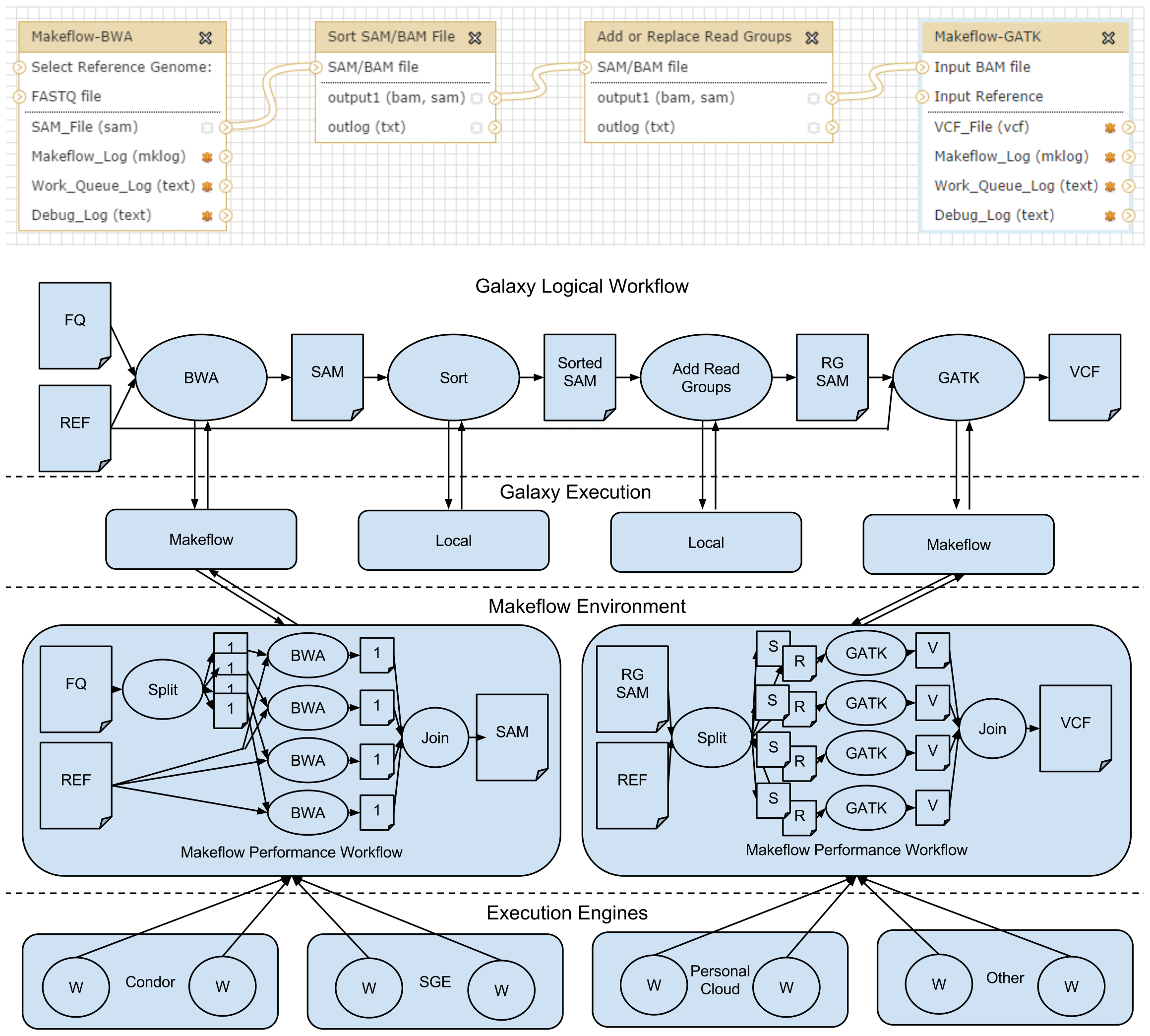


Figure 3.2. Diagram showing detail of example application. The top level shows a workflow as it is represented in Galaxy. Each box is a tool, with the names and arrows differentiating inputs and outputs. The Galaxy Logical Workflow lever simplifies the Galaxy representation to the simple logical workflow that it implies. This level shows the sequential nature of the jobs. The Galaxy Execution level defines the environment in which they are running, local being on the Galaxy instance and Makeflow denoting that the Makeflow process is local, but is creating tasks for parallelism. The Makeflow Environment level shows the process of expanding a single job. This level clearly shows the split-process-join nature of the performance workflow created. The lowest layer illustrates that workers from a number of systems can be utilized to perform the individual tasks.

GATK employs a sophisticated Bayesian algorithm to compare aligned sequences with the reference. In this workflow, GATK's HaplotypeCaller walker was used for it higher sensitivity, when compared with GATK's UnifiedGenotyper. HaplotypeCaller functions by indexing the input set and creating walkers to locate variations between the query and reference. Once a difference is detected, HaplotypeCaller performs local assemblies to fill gaps or correct mistakes. This produces a output that expresses how closely alignments match and other information about the analysis as a whole.

The runtime of GATK is dominated by the size of the reference file. Thus, the job expander splits both the query file by size and the reference by distinct reference contigs. There is added complexity after completion due to correcting quality scores, but affects the runtime of the application less than if the input were a single piece.

Using Galaxy's workflow creation system, we were able to create tools that split and join the input files. However, the tool, in order to be used in a workflow, needed to have a static number of inputs and outputs, limiting the dynamic nature of this approach. This static design requirement is lifted when expanding a job at runtime allowing the number of partition be dynamically based on the input and not an arbitrary decision made when creating a tool.

3.5 Design Considerations

In the process of implementing job expansion, we encountered several problems that resulted from converting what was previously a locally executed job into a performance workflow: dependency management, remote execution, and garbage collection.

3.5.1 Dependency Management

Problem: Jobs depend, explicitly and implicitly, on things being in the local environment. An input file is an example an explicit resource and is straight-forward to express. Explicit resources are specified in the command, and expressed by the user at launch. Implicit resources are, by nature, left unspecified and known only by the program. Implicit resources fall into two categories, the first is reference material, such as database indexes, that are expected to exist within the environment. When a user is not expected to specify these resources, modifying a tools to require expression confuses tool implementation. The second category is environment resources, Java being a great example. Java is often required, but the environment's version may not be clearly expressed. This causes confusion for the developer as they attempt to use incorrect or absent versions.

For explicit resources, Galaxy tools clearly request resources using the tool XML launch page. This handles the expression of resources such as inputs and reference files. Implicit resources such as database index files, are added to a default location within the Galaxy instance. Referencing location in the environment allows the indexes to be located by the tools at execution. This solution requires users utilize existing indexes, or add references and indexes, which is performed by administrators. Implicit resources, such as Java versions, are more difficult to deal with as they can be dynamic. Versioning, in systems like Java, creates incompatibility that are difficult to handle in the program and necessitate the program expresses these requirements. These requirements need to either be provided by the developer or expressed in a manner so tools could access the intended variable or installation.

Solution: When designing an expanded tool, both the explicit and implicit resources must be expressed prior to execution. BWA alignment requires its reference be indexed and the index be present. Many users know these files, but often overlook them as they are assumed to be present. Galaxy allows this implementation, assuming that the reference is in a common location. This limits the references that the user can use as this location file needs to be updated for every additional reference, along with having the files moved to a safe location. This was addressed here by creating the index files at job execution. This adds the index files to the local environment, but must be recreated every run. In the ideal situation, the first time an index is created it is stored and the reference is for future use. This option does not require changes to the tool implementation that utilize the indexes, but must be monitored as these files are moved and collect in the system.

Handling environment settings also presents an interesting set of problems. Java is a great example, as many programs that utilize Java require a minimum and maximum allowed version. This is remedied by utilizing a script that unpacks the required version from an archive file and sets the necessary environment variables to utilize the unpackaged version. This allows the required environment to be created at any execution site specified. Packing the environment with the tools allows freedom of movement for the execution, as well as guarantees that it is utilizing the correct tool. Environment replication is a well known issue encountered distributed computing. In our simple example, a python script was created to package Java locally prior to the workflow and one that unpacked Java at the execution site. More complex situation can be handled by tools such as Docker [89] and Umbrella [87].

3.5.2 Remote Execution

Problem: Galaxy assumes that all jobs are run and handled by the machine on which Galaxy is running. This restraint was relaxed with the introduction of CloudMan [1], which allows the user to send jobs to a number of different cloud resources. CloudMan allows for the creation of a runner that submits jobs to a cloud resource. Galaxy's assumption that each task in a workflow is a single job allows for easy mapping from a Galaxy workflow to the scheduled task on a cloud resource. An issue that arose in our implementation is that the tool creates tasks that cannot use cloud resources without having a mechanism to launch them within the Galaxy controlled environment.

Solution: This was solved by utilizing Work Queue as an execution engine. Using

Makeflow in conjunction with Work Queue, the tool uses a single port on the Galaxy machine that communicates with Work Queue workers on a variety of platforms. A pool of Work Queue workers was used to supply a steady stream of cloud resources. This allows tools to be used at different sites and execution engines. Work Queue utilizes project names and passwords to allow workers to connect without knowing the specific location ahead of time. The project names allows the tools to be configured by allowing Galaxy to name projects dynamically or for the user set it. Allowing users to devote resources to a project, even if the resources on the workflow management system are limited.

3.5.3 Garbage Collection

Problem: When running an application, it is important to create a clean namespace for the application to work in, and clean the work-space after. Galaxy enforces this by creating a job working directly when a tool is launched. Galaxy runs the entire application from within this newly created directory. Tools limit the amount of space used by specifying the absolute path to the scripts, executables, inputs, and outputs. When using this method, tools make no noticeable footprint within it and require no clean up. Some applications create temporary files and miscellaneous outputs by default that are not cleaned by the tool. The tool implementation creates partitions of the input, as well as the creation of many intermediate files that are ignored by Galaxy, and would exist only at the working directory.

Solution: The solution strives to minimize the footprint and clean the directory. At the end of the a tool execution, after the outputs are transferred back, the tool utilizes the Makeflow clean option. This option removes all intermediate files and output files that are located in the current directory. Following this, the only remaining things to clean are executables and inputs that were fully transferred to the directory. It is the responsibility of the developer to clean any files that were explicitly added, as they are neither dynamically created nor removed by Makeflow.

3.6 Opportunities

We also observed two opportunities for better integration between Galaxy and Makeflow, but did not address them in this work: expression of job status and check-pointing.

3.6.1 Expression of Job Status

Problem: Galaxy expresses a job as running, waiting, or complete. However, there is no clear indication of either a job's progress, or why a job is waiting. Additionally, the completion status relays to the user only if a job failed or completed as expected. Failures can be clarified by looking at the logs and system output, but leaves investigation of the cause to the user. The job running and completion status relay to the user the barest amount of information about the job. Some tools, such as GATK, give progress of what is being done and progress through the input. This helps estimate the time needed to finish as well as assure the user that processing continues. These simple estimates would help Galaxy users to better understand the tools and Galaxy's overall progress.

Solution: Our tool implementation does not directly address this design choice, but a simple solution may be found. If, within a tool, a status file were designated, the workflow management system could be configured to follow the tail of the status file. This would allow any tool to create a status file of a supported format to allow the workflow management system to track progress. A simple example of this would be processes reporting their percentage done. The workflow management system would need a means of expressing status in a location other than the output files history. The workflow management system is informed about the location of the working directory and status file to draw from. After this, tools need to simply create a status file to enable status updates. If a tool did not create a status file, then no more information would be relayed than already provided.

3.6.2 Checkpoints and Partial Failure

Problem: Completion in Galaxy is a binary state, where the job has either failed or completed, but there is no concept of checkpointing or partial failures. This is adequate for tools that are extensively tested and there is little possibility of system circumstances interfering with completion, such as intermittent network connection. Galaxy treats jobs as local Unix tasks, where the process is either done or not. Performance workflows, however, can be represented as partially finished, as either a result of failure or the workflow is still computing. These partially completed workflows have checkpoints, that allow for the workflow to be moved and restarted. For example, Makeflow can be rerun using the Makeflowlog, when intermediate data is present. Though workflow management system can preserve the logs, it is difficult to preserve the working directory for reuse. A manner of expressing both checkpoints and non-fatal failures allows the underlying system to better express and handle failures. Also,workflows are easily created and run, but there is no functionality that allows for workflows to be rerun, as you would for a single tool. Utilizing previously computed results in restarting a partial workflow would improve usability and reproducibility.

Solution: The solution would be to define a means of returning either the working directory, if the user has machine level access, or creating an archive for the user to modify. The user resubmits the job after determining it was a non-fatal error, or modifies to correct a user error. This package or directory would then be resubmitted through a generic tools that detects the required tool. This solution fundamentally changes how workflow management system views tools and would require careful thought for correct design.

As Work Queue utilizes network resources, failures that are unrelated to the execution tool may occur. Failures in Work Queue are not necessarily failures in the workflow, but possibly an issue on the resource on which the task ran. In this case, running the Makeflow-Work Queue process again from the same directory and log changes the outcome of the workflow, while not recomputing any previously successful work.

A mechanism within the workflow management system that allowed directly rerunning a workflow introduces a similar issue for workflows. Assuming the workflow management system allowed rerunning workflow directly, utilizing the previous results in a rerun would be as straight forward as using the links within the workflow to determine the work that has been previously done. The same result could be reached by creating a workflow log that would link previous progress to a new run of the workflow.

3.7 Evaluation

To evaluate the combined system, we implemented dynamic job expansion on BWA and GATK as described above, and then ran the combined workload in four different configurations, using a campus Condor [107] pool to provision workers on demand for the expanded jobs. We varied the amount of query data in the first three configurations, using workers configured to each run a single task. In the fourth configuration (described below), the workers were configured to run four tasks at once, sharing a local cache.

Figure 3.3 shows the timeline of execution for each configuration. The thin line of each graph shows the number of tasks available to be executed, the thick line shows the number of tasks executing, and the gray bars show data transfer over a 10 second period. Note that the left axis measures tasks ready/running, while the right axis measures data transfers. In the first graph, dotted lines indicate the phases of the

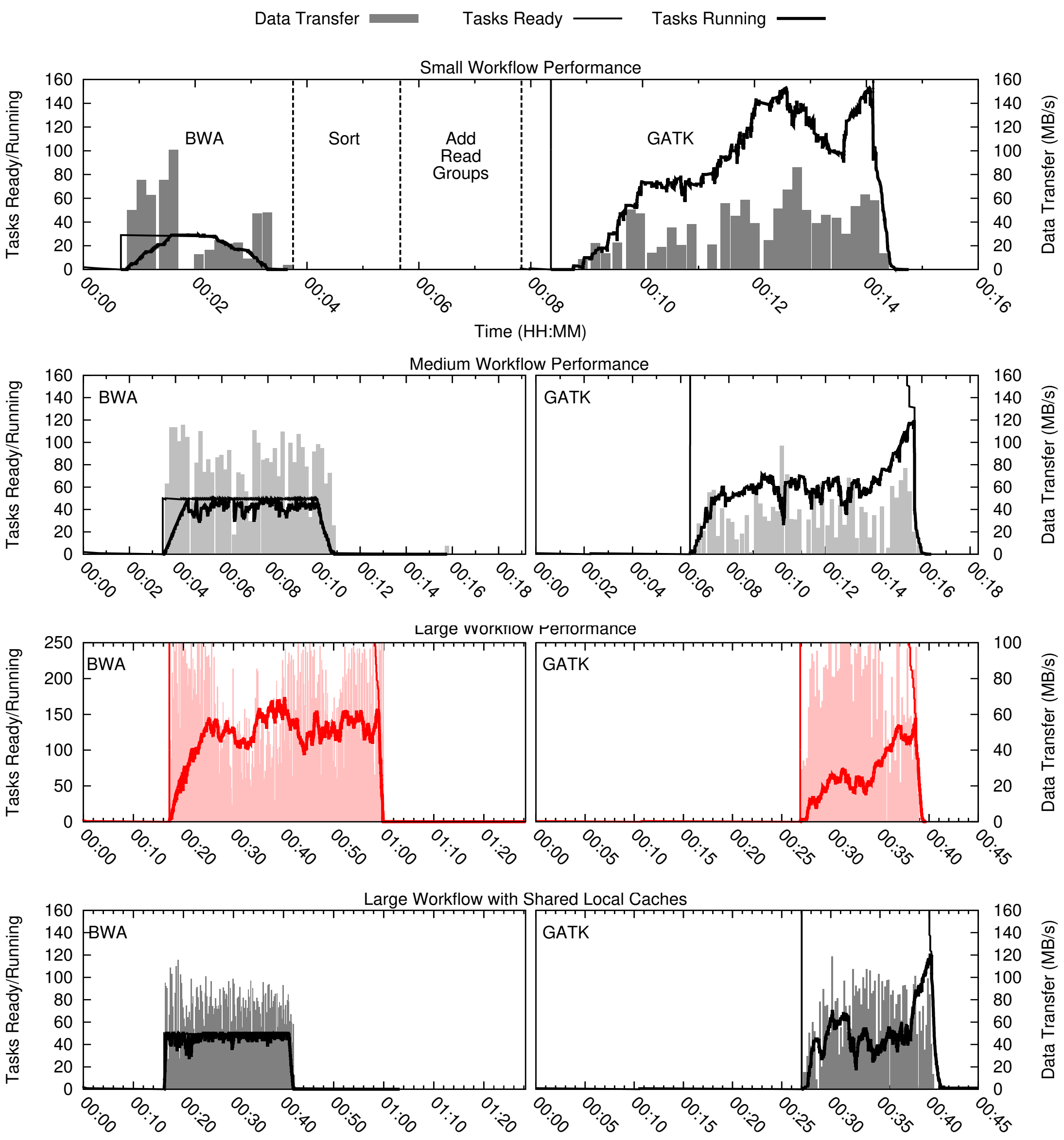


Figure 3.3. Results of BWA-GATK workflow on various datasets. This shows the execution of BWA, sorting, adding read groups, and using GATK to refine the results. The thin line of each graph shows the number of tasks available to be executed. The thick line shows the number of tasks executing. The gray bars show data transfer during a 10 second period. The left axis corresponds to the two lines, while the right axis corresponds to the data transfer. The four graphs, from top to bottom, show the small, medium, large, and large with shared cache workflows. As you can see we are able to dynamically create partitions that allowed for greater parallelism and performance in both BWA and GATK.

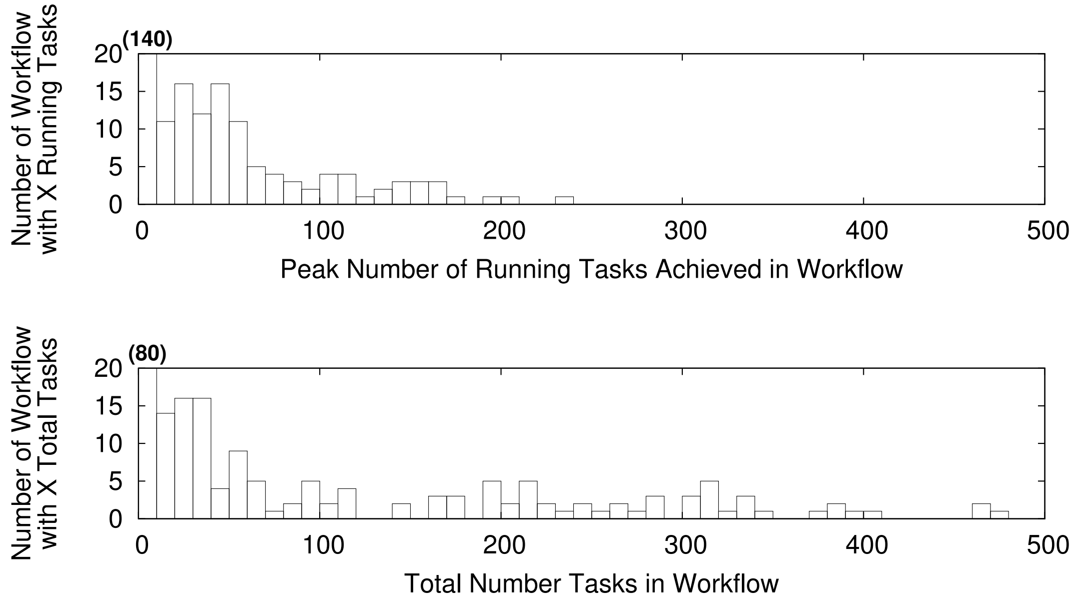


Figure 3.4. Histogram showing the sizes of dynamically expanded workflows. The top image shows a histogram of the number of dynamically expanded workflows that had a max of X workers running. The bottom image shows the number of total tasks that each dynamically expanded workflow had over the course of its execution. All workflows shown were run with Work Queue and managed through Makeflow, and contains a mix of workflows run concurrently and alone. The difference between the graphs comes from two sources. The first being the tiered nature of the workflows, only allowing a portion of the total tasks being executed at any given point due to dependencies. The second being that for BWA the amount of concurrency was limited to 50 to prevent overloading the network. The last is the potential for better task partitioning and handling in future iterations as not every situation is perfectly matched with workers.

TABLE 3.1

DATASET SIZE AND TASK PER WORKER REFERENCE

Config	**Query**	**Reference**	**Tasks/Worker**
Small	0.6 GB	36 MB	1
Medium	7.5 GB	36 MB	1
Large	32 GB	36 MB	1
Large-Shared	32 GB	36 MB	4

logical workflow. In the later graphs, only the BWA and GATK phases are shown.

On the largest dataset, the sequential version of BWA ran 19 hours, while the expanded version completes in one hour, 3 minutes. This resulted in a speedup of 18X on utilizing up to 50 workers. Figure 3.3 Row 4 Column 1 shows BWA, with the bold line in front representing the splitting task, the jagged section running BWA, and the end bold line joining the results. The sequential version of GATK ran for days, while the expanded version completes in 43 minutes for a speedup of 402X utilizing up to 125 workers. (The super-linear speedup comes from keeping the memory consumption of each task within physical memory.) Figure 3.3 Row 4 Column 2 shows splitting, GATK, and joining similar to BWA. The four-job logical workflow is accelerated by 61.5X overall.

In each configuration, it can be noted that neither the available workers nor the running tasks are ever constant. The workers vary due to competition from other users of the shared Condor pool. The variance in running tasks is due to the structure of the workload, and also due to the data transfer between the master and the workers. That is, the master can only dispatch tasks when the bandwidth can support transfer of necessary data to workers.

The main barrier to scaling up further is data transfer. For both BWA and GATK, not only must the query and reference datasets be sent, but also the software tools and dependencies such as the Java virtual machine. Each of these unique items is cached at each worker node and reused for future tasks. As the workload progresses, more workers have the necessary data cached, and parallelism can increase.

The first three configurations use worker processes that can execute one task at a time. This turns out to be inefficient, because the physical machines are multi-core and often end up running multiple workers simultaneously, each with its own distinct cache that must be managed. To improve this situation, we reconfigured the workers to consume four cores each, thus sharing a single local cache among four running tasks. The result of this can be seen in the bottom graph of Figure 3.3, where the running tasks grows more quickly and moves less data. In this configuration, we saw an 24 percent reduction in outgoing data from the master to the workers, from 69 GB to 52.1 GB. This is substantial, because as the number of concurrent tasks increases, the scalability is limited by the systems bandwidth.

The ideal scenario shown helps to showcase the benefits of this system design. To understand non-ideal situations, we looked at data gathered from the catalog server, which matches workflows with workers, over the several month period of operation to see actual behavior. Figure 3.4 shows a comparison of histograms grouped by the number of tasks in each group, running and total. The disparity between the running and total histograms shows that there are limitations in the current implementation. This is the result of several design decisions. The first is the limited concurrency through BWA. This only allows the running tasks to reach 50, despite the total number of tasks. This was done to limit the load on the network. The second is caused by general traffic on the machine that limited the ability to supply workers with work. These area provide an opportunity to improve the performance and further limit strain on the system. Figure 3.5 gives a full view of the system and the comparative concurrency.

Overall, dynamic job expansion has a dramatic impact on the overall performance of the workload, to the point where attention must be paid to improving the performance of the intermediate sequential steps.

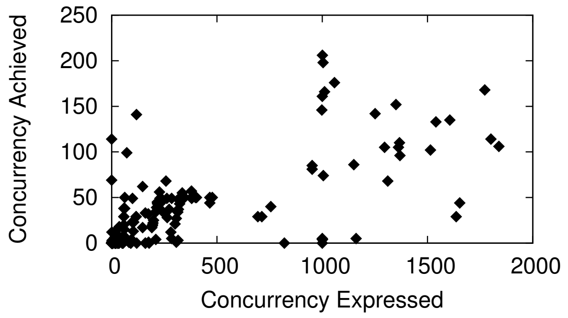


Figure 3.5. Scatter-plot of BWA-GATK usage in Galaxy. The above image shows a scatter-plot of the concurrency expressed, in terms of number of tasks, opposite of the concurrency achieved, the number of running tasks. As mentioned above, there are several reasons for disparity between expressed and achieved, but it also showcases area for improvement in workflow creation. This would benefit from modeling functions to better partition the workflows.

3.8 Conclusion

I have illustrated a method through which scientific analysis can be dynamically partitioned for concurrency with limited user knowledge or interaction. This allows the user to utilize workflows for performance with only the small learning curve of understanding the Workflow Management System; without needing to learn workflow design to utilize them. Dynamic workflow expansion shifts the control and care of resources to the resource provider, allowing to the correct usage and management to be monitored and controlled.

Dynamic workflow expansion defines the static scientific intent, and allows the data to be changed for each workflow. Workflows can now be quickly applied to a number of datasets, with the scale updating to meet the computational demands without intervention of the user. This is a first step toward creating abstractions that allow scientific workflows to be used flexibly with many configurations.

With Dynamic workflow expansion, workflows can be quickly adapted to new data and the likelihood that these will be used concurrently is inevitable. As workflows are scaled up and more are run concurrently, the increased contention for resources requires careful management to avoid failures. In the next chapter, I will be examining how static workflows can be analyzed to estimate and manage the resource needs of a workflow, specifically for storage.

CHAPTER 4

STATIC ANALYSIS AND DYNAMIC MANAGEMENT OF WORKFLOW STORAGE

4.1 Introduction

Previously I evaluated ways to abstract the creation and partitioning of the workflow without user interaction being required utilizing dynamic workflow expansion. As a result the ease of running several workflows concurrently is increased, which leads to an increased need to manage and track workflows resources. Using these methods, resource limitations can cause workflows to deadlock when the user attempts to utilize more resources than are available, which can easily be seen by hitting storage quotas. To allow for these dynamically expanded workflows to be utilized concurrently methods are needed for calculating and managing these resources more effectively.

Cores and memory, which are released on job completion, contrast sharply with storage, which persists as files and logs through the entire execution. As storage reaches its limit, the number of tasks that can run simultaneously is limited by the available storage space. A large proportion of the storage consumed may lie in intermediate files between tasks in a workflow. As a result, the order in which tasks execute has a significant effect on the storage consumption of the workflow as a whole. Additionally the life of an intermediate file may exist only long enough to be consumed by the next task and then may be deleted, or may persist through the entire workflow execution. Existing work [8] focused on basic reduction of storage usage, using clean-up jobs to delete files which are no longer needed. This chapter examines methods using the full view of the workflow, estimating and managing with global limits in mind.

I will examine two coordinated techniques that allow users to manage storage at runtime in user-level workflow-structured applications. First, I present how the graph structure of workflow applications allow me to observe the minimum and maximum storage required by a workflow *before execution* enabling a scheduler to judge whether a workflow will be able to *run to completion* with the available storage or whether more resources must be obtained. Second, I demonstrate an online accounting method by which a workflow manager tracks not only the actual storage use, but also the future storage needed to complete the current graph structure. This method is necessary to *avoid deadlock* that would be caused by a naive approach. A third technique which was explored in 'Combining Static and Dynamic Storage Management for Data Intensive Scientific Workflows' [60] demonstrates several techniques by which individual tasks may be monitored and contained so they do not overflow their expected storage consumption. A key challenge here was to *identify storage exhaustion* so the workflow manager can stop the task and re-plan.

4.2 Definition of Files and Storage in Workflows

Files are divided into three categories: *input files* which exist before the workflow begins, *output files* which are produced by the workflow and must be retained, and *intermediate files* which are created within the workflow but may be removed once they are no longer needed. The size of input files is known in advance, and estimates of the size of intermediate and output files are stated in the workflow.

Figure 4.1 shows the relationship between a workflow, running tasks, and the available storage. The workflow itself is fed into a *workflow manager* which dispatches individual tasks to a batch system that selects an execution node for each task. Each

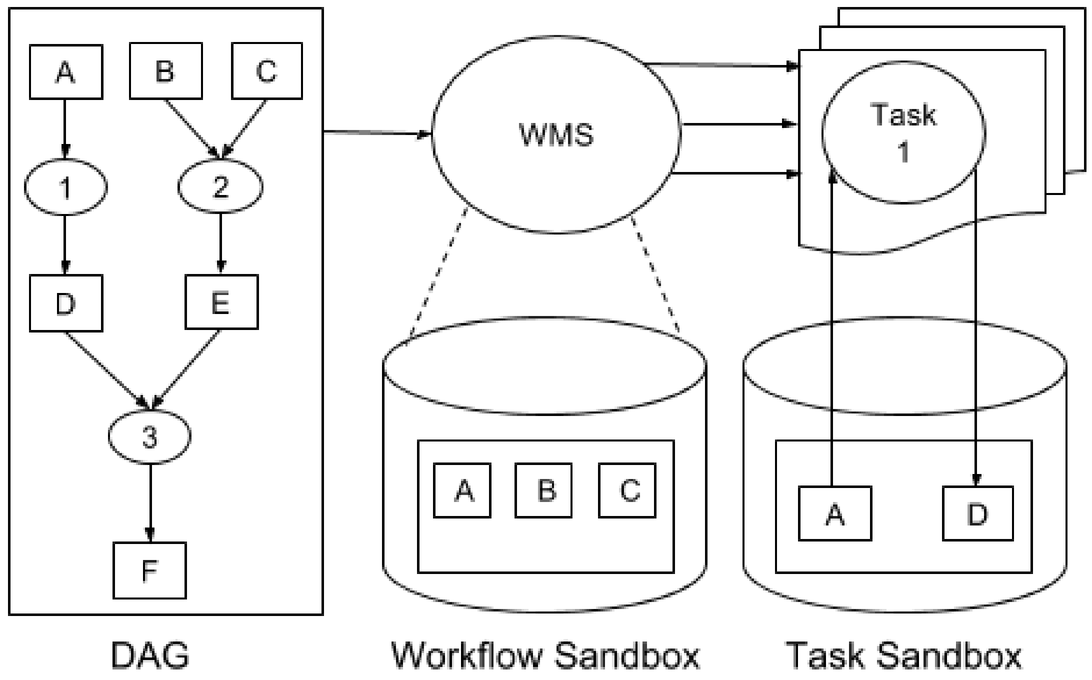


Figure 4.1. File relationship between workflow and task. In the DAG, files are represented as squares and programs as circles. When a node is submitted, the input files are sent to the task sandbox, where upon completion the output files are retrieved. These files are held in the workflow sandbox, which is managed by the Workflow Management System.

task has a *task sandbox* in which it executes, which must be large enough to contain the inputs, outputs, and auxiliary files needed for that single task. Likewise, the workflow as a whole has a *workflow sandbox* which must be large enough to contain the inputs, outputs, and intermediates files of the workflow.

Typically, the workflow sandbox is stored in a large parallel filesystem used for user data and home directories, such as Panasas [120], Lustre [93], or Ceph [119]. We assume that this storage is large and fast, but of finite capacity. There are two common configurations for the task sandbox. Some systems have storage local to each node, so the task's storage consumption is different from the workflow. Some systems do not have storage local to each node, and so the task's sandbox exists in the global filesystem and must be accounted against the workflow's storage. In either case, each task's consumption must be allocated and measured.

4.3 Storage Management Components

In the context of workflow storage management, I make two contributions:

1 - Static analysis of the storage footprint. I present an algorithm which statically determines the storage footprint of a DAG before execution. The algorithm uses a single-pass, bottom-up approach to determine the storage needs for the execution of each task. As the algorithm traverses up the tree it accumulates values that correspond to the storage needs of the task's descendants in a global context. Once the traversal is complete, the set of head nodes is used to determine the overall needs of the DAG as well as inform concurrent branches on the loose ordering to maintain limits. The algorithm makes a single pass through the graph, limiting the cost of analysis. Utilizing this analysis to prevent deadlock, as opposed to a scheduling algorithm, limits the runtime costs while still providing a precise minimum and maximum data footprint.

2 - Online management of the storage footprint. The dynamic management of the DAG is performed in two ways: storage allocation and task dispatch. The storage allocation works as a transaction hierarchy. When a task is committed, space is reserved for descendants as well. When a node is selected to run, the data footprint from static analysis provides a limit and the length of that commitment. When a task completes, the appropriate files are deleted, and the committed storage is updated. The task dispatch component works in conjunction with the storage allocation, by using the commitment hierarchy and the eligible node's footprint to determine if there is enough available space. If so, the node's footprint is committed and the node is submitted for execution.

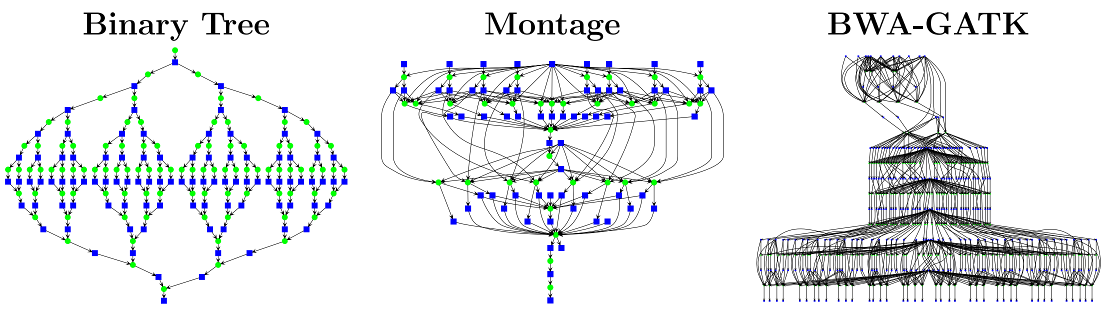


Figure 4.2. Diagram show three example workflows. Three example workflows used to evaluate storage management strategies. **Binary Tree** is a synthetic workflow which generates 1GB files in a tree structure, resulting in a large amount of intermediate data. **Montage** is a widely used workflow benchmark which produces image mosaics from raw astronomical images. **BWA-GATK** is a bioinformatics workflow that performs alignment and genotyping of sequences related to the oak tree. Each graph shown is reduced in cardinality from the actual workflow in order to more clearly show the general structure.

4.4 The Storage Footprint

I begin static analysis by defining the storage footprint of a workflow and computing the footprint manually for several simple examples. This will serve as a basis for the static algorithm in the next section.

For any given workflow, the **absolute maximum** storage it can consume occurs when all files (input, intermediate, and output) exist simultaneously. If the target storage system has enough space for the sum of all files mentioned in the workflow, then it is not necessary to delete any files during execution, and there is no storage management problem.

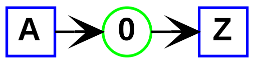
If storage space is limited, then I may delete intermediate files incrementally, whenever they have been consumed and are no longer needed by any node in the workflow. I define the **storage footprint** of the workflow to be the maximum amount of storage consumed during execution with the policy that all files are deleted at the first possible opportunity.

Footprint is affected by the concurrency used to execute the workflow. I define two extreme values of the footprint:

- **Maximum storage footprint** is the largest possible footprint achieved when tasks run with maximum possible concurrency, subject to the workflow ordering constraints.
- **Minimum storage footprint** is the smallest possible footprint, achieved when only one workflow branch is executed at a time, and possibly concurrent tasks are executed in the order that minimizes the footprint.

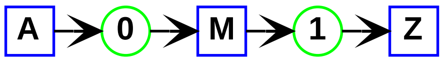
By computing these values *before* executing the workflow, the algorithm can give the user a realistic assessment of the likelihood of success. If the available storage is less than the minimum footprint, the workflow cannot run to completion *at all*, so the end user is advised to look for another system or acquire more storage. If the available storage is between the minimum and maximum footprint, the workflow can complete but concurrency must be limited dynamically. If the available storage is at or above the maximum footprint, then the workflow can run at maximum concurrency if files are deleted at the first opportunity.

Here are the footprints of a few simple workflows:

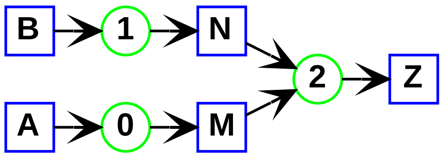


Example 1: A single task 0 reads an input file A and produces an output file Z. The size of a file is defined as $|X|$, where X is the file. At some point during the execution (however briefly) both A and Z must exist simultaneously, so the footprint of the workflow is the sum of the size of the files $|A| + |Z|$ which is abbreviated as $|AZ|$. After the task completes, A may be deleted, but Z remains, so the residual

file of the workflow is Z.



Example 2: Two tasks execute in sequence. Intermediate file M is then created by executing task 0 with input A, after which file A is no longer needed and can be deleted. Next, output file Z is created by executing task 1 with input M, at which point file M can also be removed. This results in only output file Z remaining in the end. A and M must exist simultaneously, and M and Z must exist simultaneously, so the footprint is the $\max(|AM|, |MZ|)$, and the residual is the sole output Z.



Example 3: Two tasks combine outputs in a third. This workflow can execute three different ways:

- Case 1: If tasks 0 and 1 can execute simultaneously, then files A, B, M, and N must co-exist. Files A and B can be deleted, at which point M, N, and Z must co-exist. The footprint is $\max(|ABMN|, |MNZ|)$.
- Case 2: If task 0 executes first, then files A, B, and M co-exist, after which file A can be deleted. Then, task 1 executes, so files B, M, and N co-exist. File B can now be deleted. Finally, task 2 executes, so M, N, and Z co-exist. The footprint is the largest of the three steps: $\max(|ABM|, |BMN|, |MNZ|)$
- Case 3: If task 1 executes first, the combinations are similar to Case 2:

$\max(|ABN|, |AMN|, |MNZ|)$

As can be seen, the maximum storage footprint occurs in case 1, while the minimum storage footprint is the minimum between cases 2 and 3. This presents the runtime manager of a workflow with a trade off – increased concurrency can result in increased storage consumption. If this is not carefully quantified, storage may be accidentally exhausted.

4.5 Example Workflows

Figure 4.2 shows three workflows that are used as running examples. Each of these workflows can be generated at various scales; the figure shows a low-degree example to make the macro-structure clear. Each presents a somewhat different storage management challenge; I can give a qualitative sense of the footprint by examining the workflow structure.

The **Binary Tree** workflow is a synthetic benchmark consisting of processes that each consume and produce a single file of 1MB. Each file is consumed by two children until the desired depth d, and then the data is reduced to a single file in a similar manner. If all branches are executed concurrently, the maximum footprint is $2^d + 2^{d-1}$, when two levels co-exist at once. The minimum footprint is $d + 2$, when a single branch plus one task must exist simultaneously. Many values in between may occur if the workflow proceeds unevenly. The footprint tends to peak in the middle of execution.

The **Montage** [68] workflow computes mosaics of astronomical images, and is widely used as a benchmark for evaluating workflows. It can be generated at a variety of scales by varying the angle of sky (and thus number of images) to be processed. Although Montage has been previously used to explore storage consumption [49], it is an unusual workflow in that most intermediate files it creates are used by multiple later stages, such that most files cannot be deleted until the final chain of individual jobs. Thus, the footprint tends to increase slowly until reaching a peak near the end of the workflow.

The **BWA-GATK** [58] workflow combines two common bioinformatics tools into a large scale parallel application. A large genomic query file is split into task-sized pieces, and the BWA alignment tool aligns the queries to a reference dataset. Then, the GATK tool uses a Bayesian algorithm to compute the quality of successful alignments. The workflow size is increased by adding more data from multiple organisms. This workflow has an irregular footprint over time: each stage of the workflow produces files which are used only once and then may be deleted. In some cases, single files are consumed by single tasks in parallel, so the footprint changes in a fine-grained manner. In other cases, multiple files must be consumed by all tasks in a stage, creating a storage barrier in which nothing is deleted until all tasks complete.

These three workflows are not intended to present a complete profile of workflow behavior, rather show common behaviors which may result in deadlock given certain conditions. The static and dynamic algorithms I present are suitable for running these workflows in most distributed computing environments. In this chapter I utilized a centralized batch system due to its wide-spread use, but the work I present does not rely on a batch system environment. Further, I do not make any assumptions about typical file sizes, workflow behavior, or workflow needs. I do assume the user can accurately estimate characteristics of their workflow to account for any special needs or storage behavior it may experience during execution.

4.6 Static Analysis Algorithm

The algorithm used to analyze the abstract DAG utilizes a single pass bottom-up approach for determining the estimated storage utilization. The algorithm is presented in Algorithm 1. The goal of the algorithm is to determine the storage using information gathered at each node and passed upward.

Algorithm 1 Algorithm to Measure Storage Footprint.

Term	**Definition**
n	Current node under examination
n.descendants	Nodes that utilize a file produced by n
n.children	Nodes related to n where no other descendant of n is parent, subset of descendants
n.residual_nodes	List of nodes where all children are used
n.residual_files	Files held until nearest residual node
n.diff	Difference in size between n.min_footprint and n.residual_files
n.run_footprint	Files used/created during n's execution
n.diff_order_footprint	Files forming largest footprint during diff order traversal
n.wgt_order_footprint	Files forming largest footprint during wgt order traversal
n.min_desc_footprint	Files forming minimal space needed to execute to next residual
n.min_footprint	Files forming minimal space needed to execute self and children
n.max_desc_footprint	Files forming maximal space needed to execute to next residual
n.max_footprint	Files forming maximal space needed to execute self and children

```
procedure MeasureStorageFootprint( n )
for all c in n.children do
    MeasureFootprint(c)
end for
$\text{n.residual\_nodes} \leftarrow \text{n} + (\bigcap_{c}^{\text{n.children}} \text{c.residual\_nodes})$
if $|\text{n.children}| < 2$ then
    $\text{n.residual\_files} \leftarrow \text{n.outputs}$
else
    $\text{n.residual\_files} \leftarrow \max(\text{n.outputs}, \bigcup_{c}^{\text{n.children}} \text{c.residual\_files})$
end if
$\text{n.run\_footprint} \leftarrow \text{n.inputs} + \text{n.outputs}$
n.diff_order_footprint
$\text{tmp footprint} \leftarrow \text{n.outputs}$
for all c in n.children sorted by c.diff do
    if $|\text{footprint}| + |\text{c.min\_footprint}| > \text{n.diff\_order\_footprint}$ then
        $\text{n.diff\_order\_footprint} \leftarrow \text{footprint} + \text{c.min\_footprint}$
    end if
    $\text{tmp footprint} \leftarrow \text{footprint} + \text{c.residual\_files}$
end for
$\text{n.wgt\_order\_footprint} \leftarrow \max_{c}^{\text{n.children}}(\text{c.min\_footprint}) + \sum_{r}^{\text{n.children - c}} (\text{r.residual})$
$\text{n.min\_desc\_footprint} \leftarrow \min(\text{n.diff\_order\_footprint}, \text{n.wgt\_order\_footprint})$
$\text{n.min\_footprint} \leftarrow \max(\text{n.parent\_footprint}, \text{n.min\_desc\_footprint})$
$\text{n.max\_desc\_footprint} \leftarrow \bigcup_{c}^{\text{n.children}} (\text{c.max\_footprint})$
$\text{n.max\_footprint} \leftarrow \max(\text{n.parent\_footprint}, \text{n.max\_desc\_footprint})$
$\text{n.diff} \leftarrow |\text{n.min\_footprint}| - |\text{n.residual\_files}|$
```

A residual node is defined as any node that is the lone child of another node or nodes. This is useful as it provides a limit on the look-ahead needed to accurately determine storage needs of the parent node. If a node's children culminate in a single node the storage impact of these nodes is limited between the node and the residual node where they culminate. The storage for the node's children can be calculated and saved at the node for future use. The value at the node now represents the space needs to be committed for guaranteed execution.

Traversal begins from leaf nodes, where the residual nodes and files include itself and its outputs. The only relevant footprint at this location is the run footprint, which is also the minimum and maximum footprints. As I traverse up the DAG, there are two states in which a node resides. A node in the first case has only a single child node, at which point the residual nodes, files, and run footprint are equivalent to the child's residual nodes, files, and run footprint respectively. As there is only one possible ordering for child execution, the minimum and maximum footprints are defined as the minimum and maximum respectively, between the node and its child's footprint.

In the case where there are multiple nodes, the ordering can affect the overall footprint. To evaluate the interplay between children I find the common subset of residual nodes shared by each child. This becomes the residual node set for the current node. Of the remaining uncommon residual nodes, the residual files and the largest footprint are found for each child. The node's residual file set is the union of all children's residual files. I evaluated two methods of traversal, each with its own goal. The straightforward approach is to find the largest footprint among the children and add the remaining children's residual files. The other approach is to order the children by the difference between their minimum footprint and residual files. Traversing the nodes in this order favors nodes that consume and release space instead of nodes that hold their consumed space. To pick between which is better the minimum between the two is selected as they both account for executing all of the nodes. The minimum footprint is determined by utilizing the largest of the current nodes footprint and the footprints reported by its children. The maximum is the sum of all the children nodes' maximum footprints.

Figure 4.3 and Table 4.1 shows the algorithm applied to a simple workflow. Assuming each file is of size 1, the minimum footprint (ACLMN) occurs if the bottom-most branch of the workflow is executed first. Files LMN exist simultaneously, before being reduced to W, allowing the rest of the workflow to proceed. The maximum footprint (ACDLMNX) occurs when all three branches execute concurrently, so that DLMNX all must exist at once, along with the input files (AC) to the tasks that created them.

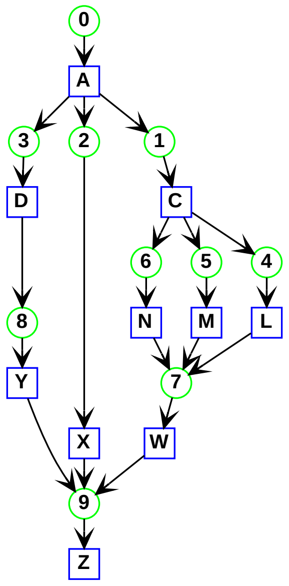


Figure 4.3. Worked Storage Example DAG

TABLE 4.1

WORKED STORAGE EXAMPLE VARIABLES

Node (Res Nodes)	Min Footprint	Max Files	Residual Footprint	Run Footprint
9 {9}	4 {WXYZ}	4 {WXYZ}	1 {Z}	4 {WXYZ}
8 {9,8}	2 {DY}	2 {DY}	1 {Y}	2 {DY}
7 {9,7}	4 {LMNW}	4 {LMNW}	1 {W}	4 {LMNW}
6 {9,7,6}	2 {CN}	2 {CN}	1 {N}	2 {CN}
5 {9,7,5}	2 {CM}	2 {CM}	1 {M}	2 {CM}
4 {9,7,4}	2 {CL}	2 {CL}	1 {L}	2 {CL}
3 {9,8,3}	2 {AD}	2 {AD}	1 {D}	2 {AD}
2 {9,2}	2 {AX}	2 {AX}	1 {X}	2 {AX}
1 {9,7,1}	4 {CLMN}	4 {CLMN}	3 {LMN}	2 {AC}
0 {9,0}	4 {AWXY}	7 {ACDLMNX}	3 {WXY}	1 {A}

The variables in this table correspond to the worked example in Figure 4.3. This worked example is further explored in Section 4.7.2.

TABLE 4.2

STATIC ANALYSIS RESULTS

Size	**Min (GB)**	**Max (GB)**	**Abs (GB)**	**Tasks**	**Analysis Time (S)**
			Binary Tree		
3	5	12	22	22	0.0015
5	**7**	**48**	**94**	**94**	**0.0065**
10	12	1536	3070	3070	0.2776
15	17	49152	98302	98302	10.631
			Montage		
0.01	0.07	0.12	0.13	35	0.0041
1	1.87	2.78	2.98	998	1.9558
1.99	**4.01**	**6.49**	**9.33**	**2984**	**2.3500**
			BWA-GATK		
1	2.69	2.69	3.614	53	0.0117
5	2.75	13.46	17.99	265	0.0439
10	**2.82**	**26.93**	**35.94**	**530**	**0.0749**
25	3.06	67.31	89.83	1325	0.1659
50	3.43	134.63	179.64	2650	0.3082
100	4.19	269.25	359.24	5300	0.4145
500	10.25	1346.26	1796.11	26500	2.8163

Each section of this table refers to one of the three workflows used in our analysis. The size column refers to different attributes for each workflow. Size refers to the number of split levels in Binary Tree, the degree of the sky being analyzed in Montage, and the number of individual samples included in BWA-GATK. Min shows the estimated minimum footprint, Max the estimated maximum footprint, and Abs the sum of all files in the workflow. Tasks are the number of tasks created in Makeflow. Analysis Time shows the additional time needed to determine the footprint for each case.

Table 4.2 shows the results of applying this algorithm to the three example workflows previously described, as implemented in Makeflow [2]. For various sizes of each workflow, I compute the minimum footprint, maximum footprint, and absolute maximum. The size of the three workflows are given as the tree depth for Binary Tree, the degrees of resolution for Montage, and the number of organisms for BWA-GATK. The bold lines indicate the configurations actually run below. As discussed above, the maximum footprint of Binary Tree grows exponentially, the maximum footprint of Montage is close to the minimum footprint, and the maximum footprint of BWA-GATK is roughly linear with the width of the workflow. With the exception of the very large binary tree, the single-pass algorithm executes in a mater of seconds.

4.6.1 Limitations

This static analysis perform well in an organized consistent environment, but there are several factors that affect execution. The first is untracked files. Often in execution programs create log files or auxiliary status files. These files have limited scientific worth outside of performance and error logs, but occupy no space. If they are specified in the task then I account for them and remove them when no longer needed. When they are not specified they clutter space and are never cleaned causing more contention. Second when files vary significantly from expected size, such as log files, the static algorithm does not recompute to account for this. Ideally, I recompute and reallocate using the dynamic management, but if the limit is already close it may be beyond the point where a change will help.

To combat these issues, I have additional mechanisms to help determine if the environment is prohibiting forward progress outside of the bounds of our management. This comes up by actively cleaning old files and watching the working directory. If the filesystem report almost full utilization of storage error messages are printed to bring the users attention to the issue, though the static analysis does little to help prevent this.

4.7 Dynamic Storage Management

In the previous section, the static allocation defines the storage bounds of the workflow execution. The dynamic algorithm utilizes the static analysis results to enforce the space reservations needed for future execution.

The dynamic storage allocator uses a data structure that tracks the current space utilized by files and reservations of current and future nodes. The data structure is initialized with a base size, called the base allocation, which is specified by the end user with the help of the static analysis results. The base allocation defines the upper limit of available space for this execution of the workflow. Within the base allocation, reservations are created to hold space for a node and its ancestors. This creates a hierarchy in the data structure of node reservations above their descendant reservations. When a file is created it is accounted for in the current reservation and tracked until deletion.

4.7.1 Dynamic Storage Algorithm

The dynamic algorithm consists of three stages for each node's execution: verification, allocation, and release.

When a node is logically ready for execution, the dynamic storage allocator checks if there is sufficient space to run the selected node. The allocator utilizes the residual node set to compare against the existing data structure. The allocator starts with the lowest residual node (nodes later in the workflow) and compares the data structure with the required space for the residual node. If the residual node reservation does not exist in the data structure, the allocator checks for sufficient space in the base allocation. If the reservation exists, but there is not sufficient space, the allocator checks if the existing hierarchy can be enlarged to reserve the additional size. If there is sufficient space in the reservation hierarchy, the allocator continues up the residual list using the available space to check nodes. If there is not sufficient space, the node is postponed for submission.

If the verification step is successful, the node is allocated and submitted. To allocate a node, its residual nodes are passed to the allocator. The allocator creates a reservation for each residual node that does not exist in the data structure and grows smaller allocations to accommodate. This proceeds for all residual nodes until a hierarchy exists above the base allocation. In this hierarchy, any lower node is at least as large as the nodes above it. If multiple nodes share a common residual node, the common node is at least as large as the sum of the higher nodes. In this way, space is reserved for the widest part of the ancestors to guarantee space.

After execution, the allocator must release the completed reservations. During this release stage, files that are no longer needed are deleted and their space is marked as available in the data structure. The reservation for the completed node is released, and files within the reservation are transferred to the containing reservation. These files continue to exist until they are no longer needed. If output files are larger than the existing reservation, the reservation attempts to grow and accommodate the increased size. This can cause deadlock or failure from resource exhaustion. Due to these changes, the static analysis is outdated and can no longer guarantee execution.

4.7.2 Worked Example

Figure 4.4 demonstrates the dynamic algorithm using the earlier worked example (Figure 4.3 and Table 4.1).

Step 1: With Node 0 available to run, the algorithm checks the stack to ensure there is enough space. Traversing the residual nodes of Node 0, Node 9 is added to the stack with the size needed for the nodes between 0 and 9. Node 0 is then reserved on top of Node 9. Once executed, File A moves from the space reserved by Node 0

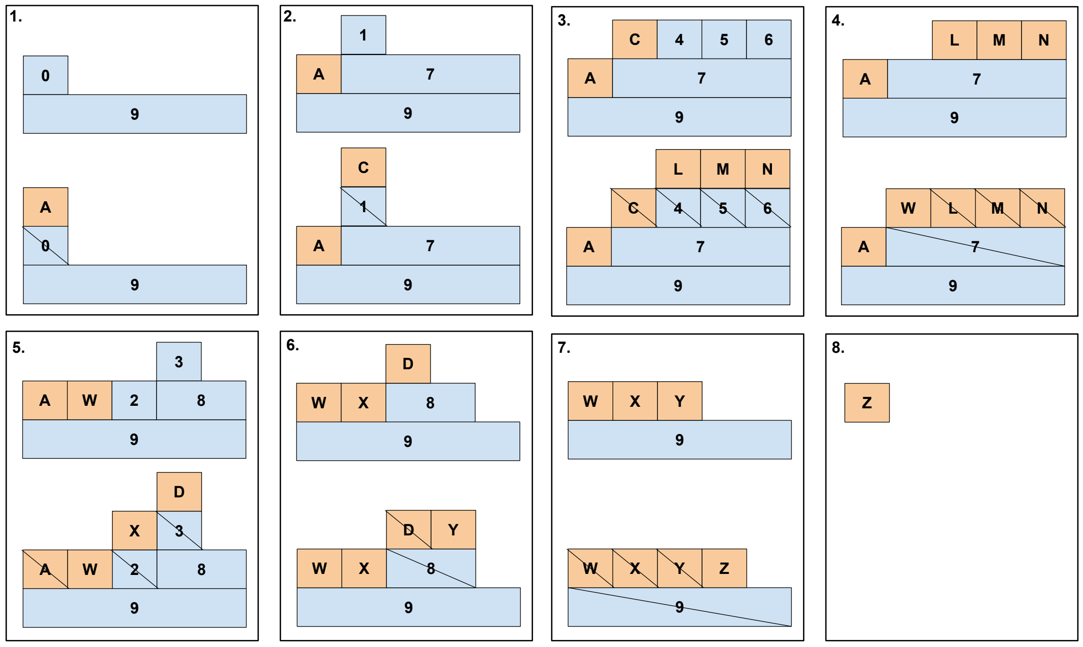


Figure 4.4. Diagram showing dynamic allocation data structure. Shown is the execution of the worked example from Figure 4.3. Numbered boxes represent storage allocations for nodes. Lettered Boxes represent allocations for specific files. Boxes with slashes through them are allocations which are being removed because either the node is complete or the file is no longer needed. Allocations are removed when no longer relevant.

to Node 9, and node 0's reservation is removed.

Step 2: With File A existing, Nodes 1, 2, and 3 are available to run. The algorithm will check Node 1. Traversing Node 1's residual nodes, Node 9 is reserved. With the remaining space in Node 9, the algorithm can fit Node 7 and Node 1. A reservation is then added to 7 above 9, and a reservation for Node 1 is created on Node 7. After completing Node 1, File C is created and Node 1's reservation is removed.

Step 3: The available nodes to run are Nodes 2, 3, 4, 5, and 6. There is not enough space for Node 2 and 3. However, in the reservation for Node 7, there is space for Nodes 4, 5, and 6. Space for Nodes 4, 5, and 6 is reserved. Upon completion, files L, M, and N are created and the node reservations are released. File C is no longer needed and is removed.

Step 4: With Nodes 2, 3, and 7 available, again we check which nodes will fit. The reservation for node 7 already exists, so Node 7 is executed. After outputting file W, files L, M, and N are removed along with node 7's reservation.

Step 5: Space now exists for Node 2 and 3, so reservations are created for Nodes 2, 3, and 8. After running, file A is removed, and reservations 2 and 3 are released.

Step 6: Reservation for Node 8 exists. File Y is created. File D is removed, and Node 8's reservation is released.

Step 7: Reservation for Node 9 exists. File Z is created. File W, X, and Y are removed. Node 9's reservation is released.

Step 8: Final output File Z exists. The workflow is complete.

4.7.3 Impact of Local Storage

Local storage can affect the dynamic storage management in several ways. First, if the local storage that is utilized for execution is accounted for within the same quota, then the dynamic management needs to adjust. In this case, I need to account for both the active space utilized during execution and the space consumed in caches are used for data movement. In the prior case, when a node is allocated the requisite space needed for execution is added to the allocation, but removed after execution. Caches are more difficult as they are essentially copies of the existing data and may persist between execution increasing the complexity. One method for handling this is to set a limit on individual size of a cache such that old files are removed and the space is statically accounted for.

Second, in cases where temporary space is unaccounted for, such as in scratch spaces or local temporary directories, I do not need to account for space. These resources can still become contentious. In these cases I rely on the remote execution to report on limited space to maintain limits. Unfortunately, system wide storage contention on scratch storage is not in the purview of this chapter.

Currently, I do not consider either as our remote execution happens on local temporary space outside of the quota, though inclusion of these factors may become necessary in some execution environments.

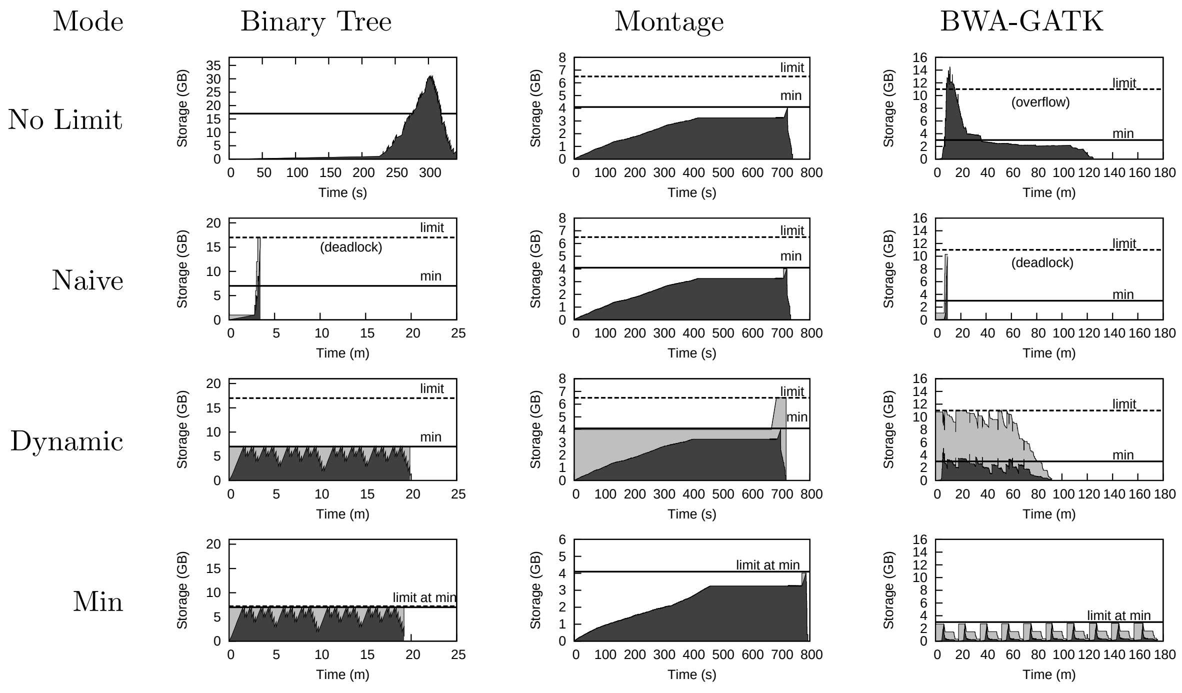


Figure 4.5. Storage limits applied to example workflows. Timeline of storage consumption for each of the three example workflows, in different configurations. The top row shows uncontrolled executions which exceed the desired limit. The second row shows naive storage limits which can result in deadlock. The third row shows a limit applied using the dynamic algorithm. The fourth row shows the minimum storage footprint enforced. In each graph, the dark area shows storage actually consumed, while the light area indicates storage committed to future use.

4.8 Overall Evaluation

Figure 4.5 shows the behavior of the dynamic algorithm implemented in Makeflow. The workflows are evaluated by submitting each task to Work Queue, with up to 100 tasks running simultaneously on remote nodes. For each, I chose a storage footprint limit (dotted line) that is larger than the minimum footprint (solid line) but less than the maximum footprint. The dark shaded area on each graph shows the storage actually consumed, while the light shaded area shows the “committed” storage value.

4.8.1 Configuration

Each workflow was created and run using the emboldened configuration entry in Table 4.2 for each corresponding tool. All three workflows executed using a shared filesystem accessed by batch job workers. Each worker was allocated a single core and at least 8GB of memory. The amount of storage allocated to the worker was based on the the sizes from Table 4.2. The machines used to run the batch job workers were part of a campus-scale cluster and thus varied in their hardware configurations however the method used to acquire batch job workers ensured the worker had access to the requested amount of resources in order to begin work.

Each workflow is run in four different configurations:

The **No Limit** row shows each workflow run with maximum concurrency and no explicit storage limits. The ordering of tasks is solely due to logical constraints, and the exact concurrency achieved depends upon the performance of the tasks in the batch system. As can be seen, each workflow meets or exceeds the minimum footprint, and in the absence of control, exceeds the desired limit or deadlocks.

The **Naive** row shows the workflow system attempting to enforce the desired storage limit by simply examining each file as it is submitted. Storage is committed for the immediate output files of a task when it is submitted. Tasks are only submitted if the committed value can be kept below the limit. This can result in deadlock (as shown) if the workflow manager commits too much space and is unable to execute tasks to consume created files.

We note that both the binary workflow and BWA-GATK experience deadlock using the naive strategy given the storage limit. The binary workflow's rapid expansion behavior caused the naive approach to very quickly exhaust its storage. In BWA-GATK, deadlock occurred when a group of the most data-intensive tasks in the workflow were dispatched at the same time. Some of the data-intensive tasks remained but could not be scheduled because there was not enough storage available.

The **Dynamic** row shows the workflow system using the dynamic algorithm described in this section to limit submission of new tasks. We note that the committed storage increases more rapidly than in the naive case because the footprint of each task is committed before submission. The storage actually consumed does not always rise to the level of the committed storage, which is an upper bound on all possible executions following each node.

The **Min** row shows the workflow system using the dynamic algorithm, but the limit is set to the minimum possible value. As previously mentioned, the Montage Min graph utilizes the file list approach to allocation management.

Overall, we observe that the dynamic online algorithm is able to enforce the desired constraints without falling into deadlock. As noted above, the committed value is an upper bound on actual consumption. Noting this, there still exists room for improvement in the upper bound. The difficulty in optimally utilizing an upper bound is tied closely with the runtime behavior of the workflow. Improvements could be made if the execution time of tasks is combined with the expected growth of a branch in order to overlap branch executions more smoothly. The footprint is currently defined as a means of preventing over-commitment. Optimal storage use focuses more closely upon scheduling as a solution and relies on consistent execution behavior while static footprint analysis relies only on file sizes.

During real execution, we expect the static analysis and dynamic storage algorithms to have broad appeal. The static algorithm can help a user understand the needs and behavior of their workflow better while the dynamic algorithm will help keep the workflow constrained when necessary. The runtime task constraint using loop devices is another way of guaranteeing the user's storage requirements are not exceeded. We believe all three tools have general appeal to scientific workflows regardless of the workflow's behavior or size. The limiting factor of our tools' appeal comes from the user's storage needs.

4.9 Conclusions

I have illustrated two techniques for managing storage space in workflows: a static algorithm for offline analysis of storage consumption; and, a dynamic algorithm which enforces runtime limits while avoiding deadlock. The dynamic algorithm is an upper bound on consumption and can overestimate in cases where the same file is used in multiple places in the workflow. Utilising these methods we are able to see how storage can be estimated and tracked from a full workflow context.

In the larger context, these methods provide a means of managing workflow resources and preventing over use from workflows. Both of these methods can be used to limit the effects of moving between sites and changing data configurations. More importantly, these methods provide a means of abstracting knowledge and control of workflow specific resources to a higher level, such that workflow management systems or batch execution systems could leverage this information to prevent deadlock and failures. Using these techniques with dynamic workflow expansion allows workflows to safely expand and execute on new sites with mechanisms to analyze and manage resources, further increasing the flexibility of the workflow.

Using the dynamic expansion and resource management methods, we can quickly move to new sites with larger data. However, as we change the execution assumptions it is unavoidable that we encounter a site with a different environment than was originally tested. When these differences diverge more dramatically from the original implementation, methods are needed to adapt workflows to new sites. I will next describe an algebra to adapt workflows to different environments, further increasing workflow flexibility and portability.

CHAPTER 5

AN ALGEBRA FOR ROBUST WORKFLOW TRANSFORMATIONS

5.1 Introduction

In the prior chapters I explored methods to dynamically creating and managing workflows. These methods allow for workflows to be generated as needed for different datasets on previously unused sites. These methods provide a robust and flexible solution to creating workflows and managing their resource needs but relied on the site to provide a compatible execution environment. Differences between execution sites makes porting workflows difficult and debugging complex. It is common for workflows to assume: libraries and programs are available; use applications configured for a single operating system; or, rely on unspecified configurations; all of which may cause workflows to fail on different sites. Accommodating each site's configuration requires a number of unique transformations to properly execute a workflow. The tasks themselves do not change, but the environment, error handling, and configuration may.

A typical use case is the need to deploy the same operating system and software stack on several available compute sites. Unfortunately, each site may have a unique operating system or lack the necessary software requirements. Users need a way to quickly switch between each site without rewriting the workflow for each site. A possible solution is to use containers. But how does a user easily apply these containers to tasks? This is further complicated when each site may use different container technologies (i.e. Docker[89] and Singularity[74]).

The ability to combine available tools is required to handle unique configurations and environments. As the number and variety of tools increases, the complexity of combining them increases as well. For example, if Singularity and a custom script are both applied to a simple task by prepending their commands, characteristics of execution like exit status, provenance of files, and the final executed command become opaque. Properly nesting the container inside of a script allows for differentiating failures, debugging, and consistent execution. Each additional layer must become a more nuanced transformation as nesting technologies, such as containers, resource monitoring, and error handling, becomes necessary. Different combinations of tools are required depending on the site's unique configuration. The variable nature of required tools indicates the importance of only applying tools to a workflow as needed, rather than adding them to the workflow specification at each site.

To address this problem, I define an algebra for workflow transformations to address the complexity of nesting different tools and technologies. Based on the sandbox model of execution, this algebra formalizes the operations for applying transformations to tasks, producing new tasks. These transformations can then be applied in series to produce a task that incorporates all applied transformations. Using formalized task transformations, I are able to precisely apply multiple transformations to a workflow and cleanly map to each task.

This algebra was expressed using JSON so that it is independent of (and therefore portable to) a variety of systems. Using this JSON expression, a driver was written in Makeflow[2] that allows us to apply transformations to a full workflow. I will discuss the challenges in applying transformations and how these methods can be applied incorrectly and incompletely. To show the efficacy of this solution I show several case studies. The first uses a Singularity container to: provide consistent environments; a resource monitor to give accurate usage stats; and, a sandbox to isolate the available files and workspace. The second shows a failure handler that captures a core dump and converts it into a stacktrace, streamlining analysis and lower data transfer. The final case study executes the same workflow on several sites using an environment builder that dynamically builds required software at each task.

5.2 Challenges in Transforming Workflows

A workflow primarily describes the researcher's work to run a set of simulations, to analyze a dataset, to produce a visualization, etc. However, like any kind of program, there may be a number of secondary requirements that must be met to complete the work: a particular software environment should to be constructed, resource controls for the batch system will be selected, monitoring and debugging tools should be applied to the task, and so forth. This might involve setting environment variables, providing additional inputs, capturing additional outputs, invoking helper processes, and more.

The first version of a workflow, constructed at a particular computing site, may have all of these aspects intertwined with the definition of the tasks to be done. The application may depend upon software environments installed in fixed paths in a shared filesystem. Environment controls may be set within individual tasks. Resources may be hard coded for a particular batch system The graph structure may reflect the current set of debugging tools enabled. While this may work well at the first site, it may become necessary to move the workflow to another site in order to improve performance, increase scale, or to apply the workflow in a new context. All these site-specific controls are unlikely to work in the new context, and the receiving user is then stuck with the problem of disentangling the core code from the local peculiarities.

An appealing approach to this problem is to define simple modifications that can be individually applied to tasks (transformations) in order to achieve specific local effects. For example, one might have a transformation to run a task in a container

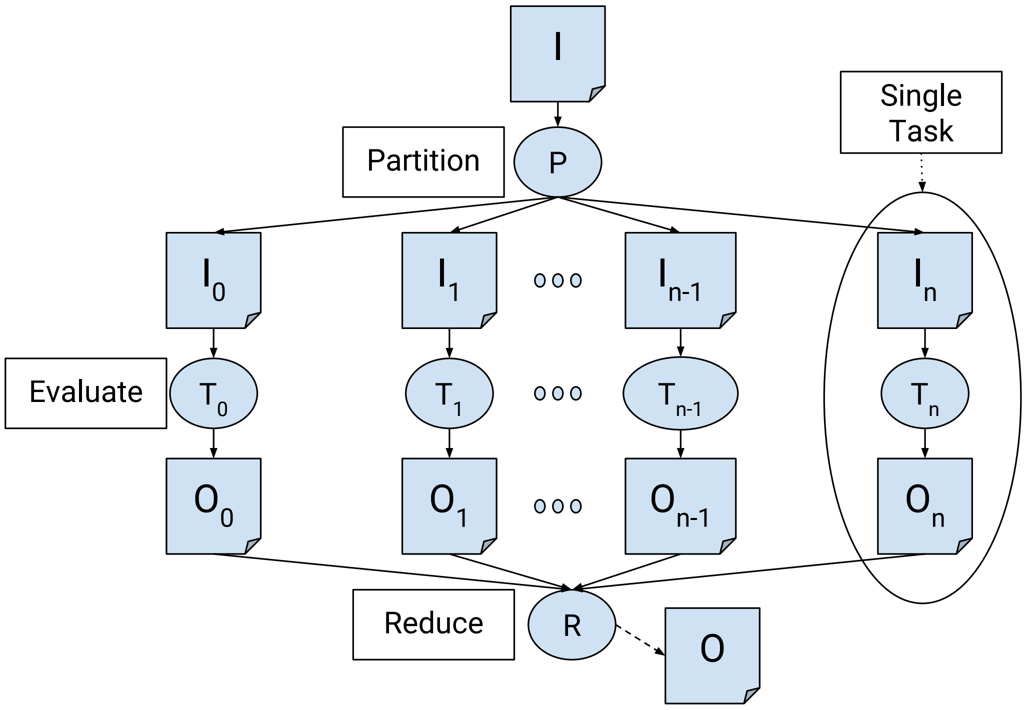


Figure 5.1. Diagram showing an example workflow. This basic workflow shows standard split-join behavior. The first task partitions the work, the next set of tasks analyze the individual partitions, and the last task joins them all back together. Each task executes independently from each other and are often run on batch execution systems.

environment, another transformation to perform monitoring and troubleshooting, and a final transformation to configure a software environment for the local site. With this approach, the scientific objective of the workflow can be expressed in a portable way. A set of external transformations are used to modify the tasks as needed for the local site. Porting a workflow from one site to another becomes the simple job of adjusting a few transformations rather than rewriting the workflow from scratch. If it is necessary to transform the workflow in a new way, a new transformation can be written and shared with others so that it can be applied to many workflows.

However, our experience is that designing and using transformations is not so easily done. What may seem like a simple and obvious transformation can end up creating complex interactions and incorrect results. As a simple example, suppose that I want to run each task inside a Singularity container named `centos.img`. At first, this sounds as simple as prepending `singularity run centos.img` to each command string then running the task. While this works in limited cases, the general case for workflows with complex task definitions fails. There are several reasons for this:

Substitution semantics. Using basic string substitution to embed one command inside another often complicates execution. Commands that use input/output redirection, consume files, or change the environment collide using basic substitution. Addressing this uncertainty with shell quoting only further complicates the matter, and may change the execution.

Workflow modifications. Applying a transformation to a task not only changes the individual task, but may also have an effect on the global structure of the workflow. A command transformed by a container now has an additional input (`i.e. centos.img`) which must be accounted for as a dependency in the workflow. Container images are large and affect the scheduling and resource management of the workflow. In a similar way, the container produces additional outputs which must be collected and managed by the workflow.

Namespace conflicts. Transformations can modify the local filesystem namespace. Log files with fixed names, temporary generated files, files based on task input files, or modifications to the working directory all alter the task and the workflow. These actions blindly modify files outside the workflow or cause race conditions with other concurrent transformations. Since it is not always possible to alter these hard-coded paths into unique filenames, collisions are inevitable.

Troubleshooting complications. The exit semantics of a transformed task are complex as it is not sufficient for a transformation to simply return the task's integer exit status. Each exit status should be differentiated, as transformations may fail separately. For example, preparing the environment may fail because a necessary software dependency is not present, a container may fail when pulling the container image over the network, or a resource monitor may exit when resources are exhausted. In each of these cases, the user must have a means of distinguishing between *transformation failure* and *task failure*. When multiple transformations are applied, the result of the task looks more like a stack trace than a single integer.

To address these challenges, I need a more rigorous way of defining tasks and the transformations on those tasks such that any valid transformation applied to any valid task gives the expected result in a way that can be nested. In short, I need an algebra of workflow transformations in order to make scientific workflows more robust, portable, and usable.

5.3 An Algebra of Workflow Transformations

I designed a formal abstraction to accommodate the execution behavior of various tools. This formalism isolates each transformation for consistent execution, allowing for organized nesting. In particular, our abstraction describes how to define a transformation for a given tool, as well as aspects of execution to consider. Transformations are based on the sandbox model of execution, which describes all aspects of execution for which a transformation is responsible.

5.3.1 Notation

For the purpose of expressing tasks and transformations in a precise way, I use a notation that is based on JavaScript Object Notation (JSON). In addition to the standard JSON elements of atomic values (`true`, `123`, `"hello"`), *dictionaries* `{ name: value }`, and *lists* `[ 10, 20, ... ]`, I add:

- `let X = Y` is used to bind the name `X` to the value `Y`.

- `define F(X) = Y` defines a function `F` that will evaluate to the value `Y` using the bound variable `X`.
- Simple expressions can be built up using standard arithmetic operators and function calls on values and bound variables.
- `eval X` evaluates the expression X and returns its value.

Using this notation, a single task ($T1$) in a workflow is expressed in JSON like this:

```
let T1 = {
 "command": {
   "pre":[ ],
   "cmd": "sim.exe < in.txt > out.txt",
   "post":[ ],
   },
 "inputs"     : [ "sim.exe", "in.txt" ],
 "outputs"    : [ "out.txt" ],
 "environment": {},
 "resources"  :
   {"cores":1, "memory":1G, "disk":10G }
}
```

Figure 5.2. Basic task defined using JSON.

Note that the schema is fixed. Every task consists of a command with a `pre`, `cmd`, and `post` component, a list of input files, a list of output files, a dictionary of environment variables, and a dictionary of necessary resources. Also note that the formal list of inputs and outputs is distinct from the command-line to be executed, as guessing the precise set of files needed from an arbitrary command-line is difficult.

For example, a program might implicitly require a calibration file `calib.dat` and yet not mention that on the command line. The base task's list of inputs and outputs is drawn from the structure of the DAG by the workflow manager.

5.3.2 Semantics

Makeflow allows for tasks in this form to be executed on a wide variety of execution platforms, including traditional batch systems (such as SLURM[69], HTCondor[80], and SGE[45]), cluster container managers, and cloud services. Because each of these systems differ in considerable ways, it is necessary to define precise semantics about the execution of the task and the namespace in which it lives. Once these semantics are established, it becomes possible to write transformations that work correctly regardless of the underlying system. To accommodate these varied systems, I introduce the sandbox model of execution.

The **sandbox model** of execution isolates the environment and limits interactions to only specified files. Isolating the task to run only the specified environment allows for higher flexibility about where the task can run as well as increasing the reproducibility of execution. Limiting the locally available files helps prevent undocumented file usage, enforcing accuracy of the file lists.

Applying a sandbox to a task is a multi-step process for ensuring consistent environment creation:

1. Allocate/ensure appropriate space for execution, based on resources.
2. Create sandbox directory.
3. Link/copy inputs to ensure correct in-sandbox name, based on inputs.
4. Enumerate environment variables based on the specified environment.
5. Run task defined command, using *pre*, *cmd*, and *post*.
6. Move/copy outputs outside of sandbox with appropriate out-sandbox name, based on outputs.

7. Exit and destroy sandbox.

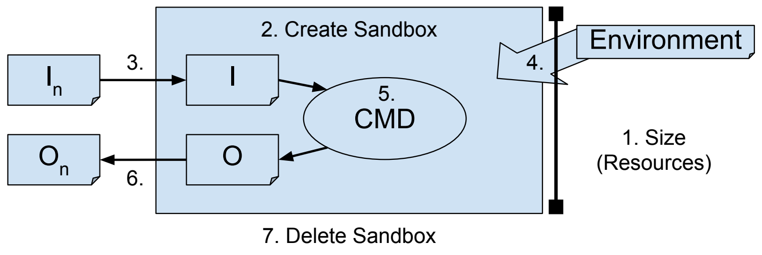


Figure 5.3. Diagram showing the sandbox model of task execution. This shows the different steps needed to isolate the task from the underlying workflow environment to prevent side-effect on the environment and filesystem.

5.3.3 Transformations as Functions

A transformation is an abstraction of a task, and provides the information needed to translate a raw program invocation into a properly defined task. A transformation contains the same fields defined in a task: a command, inputs, outputs, resources, and environment. However, it is an incomplete task with unbound variables that are resolved when applied to a task as a function.

Figure 5.4 illustrates Singularity written as a transformation. As mentioned above, the generic definition of the transformation contains unbound variables such as `T.cmd`, `T.inputs`, and `T.outputs`. When the transformation is applied to a task, those variables are bound from the task's structure. Singularity requires additional space (3G) to account for the Singularity image. Here, resources are not defined as a static value, but in addition the to underlying resource. Additionally, the Singularity

```
define Singularity(T)
{
 "command" : {

   "cmd": "singularity run image " +
           T.script + " > log." + T.ID

 }
 "inputs"   : T.inputs +
              ["image", T.script],
 "outputs"  : T.outputs +
              ["log."+T.ID],

 "resources" : {


   "disk"   : T.resources{disk} + 3G
 }
}
```

Figure 5.4. JSON defining abstract `Singularity` transformation. Describes the Singularity command, added files (such as image and output log), and increases the required disk space. Note, several of the variable are unbound, and will be resolved when applied to a task. Unaltered fields are left undefined.

transformation does not define an environment, so it is left out.

The resulting task of evaluating `Singularity(T1)` can be seen in Figure 5.5. The previously unbound variables have been resolved, such as `T.inputs` becoming `["sim.exe, "in.txt"]`. The values that were not defined or extended by `Singularity` were resolved from the underlying task, such as `cores` and `memory`. Importantly, to create a valid task even empty fields like *pre*, *post*, and *environment* are still specified, allowing for evaluation and additional transformations to be applied.

If you look carefully at Figure 5.4 you will notice two variables not bound by

```
eval Singularity(T1) yields
{
 "command": {
   "pre":[ ],
   "cmd": "singularity run image " +
          "t_ID.sh > log.ID"
   "post":[ ],
 },
 "inputs"   : ["sim.exe", "in.txt",
               "image", "t_ID.sh" ],
 "outputs"  : ["out.txt",
               "log.ID"],
 "environment" : {}
 "resources" : {
   "cores"  : 1,
   "memory" : 1G,
   "disk"   : 13G,
  }
}
```

Figure 5.5. Resulting task of applying `Singularity` to `T1`. The transformed task has all of the variables bound. The file lists have combined the previously defined files with the files added by `Singularity`. The resources are resolved and the required values account for the original task and the transformation.

the underlying task directly, `T.script` and `T.ID`. As part of the abstraction, the underlying task is emitted as a script that is called in place of the command, creating `T.script`. The ability to treat transformations as functions is achieved by isolating each transformation as a separate process. Isolating a transformation provides several key benefits: clearly defined ordering of transformations, instantiated environments persist only in that process and its children, and exit status can be attributed at each level to track failures. In practice this is achieved by producing a script that defines the task, as seen in Figure 5.6.

The second variable, `T.ID`, is key to this method's success, as the ability to uniquely identify each task provides a clear mapping to the workflow. A unique identifier is created using the checksum of the current task, which incorporates the command, input files' names and contents, output files' names, environment, and resources. This identifier is used to identify the output script and can be used by the transformation to uniquely identify files in the workflow. Additionally, as applying a transformation produces a new task, the identifier is updated after each transformation.

```
#!\bin\sh
#ID TASK_CHECKSUM

# POST function
POST(){
  # Store exit code for use in analysis.
  EXIT=$?

  # Run post commands.

  # Exit with stored EXIT which may
  # have been updated by post.
  exit $EXIT
}
# Trap on exit and call POST.
trap POST EXIT INT TERM

# Export specified environment.

# Run pre commands.

# Run core command.
sim.exe < in.txt > out.txt
```

Figure 5.6. Script created when evaluating `Singularity(T1)`.

5.3.4 Applying the Sandbox Model

This creation of a script from a task focuses on isolating just the transformation, but relies on finalization of the task sandbox laid out in Section 5.3.2. To consistently apply the sandbox model to a task I define a sandbox procedure to produce a script that creates a sandbox, handles files, and runs the command. This procedure is applied to a task prior to execution to isolate the task to a single sandbox directory.

This begins with creating a unique identifier, based on the task checksum. The identifier is used to create the sandbox and script names used in execution. In the script a `POST` function captures the exit status, executes `post` commands, and returns the outputs. This function is set as a trap to also analyze failures. Next, the sandbox is created and inputs are linked into it. The process changes directories, exports the environment, and runs the `pre` commands. After this the environment is setup, the task `cmd` can run.

5.4 Transformations in Practice

In applying the above algebra, there are design considerations to be made. To maintain the ability to nest several transformations together, it is important to consider the naming conflicts, the importance of differentiating `pre`, `cmd`, and `post`, file management, resource specification, and how the environment of a task is extrapolated.

5.4.1 Composability versus Commutability

An important aspect of this algebra is the ability to reason about how the combinations of different transformations interact and if they can be applied to created a valid task. Using the previously defined application of transformations I find that the set of transformations are composable, but not commutable. These transformation are not commutable because the ordering in which they are applied changes the core evaluation of the task. This is by design, to allow for the differentiation of transformation ordering.

Transformations, in general, are composable. Any transformation can be applied to any task and produce a valid task, with the exception of static name collisions. A static name collision can result when an application uses hard-coded or default names for files, careless naming, or even randomly generated names. Running a single transformation at a time may not cause a collision, but nested transformations and concurrent tasks make collisions inevitable, as is often seen with output logs and files sharing names between tasks.

Naming is resolved at the local level by detecting when applying a transformation creates overlapping names. If collisions are detected, the transformation is not applied and a failure is returned. Though this restricts some combinations, this can be overcome by better understanding the application and using options to produce unique files.

However, if the same restrictions were applied to tasks across the workflow, transformations with static names would be prohibited entirely. As this may be inevitable, static files may be remapped to a unique name in the workflow. As each task is isolated in a sandbox, static files can be renamed when moving to the global namespace using the task identifiers. Remapping of the file relies on a more verbose file specification as a JSON object instead of a string filename. JSON object specification enables the wrapper to specify an *inner_name*, specifying the name inside the sandbox, and the *outer_name*, specifying the name in the workflow context. An example of how this would look with a statically named file can be seen in Figure 5.7 which defines a resource monitor transformation.

```
define RMonitor(T) {
 "command" : [
  "cmd": "rmonitor -- " + T.script
 ]
 "inputs"  : T.inputs + ["rmonitor",
              T.script]
 "outputs" : T.outputs +
             [{"outer_name="summary."+ID,
              "inner_name"="summary"}]
}
```

Figure 5.7. Verbose JSON object file specification. In this example the resource monitor uses a statically name default summary, ”summary”. In this case, the the summary file is statically named, but will collide in the global workflow context. To avert this collision the file is specified with its static inner_name, and a unique outer_name using the task’s ID.

5.4.2 Command Description

Commands express the setup, execution, and post processing of a task. Commands are broken up into three parts, *pre*, *cmd*, and *post* based on the command structure outlined.

Pre is a set of commands that run prior task invocation and setup the task sandbox. This includes setting environment variables, configuring dependencies, and loading modules or software. For example, a Docker transformation would use *pre* to load or pull images.

Post is a set of command that run after task invocation and is used to handle failure by interpreting or masking them, create outputs to prevent batch system failures from missing files, or validate correctness of outputs. *Post* can differentiate docker failing to load an image from task execution failure, allowing more nuanced debugging.

The *cmd* string outlines the context in which the underlying command is invoked.

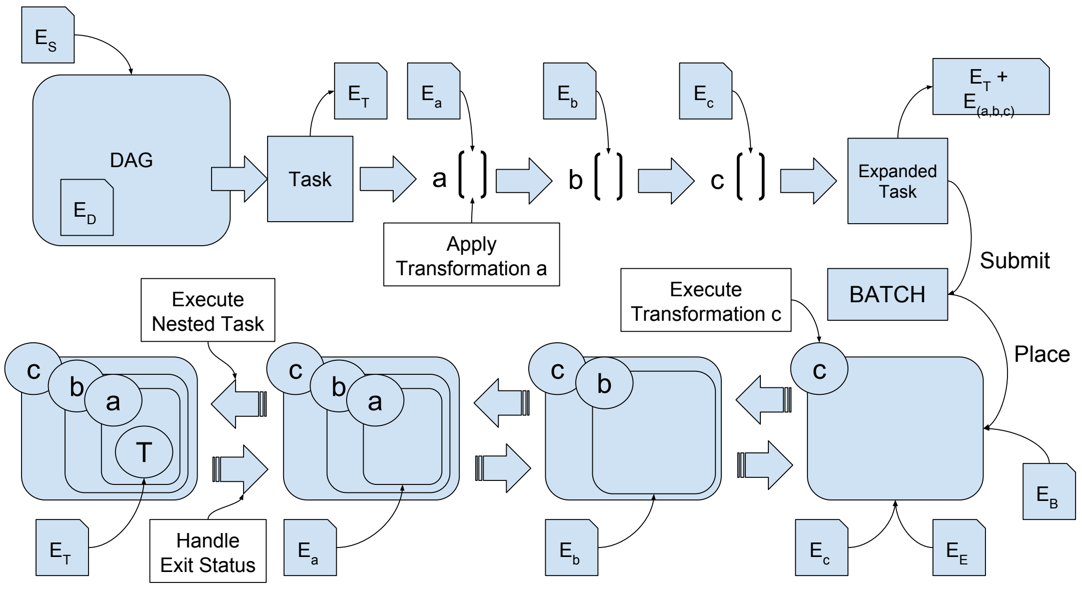


Figure 5.8. General approach to Sandbox model of execution. The environment that exists at task execution is the result of several sources. The environment starts at the DAG where variable are resolved internally and from the host machine. These values define the task's initial environment. Transformations are applied to this task which extend the environment, but are only applied at execution. At the execution site, the environment is defined by the execution node and batch system. As execution starts, each transformation is applied and invokes its environment, limiting their affect to that transformations execution.

Cmd outlines how the underlying task is called and isolates the effects of the calling transformation.

A benefit of separating the command into these parts is that it allows us to differentiate the failures or problems that result from each part. This is useful when determining that the setup of your container failed so the task should not run or to prevent the failure of *post* analysis from indicating a task failure falsely. This separation also allows for each transformation to be clearly expressed in a script, enabling simplified debugging.

5.4.3 File List Management

As transformations are applied, the list of inputs and outputs grows. It is key for the correct organization of transformations that the set of required files is outlined by the task structure allowing the submitting system to confirm required inputs and verify expected outputs. It is possible for a transformation to rename or mask an existing file in the list. By doing so, the transformation changes the context of the task when evaluated. This can be done to allow for redirecting shared files or when using installed reference material. Maintaining a correct set of files helps prevent task collision. This information can also map a *pre* or *post* application onto the files, estimate the space needed for execution, or log these files for later analysis.

5.4.4 Resource Provisioning

The resources define the necessary allocation for proper task execution. This value is extended and augmented by transformations as the context and required resources change. Commonly, as transformations are applied, additional disk space is needed to store new files (like container images).

Resource provisioning may not only be additive as the transformations are applied, but also maximal. This is typically the case used for cores. The number of cores does not expand as transformations are added, but reflects the largest number of cores needed by any transformation. For example MPI utilizes a static number of cores, and to reflect that the resources specification uses the maximum of the provided value and the previous resource specification. The value of the resources required for a task tracks the largest set of each resource. After the transformations have been applied, the final task contains a single specification reflecting the total expected usage.

5.4.5 Environment Elaboration

An important aspect of a task is the environment where the task is executed. The environment defines a variety of values that control things such as available executables, required libraries and values, or even available machines on a cluster execution node. However, the environment is often overlooked or ignored by the researcher, which can cause corruption, errors, and failures. This can be addressed directly on a single site, but as more sites are utilized managing these environments becomes unrealistic. Here I will discuss how the task environment is defined, when transformations are applied, and how the final environment is resolved at execution.

The environment provided to a task varies between sites and evolves as a task is transformed and evaluated. The workflow is executed in the context of the submitting machine's environment (E_S). As the workflow is evaluated, the environment is defined internally and from E_S, resulting in E_D. Tasks are created and the environment that is specified by the task is derived from E_D, resulting in E_T. E_S is not included as variables would be incorrect or reference non-existent programs, libraries, and values at execution.

After the task is produced, transformations are applied that may append, update, or mask the provided variables. As a transformation is applied, E_T is written out to a script. The transformation can also use the values set in E_T to evaluate its environment. Applying these transformations produces a chain of environments E_{Tr} (E_a, E_b, E_c in Figure 5.8) as a result.

Tasks are placed on an execution node. The environment on the execution site varies from that of the submission site and is influenced by the batch system and execution node. The batch system environment(E_B) provides information about the assigned machines, available cores, and location of software modules and may be crucial for applications that use MPI or modules. The execution node environment (E_E) defines information such as local disks and available hardware.

The simple method of applying the environment is to apply all variables either at the beginning of execution or just prior to the task invocation. If applied initially, there are likely uninstantiated or unbound dependencies. If just prior to task execution, the context of each layer is evaluated using incorrect values or software. Both ultimately lead to a disconnect between the intended and the actual environment. To prevent this, as tasks are invoked each transformation creates a process that only applies the specified environment, limiting the environment's scope. Some transformations, such as containers, wipe or mask the provided environment. As transformation environments are applied, this should be taken into account, as the order and manner environments are instantiated may not carry through each transformation.

5.5 Applications of Transformations

We will now look at several example applications of transformations and how they can be used to improve the portability and robustness of workflows.

5.5.1 Sandbox Transform

A sandbox transform creates a directory, transfers files, and runs the command of the task. This simple lightweight transformation isolates the execution namespace from the workflow namespace, which allows for file renaming. The sandbox is removed after successful execution, which eliminates local unspecified files from polluting the workflow namespace and disk quota. If a task does fail, the sandbox can be captured and analyzed.

5.5.2 Container Transform

A container transform utilizes the whole command structure. A container *pre* command is used to pull down or unpack containers. This is done separately to differentiate failure of initialization from the invocation. A container *cmd* invokes the container with the nested command, which creates it own isolated process. Finally, a container *post* command cleans up container images, reporting exit status.

Calling the nested command from the container isolates the container arguments from the shell script invoked. This prevents issues with differentiating arguments, isolating file redirects, and instantiating an environment inside the container. Containers can also mask the execution environment, which can prevent an environment specified earlier from existing inside the container. Containers often increases the required resources to account for additional files, like container images.

5.5.3 Resource Monitoring Transform

A resource monitor transform measures the utilized resources during task execution. If limits are specified, the monitor will stop the task and report if the resources are exceeded. As the resource monitor relies on the expected resources, it can utilize the adjusted specification to adapt as transformations add required resources. Other functionality includes monitoring the files that are accessed and creating a time series of the utilized resources. The resource monitor uses the *cmd* to track the process. The resource monitor benefits from the sandbox directory as the isolation allows the sandbox to be monitored for disk usage. This does not increase the resources but is used to enforce them. In addition to the executable and usage summary, the resource monitor creates additional outputs such as the list of accessed files and resource usage time series, all of which are added as outputs.

5.5.4 Environment Transform

Regardless if a submit script, container, or virtual machine is used to run a task, there is often a need for configuration just prior to task execution. This is necessary in cases such as redirecting environment to reference data, configuring variables to

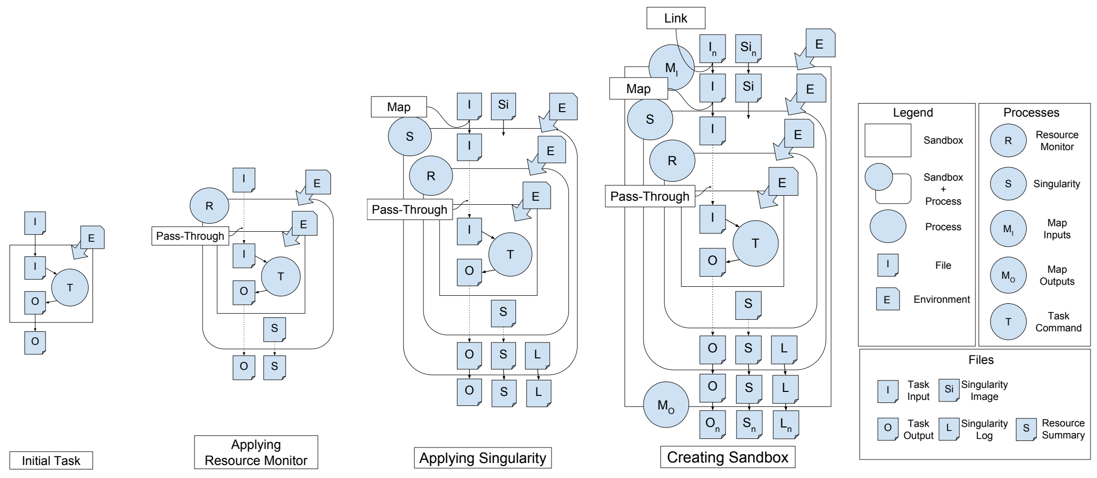


Figure 5.9. Evolution of task as transformations are applied. Starting from the left, we have the initial task with a single input and output. Next, a resource monitor is applied which passes through the original files, but also creates a summary of the resources used. After the resources monitor, a Singularity container is used to provide a consistent operating system, requiring a image to run from and creating a log. Finally, a sandbox is created to isolate execution, limiting file access when singularity maps the current directory.

include new libraries (such as `LD_PRELOAD`), or specifying a precise version of Java (by setting Java home and library paths). These types of transformation rely on the *pre* command to initialize the environment. This can also be done using the *environment* dictionary, though these value are directly exported and do not allow for nuanced initialization.

5.5.5 Failure Handling Transform

A transform that analyzes and handles errors at the task execution site allows for evaluations of the environment where the error occurred. Running evaluations only on failure limits the overhead on normal tasks and lessens the analysis burden of the user. This is used in determining software configuration/version incompatibilities, verifying if failure was due to limited resources, analyzing output files to prevent corrupted output, or process core-dumps into stack traces. Regardless of workflow size, automating error handling helps to handled errors allowing the user to analyze and address problems quickly. The error handling generally relies on the *post* to perform analysis based on the reported exit code or outputs.

5.6 Case Studies

5.6.1 Resource Usage in a Container

Resource monitoring helps to build an understanding of how a task behaves to accurately assign resources. If the task requires a container for execution, the resources utilized may be mischaracterized. As a result, it is useful to be able to separate the resource utilization of the task and the container. To do this, I first applied a resources monitor transformation to the task which will measure the resources used only by the task. Second, I applied the container transformation that allows for the application to run on different platforms.

The definitions of these transformations can be seen earlier in Figure 5.4 and Figure 5.7 for Singularity and the resource monitor respectively. To visualize the complexity that occurs when combining these Figure 5.9 illustrates each transformation.

```
makeflow bwa.mf –apply rmonitor.jx –apply singularity.jx
```

Using the above `makeflow` call, I executed a workflow that runs BWA[78]. This workflow partitions a large query and runs each chunk concurrently. This workflow was used as the basis to evaluate nesting the resource monitor and Singularity. This workflow was run in four configuration, both Singularity and the resource monitor, just Singularity, just the resource monitor, and the workflow with task sandboxes. I can examine the distribution of task execution time under these different configuration in Figure 5.10 and see that there is minimal additional overhead for each transformation. For these runs, Singularity utilized an image on a shared filesystem to limit the sandbox creation time, which can also be accomplished using a link. In situations with no shared filesystem the image is transferred and affects performance.

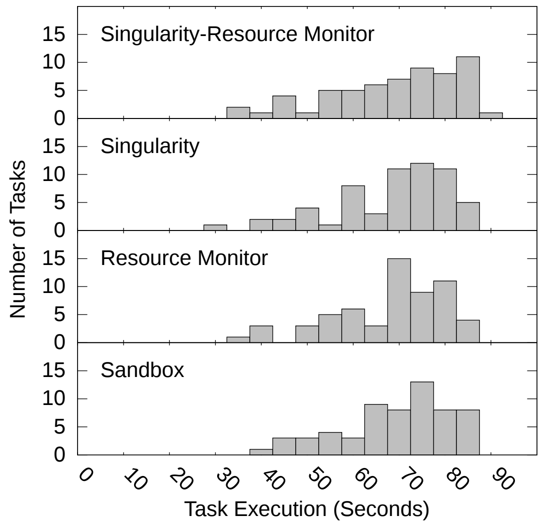


Figure 5.10. Histogram of task execution with nested transformations. The distribution of task execution grouped by applied transformations. The first configuration runs the resource monitor inside of a Singularity, the second runs just Singularity, the third runs just the resource-monitor, and the last runs the task inside of an application sandbox. We see that the distribution of execution time is consistent between runs, and the amount of transformation overhead is minimal.

5.6.2 Failure Analysis

When moving between sites or changing data it is possible that an application can intermittently fail causing a core-dump. These core-dumps are unwieldy to move around and provide limited insight into the cause of the failure and its environment.

To address this we wrote a transformation that analyzes a core-dump at the execution site, and sends back the resulting stack trace that is produced by GDB (GNU Project Debugger). This transformation provides several keys benefits. The first is that it allows for automated analysis of core-dump failures for the user. Performing this in the execution sandbox provides early resolution about the app that failed and which task created it. Also, core-dumps are bloated and contain all of the memory and stack, which are consolidated considerably in a stack trace. This consolidation limits the amount of data transferred back to the user.

Enabling the capture of core-dumps depends on your systems default settings. A common default uses the "core" prefix for core dumps, but also limits their size. To accommodate this, we set the *ulimit* to unlimited. After execution, if a core-dump was created we process it using GDB. This creates a stack trace that condenses the program failure. We implemented this transformation as seen in Figure 5.11.

To evaluate this we wrote an application that allocates 1MB of memory and then fails roughly 20 percent of the time, creating a core-dump. We scaled this experiment to see the difference of sending the stack trace instead of the core-dump. As expected, we say differences of several orders of magnitude of transfers, reducing as much as 2.4 GB down to 0.5MB. Table 5.1 shows a comparison of workflows from 10 tasks up to 10000. Using this technique on larger memory intensive applications would yield more significant reductions.

5.6.3 Complex Software Configuration

In scientific computing, researchers often construct and rely on a complex stack of analysis tools which are constructed over months to years of work and configuration. The resulting complexity often prevents researchers from scaling up or sharing their configuration with collaborators. There are several tools and solutions that

```
define StackTrace(T) {
 "command" : [
  "pre" : ["ulimit -c unlimited"],
  "cmd" : "./" + T.script,
  "post": ["gdb " + T.command{cmd} +
         "core* -ex bt > stack."+ T.ID ,
           "touch stack." +T.ID]
 ]
 "outputs" : T.outputs+ ["stack."+T.ID]
}
```

Figure 5.11. JSON showing stack trace transformation. The stack trace transformation allows a user to capture a core-dump of a failed task and convert it into a stack trace. This is done by setting the ulimit to allow the full core-dump, running the script, and then analyzing the core-dump with GDB. The step of touching the stack trace file prevents non-failed tasks from missing output.

exist to address this such as containers and build management tools (like Nix[41] or Spack[44]). This problem becomes more complex when the selected solution is not supported on a different platform, such as different container support or required installation permissions (super-user). For this reason we selected VC3-Builder[111], which is a user-level environment specification and construction tool.

As an example of complex software we use MAKER[18], a bioinformatic analysis pipeline which relies on 39 separate packages and a installed size of 4.2G. A MAKER installation requires careful dependency management as several tools rely on hard-coded, installation specific paths and installing by hand can take several hour. As a result, MAKER is often limited to a single carefully configured site. This is addressed with VC3-Builder and we want to leverage this to use MAKER on several sites for one workflow.

The transformation for VC3-Builder often simply invokes `vc3-builder`, specifying

TABLE 5.1

COMPARISON OF CORE-DUMP AND STACK TRACE DATA.

Workflow Tasks	Failed Tasks	Total Core Dump Size	Total Stack Trace Size
10	4	3.7MB	0.8KB
100	24	28.6MB	6.4KB
1000	174	214.9MB	48.8KB
10000	1957	2.4GB	553.1 KB

the required dependencies, and passing the command. However, because of MAKER's complexity several required packages have restrictive licenses requiring the user pass and unpack the libraries. In Figure 5.12, we can see a file, `manual-distribution.tar.gz`, which contains the restricted packages. Using *pre*, we are able to set up the correct directory structure, unpack the manual packages, and prepare for `vc3-builder`.

This workflow was executed using Makeflow and distributed with Work Queue[16], a master-worker execution platform. Workers were created on each target site, and tasks were distributed as worker were scheduled. To show the flexibility of transformations, workers were created on Stampede2, Jetstream, and HTCondor. Table 5.2 shows the task execution for each system, all of which were calculated using a single workflow.

5.7 Conclusion

I have now outlined and implemented an algebra for task and workflow transformations. This algebra, based on the the sandbox model of execution, allows transformations to be applied but stay distinct in execution, providing a crucial part of

```
define VC3-Builder(T) {
 "inputs" : T.inputs + ["vc3-builer",
            "manual-distibution.tar.gz"]
 "command" : [
  "pre" : [ "mkdir -p vc3-distfiles",
            "cd vc3-distfiles",
            "cp ../manual* ./",
            "tar xzvf manual*",
            "cd .."],
  "cmd": "./vc3-builder --require maker"
        + T.script, ]
 "resources":{
  "cores" : 4,
  "disk"  : T.resources{"disk"} + 4G,
} }
```

Figure 5.12. JSON showing VC3-Builder transformation. VC3-Builder is typically self-contained, and the specified *cmd* is sufficient for most software. MAKER, however, relies on several libraries with restricted licenses that must be provided by the user. As a result, the transformation must create the install structure and extract these libraries to the correct location prior to VC3-Builder. We specify cores for the make threads and increase the disk for the installation.

abstracting the scientific intention from the specific implementation. Using transformations, the core scientific analysis can be clearly defined with the specifics of execution, such as a container or environment setup, being applied as needed. With this technique, a user can take a workflow as is, move to new resources, and quickly adapt to the specific needs and configurations of the site. It is now be possible for site providers to create transformations that allow for more general changes in a way that fits their site.

Workflow transformations allow static workflows to quickly adapt, providing increased flexibility. When transformations are coupled with dynamic workflow expansion and resource management, we can now see a clear path transforming scientific

TABLE 5.2

MAKER BUILD-TIMES USING VC3-BUILDER ON VARIOUS SITES.

Build Time	Stampede2	Jetstream	HTCondor
(HH:MM)	01:29	00:22	00:30

analysis into a workflow that can be adapted for new data, sites, and configurations. This combination of techniques works well for static workflows using applications with defined or modeled behavior, but as computational demands grow we also need to consider more dynamic approaches to the workflow definition.

In the next chapter we will explore how the lessons learned about workflow management and transformations can be applied to dynamic workflows, and what other considerations are needed.

CHAPTER 6

APPLYING STATIC TECHNIQUES TO DYNAMIC WORKFLOWS

6.1 Introduction

In the previous chapters I explored several techniques for dynamically expanding static workflows and transforming the workflows to adapt to new sites and configurations. The challenges associated with relocating and transforming workflows are not isolated to static instances, but also applies to dynamic workflows. Historically, HPC applications and workflows are built in isolated environments supported by system administrators. In order to extract the maximum possible performance from specialized hardware, application creators rely on custom software stacks, hardware optimized code, and I/O behavior tailored to exploit high performance filesystems. As a result, these applications become dependent upon the specific environment in which they were created. As was demonstrated in Chapter 5, getting an application to run in a new environment is challenging and once running, may not be optimized or configured similarly for the hardware.

To demonstrate how the previous methods can be adapted to dynamic workflows I present a case study transforming the MAKER[17] bioinformatics pipeline. MAKER is a powerful pipeline, that incorporates many common data processing steps for genomes. MAKER simplifies genome analysis by managing a number of different applications and their dependencies, but this makes migrating MAKER difficult. MAKER has a large number of software dependencies that must be installed, limited scalability in high latency environments, and can produce configuration and execution errors that are difficult to diagnose. In this case study I show three methods for adapting dynamic HPC workflows between platforms.

- Portable reproducible environment for HPC, Cloud, and user resources, targeting support with user permissions.
- Ability to leverage resources (threads/MPI) on local and remote resources (multiple non-contiguous machines).
- Provide feedback for scalable and dynamic system, to aid in configuration and runtime decisions.

6.2 Jetstream

Jetstream [105, 112] is a NSF funded cloud service built on OpenStack. Operates similarly to Amazon EC2 and has support for data transfer and storage. Allows users created images to provide consistent platforms for review, comparison, and verification of results. One of Jetstream's goals is to provide a service that focuses on usability and support. As a cloud service, Jetstream is able to create and host custom images and environments that are more difficult to deliver on a more traditional HPC service.

6.3 Portable Reproducible Environments

Creating and supporting a reproducible environment is a current research topic of relevance with working being done at the platform level of OpenStack and Amazon, the container level by Docker and Singularity, and the deployment level of Jenkins, and Ansible. These three different levels each provide a different way of creating a reproducible environment. As part of this work, we targeted Jetstream, but also our Condor and SGE clusters, as well as user machines. A key consideration was the user's ability to verify a setup and configuration locally prior to moving to larger, possibly costly, resources.

6.3.1 Machine Images

A machine image is a pre-built snapshot of a desired software stack. Machine images can come in a variety of formats and are supported by a variable number of platforms, such as OpenStack. Machine images are ideal when working on a singleoperating system (OS) and platform as a base to provide consistent low level integration. However, outside of the scope of a single operation system, images have less portability. This reduced portability requires a developer to maintain machine images for each supported platform. This also precludes using the image at HPC facilities that lack user-level integration with machine images. The machine image would be an ideal target were we not also targeting users without access to systems like Jetstream.

6.3.2 Container Images

A container image is, similar to a machine image, a snapshot of a desired software stack. Container technology allows for users to run a container image on a supporting site using programs such as Docker, Singularity[74], and Charliecloud[95]. Container images provide an portability at a higher level than machine images, by running on any system that supports them. This allows for container images of variable OS to run on any supporting resource. Containers are also now beginning to be supported at HPC centers, such Singularity on a number of XSEDE resources.

However, in the case of both Docker and Singularity, super-user privileges are required for installing the software. Therefore leaving systems such as campus resources and local clusters unavailable. Charliecloud, assuming unprivileged user name spaces are enabled in the kernel, does provides a user-level container system. Unfortunately, not all kernels have this enabled by default, and availability varies between resources.

6.3.3 Deployment Services

In contrast with images, a deployment services install and organizes independent software packages into a single coherent package. This often includes finding either source code or pre-compiled binaries that are compatible, installing them, and configuring different software packages together. Examples of deployment services are as simple as apt-get and make, up to automated systems such as Ansible, Spack, Homebrew, and server-level orchestration tools such as Jenkins and Puppet.

In contrast with machine images, deployment services are often lightweight and only require a small number of predetermined packages to be installed. This allows for a high level of flexibility when deploying in a diverse set of environments and onto different platforms. While some cloud platforms may offer interoperability due to an underlying OpenStack framework, most platforms will require machine images to be recreated. Using a deployment services alleviates this by adapting to the current system and using generic build information from source where necessary. Deployment services adapt well to changes in versions and allow a user to customize these on the fly and test out different configurations.

Deployment services help to codify required build steps, and when written with multiple OSes in mind can reduce the work of supporting different platforms. However, with this flexibility comes the cost of building the software at each site for each use. Additionally, builds often rely on a remote data such as git repositories or the software's host site, as in the case of MAKER. The large variance in power and scope of these tools results in a number of different situations where super-user privileges may or not be needed.

6.3.4 VC3

As mentioned previously, several platforms were targeted including Jetstream, a Condor pool, a SGE cluster, and individual machines. As a result, we targeted a deployment services to allow for flexibility on both the OS and permissions. Some sites, such as local clusters, neither had the required tools installed nor allowed user-level installation. Targeting user-level permissions and OS agnostic features provides flexibility to target users' available resources.

VC3 [111] was used to install and configure MAKER. VC3 creates a sub-shell with a self-contained environment, and organizes software in a consistent, predictable manner. VC3 is based upon the idea of tool recipes, with inspiration taken from NixOS[42]. Each tool description consists of a recipe, dependencies, version, and environment variables.

VC3 has several features ideal for MAKER. The consistent file structure and referencing is important as some MAKER dependencies rely on hard-coded paths and strict relative locations. This allowed for resources to come from a number of configurations with the same structure. VC3 also requires only user level permissions, allowing the portability to any linux platform. VC3's interface allowed invocation of MAKER and organization of input data to be consistent between systems. Though there are other tools that could perform similarly, VC3 was picked for it flexibility, unprivileged operation, and familiarity. Similar solutions were written using Ansible, though this method was only used on Jetstream.

6.3.5 Deploying MAKER

When deployed onto Jetstream, MAKER was installed using deployment services to consistently handle the complex setup. VC3 and Ansible were both used for consistent builds on more widely provided base images, such as Centos 7 Developement. This process included installing several of MAKER's required programs and libraries that cannot be distributed in an automated manner due to developer licensing.

Though deployment services provide more flexibility when installing, a machine image provides easier, faster start-up for the users. To accommodate this, a machine image was built with the VC3 package installed, allowing execution to only verify the MAKER install and not have to build it each time. Additionally, Work Queue workers could be launched from the same machine image limiting traffic to only task input and output, rely on required software to be installed. This differed on our Condor cluster where there was no shared file system or container support. As a result a compressed install of VC3 was sent, so MAKER was built for each OS only once.

A build using deployment services provides a great deal of flexibility, but as users primarily used this just for MAKER, the repeated build overhead limited benefits. This was mitigated by using a machine image with MAKER installed using VC3 on Jetstream, and compressed VC3 on Condor. Using VC3 made rebuilding images, targeting new OS, and adding new features simple. By using a static image based on the deployment services, regardless of machine or container image, we can update software without the user needing to build every run.

6.4 Scalability

We define scalability as the number of cores that any one project was able to harness at a time. In an HPC context, this translates to the number of cores by the number of machines that were allocated to your job. MPI, being able to work on distributed memory machines, could work across the boundaries of several machines. In a cloud context, jobs are often limited by the size of selectable images. Some cloud platforms allow for creating sub-networks of machines, but the typical user (a researcher trying to run an analysis) will not have the time nor expertise to configuring them.

Scalability was achieved at two levels in this work.

1. Local parallelism, such as MPI, GPUS, or threads.
2. Distributed concurrency, partitioning across machines.

Our scalability goal was to limit both involvement in the provided software and work needed to target a different concurrency model. As such we decided on using the provided concurrency model of MAKER, MPI, to execute on each worker, where a worker is equivalent to a node. Using MPI allows us to scalably utilize resources, but lacks dynamicity as new resources become available. By leveraging the existing concurrency model we avoid the complexity of interfacing with MAKER's internal architecture. This also allows for smooth transitions between versions of MAKER and any underlying programs. This, based on experience from our previous tightly interfaced work[109], makes transitioning between versions, configurations, and platforms difficult, requiring repeated almost equal effort for each transition.

Relying on the underlying concurrency model for local parallelism lets Work Queue reason about the scaling and distribution of work to all available resources. In contrast with concurrency models like MPI and threads, Work Queue does not rely on having a statically determined set of resources. Orchestrating the work distribution with Work Queue allows users to add workers to increase resource pool, use resources from several allocations or sites, and provides fault tolerance to the application as a whole. Relying on MPI for local scalability, WQ-MAKER uses Work Queue to dynamically schedule on new resources.

6.4.1 MAKER's MPI Behavior

MAKER utilizes MPI as the primary means for scalability. Concurrency in bioinformatics is often available at the sequence (contig/scaffold) level. This is a division commonly used for partitioning data, as each sequence is a unique piece of data analyzed separately from the other sequences. MAKER then creates an additional level of concurrency using each analysis tool as a sub-process in the pipeline. This allows the burden of longer running sequences to be shared between multiple cores on the same machine and allows load balancing with smaller computational chunks. How-

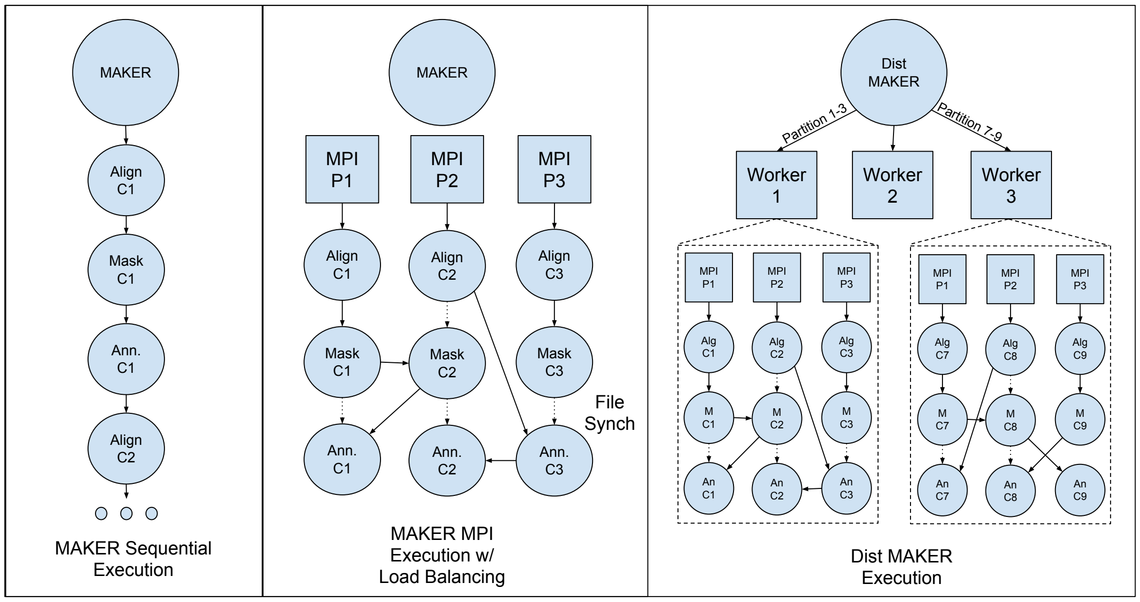


Figure 6.1. Diagram showing MAKER, MPI MAKER, and WQ-MAKER models. MAKER, without MPI, runs the sub-process analysis sequentially. MPI MAKER executes by sharing work, with MPI processes going to the pool of ready tasks and executing them. These processes are synchronized using data-structures(in MAKER) and data files(in MAKER's sub-tools) passed between sub-process (see dotted lines). WQ-MAKER partitions the data and sends it to separate workers. Each worker executes MAKER locally using MPI MAKER.

ever, the secondary level of concurrency can rely on intermediate files, for locks and tool specific data, to exist in shared space between tasks. This is not a requirement of MPI, which discourages this, though some MAKER's sub-processes rely on files for information and state. As a result, execution must also be located in a shared filesystem to allow for the outputs to be coordinated between all MPI processes.

6.4.2 WQ-MAKER

WQ-MAKER is built using the Work Queue API. The work is partitioned in different sizes, anywhere from individual sequences to the entire query file. WQ-MAKER does not split the work past the sequence level, as MAKER does with MPI, to prevent communication overhead from sub-processes. Each partition is a self-contained computational chunk that is distributed and organized after completion.

WQ-MAKER utilizes Work Queue's resource interface to allocate resources based on the partitions size and structure. Controlling at the task level allows for handling based on structure, such that long scaffolds are handled differently than short contigs. Using the resources allocated to a task by Work Queue, the worker can assign the appropriate amount of cores for MPI. To do this accurately assign resources a model is being developed as part of future work. Employing MPI on larger task, which occupy the entire worker, limits the master's management burden of monitoring workers.

6.4.3 Scaling Up vs Scaling Out

Scaling up helps to accelerate the annotation of genomes, but scaling up is not always the best usage of resources. A common assumption is that it is best to scale up using all of the available resources immediately. However, in practice this is seldom the truth as distribution of shared data (i.e. references), connecting to multiple resources, and spamming batch systems results in a gradual increase in resources, not an immediate deluge. This more gradual availability of ready resources can cause timeouts and under utilization of provided resources. This limitation leads users to under-provision applications instead of gradually adding resources as applications stabilizes. This was not addressed in this work other than to provide runtime feedback to the users about usage, so they are better informed. Work Queue masters track capacity of an application and inform users to add resources as the master can support more.

Compared with scaling up, scaling out can better utilize those resources for concurrent analysis of genomes, allowing for the same pool of resources to be shared, workers are kept busy with tasks from different masters. Work Queue allows for workers to match to multiple masters, enabling WQ-MAKER to share workers between instances. This provides an additional level of load balancing, without relying on additional underlying systems.

6.5 Exposing Execution Feedback

6.5.1 Clean Environment Builds

The first major obstacle was providing the users with clear feedback on the creation of the MAKER environment. Using deployment services to create, update, or modify the instance can initially cause errors and warnings, but once codified offer consistent builds. These build errors were typically only encountered by developers and could be diagnosed quickly. To communicate this, VC3 and Ansible need to have clear error handling and messages to identify errors. Build errors are mitigated when using static images, but can still be relevant as users want different configurations. Additionally, applications such as MAKER, where a variable number of the subsystems may be used, cursory testing does not always reveal configuration errors. As such new errors can occur when using different sets functionality, and must be clearly differentiated from runtime errors.

When using static images, like machine or container images, build errors are often self-explanatory, such as network errors, insufficient resources, or just bad luck. Jetstream's provided trouble-shooting gives possible solutions for users to attempt.

6.5.2 Deploying Workers

When deploying workers, users must log into each worker machine and manually start the worker process. This requires users to manage multiple ssh sessions and any changes to connection information (i.e. IP address or project name) must be reflected to all workers. We use an Ansible-playbook that allows the user to launch and manage workers from the host machine. For the user, this is as easy as creating

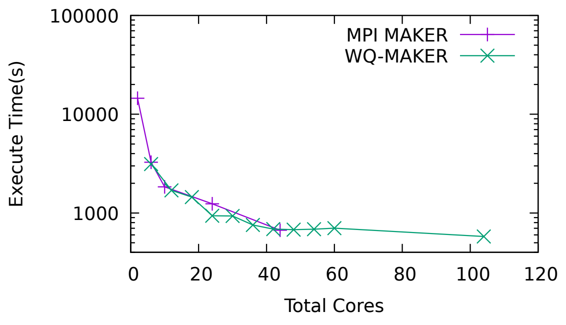


Figure 6.2. Graph showing comparison of methods on Fungal(41MB) dataset. The Fungal dataset contains 231 contigs. With similarly runtimes, we can see there was little or no overhead when using WQ-MAKER, though little gained beyond 34 cores.

an ssh-key and saving it in Jetstream, allowing the master machine to propagate commands using Ansible. On other systems, such as Condor and SGE, Work Queue maintains a worker factory that can submit workers resources become available.

Part of deploying workers is monitoring how many are actively being used by the Work Queue master. This is done using a status program that queries masters for active workers. As previously mentioned, masters also track capacity and can allow the factory to submit workers as masters are able to support more, as a result of workers being initialized or varying task execution time.

6.5.3 Evaluate Performance

Following a successful run, WQ-MAKER verifies successful runs to ensure proper execution. This is done by rectifying the final output files against the input data to ensure that all contigs were analyzed. The produced statistics are examined by WQ-MAKER to understand the behavior on this run's data. Work Queue provides a suite of graphing scripts for more in depth analysis. These graphs help understand the task execution, worker utilization, and file transfer speeds. All Work Queue graphs used in this chapter were created using these tools.

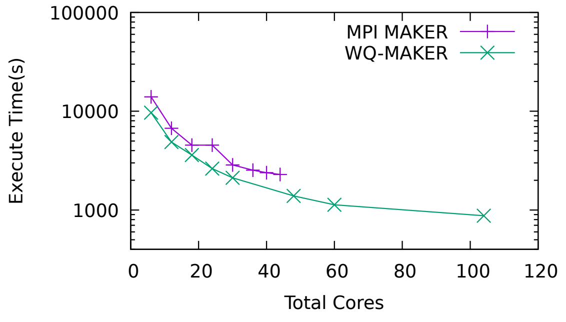


Figure 6.3. Graph showing comparison on partial Hummingbird(900MB) dataset. This subset of Hummingbird contains 5000 contigs. In this image we can see improvement of WQ-MAKER over the MPI run, likely as a result of reduced contention for resources.

6.5.4 Diagnosing Errors

Unfortunately, WQ-MAKER does not always run perfectly and it is important to help users diagnose errors and where they originated. After a run completes, WQ-MAKER prints the successes and failure of contigs. WQ-MAKER will retry failed contigs to ensure that it was not intermittent, possibly the result of network issues, software bugs, or resource contention. On repeated failure tasks are logged, reported to the user, and abandoned to avoid wasted effort retrying them further. The output of failed tasks is stored by WQ-MAKER, allowing for users diagnose the issue later.

Work Queue provides a debugging log that can be turned on to diagnose network errors, firewall issues, or file transfer failures. If an error happens while running WQ-MAKER, the Work Queue framework will print the error along with additional information in the debug log.

6.6 Evaluation

Though performance is important, this chapter did not directly measure the differences in start time between machines images, container images, and deployment services. Using deployment services we provide consistent builds, but leveraged machines images were available in Jetstream. The time needed to build the software is relevant when redeploying using deployment services. VC3 allows for multi-threaded deployment, which reduces a 1 hour build to roughly 10 minutes using between 16 and 24 cores. VC3 reuses existing builds allowing us to re-enter an existing build consistently in under a minute. The consistent file structure of VC3 allows us to build once and distributed to workers to install if needed.

WQ-MAKER was evaluated using several datasets. The MPI executions were done using differently sized Jetstream instances, up to the largest of 44 cores. WQ-MAKER used a master and a variable number of workers on medium instances, 6 cores. The fungal data set consists of 231 contigs. This data set is executed in roughly 4 hours using 2 cores locally. With increased cores, WQ-MAKER performance scales with MPI. Considering that fungal dataset is small, there is limited improvement after 40 cores, with a slight increase at 104 cores, as seen in Figure 6.2.

The hummingbird genome consisting of 5000 contigs, a medium sized genome sample. The results of running this genome through MPI and WQ-MAKER can be seen in Figure 6.3. For this larger sample we were able see improved performance over the standard MPI deployment as a result of lessening the memory burden on each partition using several workers. We were also able to see consistent reduction of execution time as we increase resources.

The saguaro cactus dataset consists of 573771 contigs This dataset took 57 hours to run using MPI MAKER on 24 cores. The raw CPU time spent during this job was 52 days of computation. Using WQ-MAKER, we ran this same dataset on our Condor cluster. Using Work Queue factory a mix of workers were launched. Each task was partitioned into 100 sequence, 8 core jobs to allow them to fit in the Condor instances. Figure 6.4 shows the number of tasks running over time as the workflow executed. The final execution time was 3 hours and 36 minutes, running 168 tasks concurrently at its peak, which equates to 1344 cores. As workers were dynamically added and removed the total CPU hours was only 1725.5, a result of WQ-MAKER's dynamic nature. Table 6.2 shows a comparison of performance using standard MAKER, MPI MAKER, and WQ-MAKER.

In total this project has been used by a number of users for annotation. We are actively developing and improving WQ-MAKER and working with users to better understand their needs. Table 6.1 shows a subset of the genomes that have been annotated using WQ-MAKER.

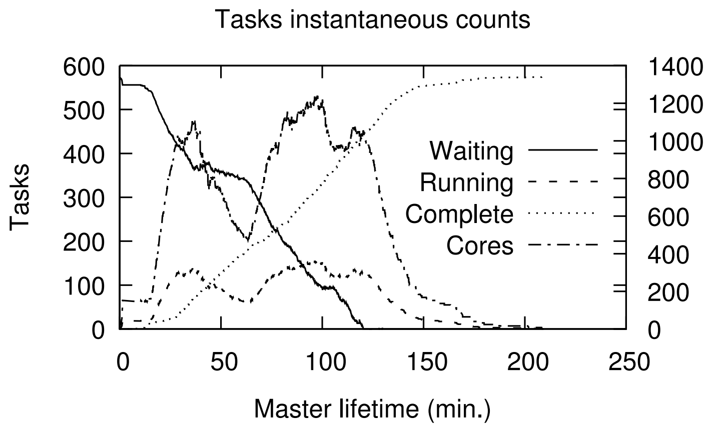


Figure 6.4. Performance of Saguaro Cactus genome(1.6GB) annotation using WQ-MAKER on Condor. The two lines of note are the running tasks and cores. The running tasks indicate the number of actively running MAKER tasks. The cores indicate the cores utilized by WQ-MAKER. Condor's volatility as a job scavenging systemcauses the variability in available resources.

TABLE 6.1

GENOMES SEQUENCED WITH MAKER ON JETSTREAM

Genome	Sequences	Workers	Runtime (hrs)
Sporobolus A	11,789 contigs	22-40	144
Sporobolus B	6,615 contigs	21-35	108
Brassica rapa	44,000 scaffolds	10	4
Zea mays W22	10 chromosomes	10	1440
Zea mays nc350	6460 scaffolds	22	72
Culex tarsalis	7478 scaffolds	40	120
MP29	5003 scaffolds	40	24
Pigweed	4126 scaffolds	20	120
Sclerotiana homeocarpa iso 10	231 contigs	10	6
Sclerotiana homeocarpa iso 11	257 contigs	10	6
Calypte anna	265 super-scaffolds	10	8
Kochia scoparia	19,671 scaffolds	21	72

TABLE 6.2

COMPARISON OF MAKER METHODS PERFORMANCE

	Execution Time	Cores	Total CPU Hours	Speedup
MAKER	52 days	1	—	—
MPI MAKER	57 hrs	24	1368	—
WQ-MAKER	3.6 hrs	80-1344	1725.5	15.8

The MAKER execution time is an estimated time based on CPU time of MPI MAKER. The results from MPI MAKER and WQ-MAKER are from actual runs of each. The speedup is calculated with reference to base MAKER, but WQ-MAKER still had a ∼16x speedup over MPI MAKER.

6.7 Conclusion

I have showcased the flexibility of the methods described through out, primarily the importance building abstraction into the underlying workflow. The workflow developed for MAKER re-engineered the previous iterations by separating the underlying computation from the difficulties of distributing in cloud environments. Relying on this abstracted design I illustrated how different methods for reconstructing the complex environment could be employed. This abstraction also allows the concurrency to be leveraged at both the underlying application-level and the workflow-level, allowing the dynamic nature of the workflow to be adjusted to the available resources. The flexible nature of dynamic workflows allows partition sizes to be adjusted toward more resources appropriate configurations.

Ideally, I would prefer to have the flexibility of dynamic responsive sizing in all workflows but because dynamic workflows require a much higher level of effort and knowledge to design, as compared to static workflows, they are out of technical reach for many users. Even implementing dynamically expanded workflows is simple in comparison, which can leverage the built-in features of the underlying static system, but does not provide the performance needed in large complex applications. In the next chapter I will explore a computational abstraction that leverages static application definitions with job coordinators designed to adapt the sizing similar to a dynamic workflow.

CHAPTER 7

CONTINUOUSLY DIVISIBLE JOBS

7.1 Introduction

Previously I evaluated the benefits of dynamic workflows and how the flexibility of partitioning allows performance to be adjusted as needed. However, the difficulty associated with writing dynamic workflows makes them an unfeasible solution for domain scientist looking to scale computation. In this chapter I propose an abstraction for separating the computation definition from the concurrent control of the execution. The goal is to allow domain scientists to write application and format specific job implementations that can be scaled with no additional expertise. In creating a flexible system, I focused on bag-of-task applications for their prevalence and to provide bounds on this solutions implementation.

For many of these applications, decomposition into a bag-of-tasks approach allows for a variety of execution platforms. Examples are seen in areas of batch systems (HTCondor, AWS Batch, SLURM), job execution frameworks such as MapReduce (Hadoop, Spark), or more general workflow management systems (Makeflow, Pegasus, Work Queue, Swift, etc.). These systems explore different ways to handle application execution, data management, and scalability. Application developers chose from a variety of concurrent systems based on needed features and then design an application using their knowledge of the underlying scientific application and the chosen system. However, having made a predetermined partition the bag-of-tasks approach often limits how responsive these systems can be. Furthermore, the application designer may not be an expert in either the underlying scientific application, scalable design, or both, producing an application that makes naive design and partitioning decisions which lead to sub-par performance and resource utilization.

I propose the Continuously Divisible Job abstraction, which introduces a dynamic sizing job interface for scientific applications. The Continuously Divisible Job interface is used with an *abstract job* for portability and operation abstraction and managed using a *job coordinator* that scales abstract jobs based on resources and utilization. The Continuously Divisible Job abstraction relies on a user specified interface to define the mechanism for how inputs are partitioned, jobs are executed, and the output handled. This interface exploits the application developers domain knowledge and allows for dynamic behavior via job abstractions. To further enhance data partitioning, we also propose *virtual files* which manage data indexing, lightweight partitioning, and just-in-time file instantiation. Virtual files help to limit the amount of redundant file reads and writes, exploit cached or shared files, and allow lightweight partitioning.

7.2 Contributions

Our contributions in the chapter are:

1. I define the Continuously Divisible Job abstraction and how it can be used to flexibly partition, distribute, and execute on large datasets.
2. I show how Continuously Divisible Jobs can be used to tune applications online for better performance and to escape bad initial partition configurations.
3. I discuss how Continuously Divisible Job coordinators can be used to construct a hierarchy of resources and coordinators that respond to performance and load balance as needed.
4. I define a virtual file abstraction and how data partitioning and movement can be minimized with indexing, lightweight partitioning, and just-in-time realization of files.

7.3 Challenges

To address the limitations of bag-of-tasks style concurrency, the methods for constructing and executing them need to be explored. In the area of concurrent execution, batch systems form the basis of computational power. Batch systems provide low-level mechanisms to describe and schedule work to a host of machines, providing large scale available resources. Batch systems are generally leveraged for high-performance computing as with SLURM[69], PBS, and Torque, or high-throughput computing such as HTCondor[108], Open Science Grid, and the WLCG. The general submission approach used by batch systems requires static submissions. Users aiming to harness more resources must partition, submit, and manage the work themselves.

As more general batch systems are limited in how dynamically work can be partitioned, more specific execution frameworks have been developed. This includes approaches such as MapReduce[33], bulk synchronous parallel[116], and more general data driven models such as workflows, all of which map cleanly with bag-of-tasks. MapReduce for example, as implemented in Hadoop, relies on the inherently parallel nature of the data analysis to scale smoothly. In the standard Hadoop setup, the execution is scheduled to the node where the data resides, relying on HDFS[101] to have created a sufficient number of data shards for high performance. This can lead to the predetermined splits having disproportionate work and long tail execution. Spark[126, 128] , which is built on Hadoop, can still fall prey to the same issues. Though Spark is able to leverage more performance by enhancing HDFS with resilient distributed datasets (RDDs), the partitioning is still programmer and system driven which can lead to poor configurations from imbalanced data and static sizing.

For more general data driven execution, scalable workflow systems offer high-level task abstractions allowing for more easily controlled scaling. Solutions for scalable workflow systems fall into two categories: static and dynamic. In a static approach the user defines the size, partitioning, and scalability of the work and relies on the workflow system for distribution and execution. This approach relies on the application developer's knowledge of the data to define and predetermine the partitioning. After this point the workflow system directs and manages the concurrency, dealing with resources, communication, and failure management. Example systems using this approach include Makeflow[2] and Pegasus[38], or could be defined more generally using the Common Workflow Language[5] or the Workflow Description Language[117]. Again, the static nature of partition decisions limit the responsiveness of the underlying workflow system. Once a workflow is defined, feedback cannot be used to adjust the size or shape of tasks. This static sizing can lead to poor performance, often lacking knowledge of number of resources, network performance, or even application execution time. For example, in Figure 7.1 we show how the total runtime of BWA is influenced by the task size on a fixed dataset.

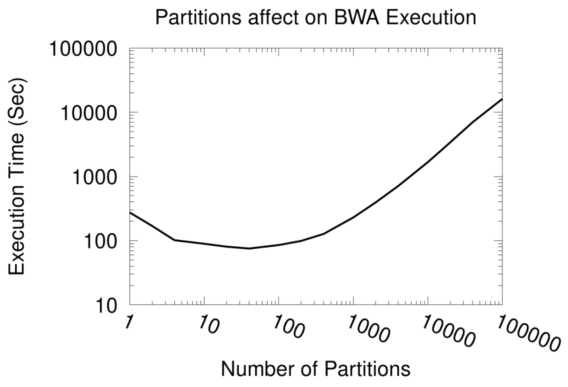


Figure 7.1. Effect of partitioning on BWA execution. This is a sample bioinformatics(BWA) workflow's performance with the input partition size varied. The number of partitions was varied from 10 to 100,000, using a statically size query of 1,000,000 sequences.

The dynamic approach requires the application developer to devise and direct the concurrency of the application. This involves a more nuanced understanding of both the application (i.e. partitioning, performance, resource requirements) and distributed design (i.e. task scheduling/ordering, failure management, resource acquisition). Examples of the direct approach include Work Queue[16], Swift[129], Parsl[9], and RADICAL Cybertools[90]. The challenge is that this approach requires both knowledge of the application core behavior and an understanding of distributed application behavior.

The existing solutions provide many options for defining and executing work, but lack flexibility when running bag-of-tasks style work. In general, these approaches rely on static partitions, either defined by the developer or the underlying system, which constrain work similarly. Some of the common challenges that arise from static sizing are high partition and execution overhead, long tail execution from imbalanced work, and rigid mapping to resources. Additionally, when the execution system is unable to further manipulate sizing it is difficult to model solutions for more complex execution configurations, such as adapting to heterogeneous resources, nested resources for data distribution, or identifying and isolating failures.

7.4 Continuously Divisible Jobs

Continuously Divisible Jobs are applications with defined minimum computational units, such as an events, sequences, or slices of input data that can be processed in large batches. This structure is common and can be seen in high-energy physics event processing for particle collisions, genome sequence alignment in large queries, and large batch simulations for model observation and validation. Each computational unit has a short execution time, often on the order of seconds to minutes. However, the collection of these units are large, often processing thousands to millions of events in a single batch, which greatly increases the execution time and resources

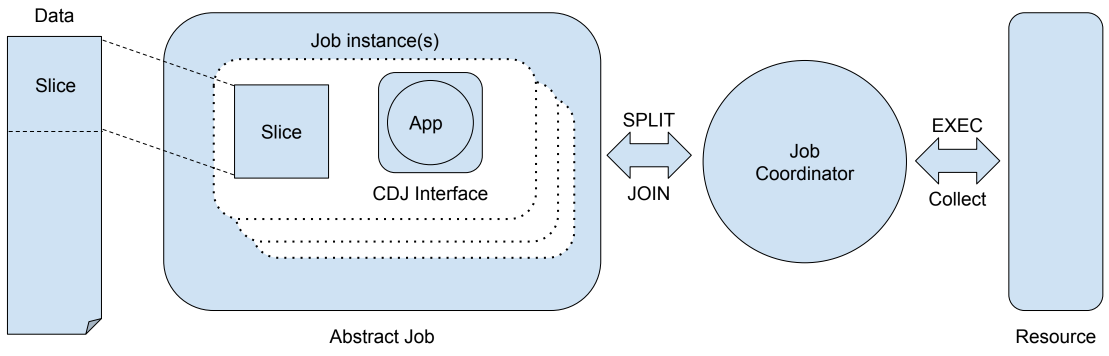


Figure 7.2. Diagram of Continuously Divisible Job architecture. This diagram outlines the relationships between the data slice, Continuously Divisible Job interface, abstract jobs, and the job coordinator. An abstract job consists of a data slice, the applications, and the interface wrapper. Abstract jobs may map a single slice, or contain several independent. These abstract jobs are managed by the job coordinator, which splits, executes, and joins the work. The job coordinator decides how and when jobs are placed on resources.

needed.

Continuously Divisible Job interfaces are implemented in terms of five functions: `SPLIT`, `JOIN`, `EXECUTE`, `TO_DESC`, and `FROM_DESC`, that can be used to dynamically handle and execute these large datasets, with each function operating on any amount of data, from a single slice of data to the full dataset. Applications that have implemented this interface can be started and managed using a job coordinator that partitions and executes the data. These job coordinators can be designed for a number of execution platforms such as batch systems, execution managers, or run locally. Using the abstract jobs, the job coordinators can be chained together to create flexible hierarchical stacks of resources that can share work and load balance as needed. These job coordinators can also be designed to tune the partitions to more efficient sizes, but more importantly can be used to escape from bad initial configurations (i.e. naive job partitions).

In this section we will define the design and capabilities of the Continuously Divisible Job interface implemented by an application, abstract jobs, and job coordinators that operate on abstract jobs to partition and execute the application. Dividing the Continuously Divisible Job abstraction allows the *mechanism* of partitioning and executing to be defined by the application domain expert, and the *policy* of executing these job with job coordinators is left to the distributed system expert or system administrator of a site.

7.4.1 Operations

To achieve the dynamic sizing and resource utilization of the Continuously Divisible Job abstraction we define a set of operations and attributes that applications need to implement. These definitions can be implemented directly by the application designer or domain scientist, as decisions on how and where to partition data, what parameters are needed for executions, and the expected environment can directly impact the validity of the results. The Continuously Divisible Job interface instructs the abstract job on the mechanism of job handling, and are the only components the application designer needs to implement.

`SPLIT(JOB`$_s$`, COUNT, SIZE)::[JOB`$_{s1}$`,..,JOB`$_{sn}$`]`

Given a number and the size of splits this creates a set of new jobs, containing the number of created jobs at the specified size and an additional job containing any remaining slices. Each new job should be able to reconcile its context in the origin job and the dataset as a whole. If the split job does not contain enough slices for the full count of partitions, split should return a set with as many as possible. Splitting a job does not necessarily perform partitioning, but logically separates the slices for execution. In a base approach this may partition data, but as will be explored later in Section 7.5 there are other methods for late or just-in-time data partitioning.

`JOIN(JOB`$_a$`, JOB`$_b$`)::[JOB`$_{join}$`]||[JOB`$_a$`, JOB`$_b$`]`

Join takes the specified job and joins it with the calling job returning a set of new jobs. The implementing application should determine if the two jobs are joinable and either return a single combined job or the passed-in jobs in sorted order. This allows for application specific join behavior for either contiguous, ordered or unordered slice combinations. The join operation may be called on jobs that have or have not been executed, requiring the application developer to handle both cases. As provided by abstract jobs, the application will not need to join executed and non-executed jobs. If the application chooses to allow for more application level management, the application can simply merge all jobs, and hold its own application level slices.

`EXECUTE(JOB)::RESULT`

The execute operation performs the application core computation. Application execution could be in the form of spawning a process, running a shell command, or simply calling a function. There are no parameters for the execute functions as all application level variables should be specified in the jobs definition. This operation may be executed remotely, additionally requiring the list of files, environment, and resources used.

`TO_DESC(JOB)::DESCRIPTION`

`FROM_DESC(DESCRIPTION)::JOB`

The `TO_DESC` and `FROM_DESC` define the basics for serializing and deserializing the application. As the location of execution is determined outside of the applications control the serialization allows job instances to be consistently packaged, moved, and reconstituted for execution, a core component used in both bootstrapping this initial job and using remote execution job coordinators. There are many methods that can be used for implementing the serialization and deserialization of an application instance such as converting objects to JSON or language specific approaches (i.e. Python pickle). System agnostic approaches such as JSON are preferred allowing for a wide range of execution environments, possibly even mixed within a single application.

In combination with the core operations implemented for an application, there is also a set of attributes that allow the above operations to be called intelligently and the overall performance tuned. These are not strictly necessary, but provide insight that allows the jobs to be run on a variety of systems without assuming shared filesystems, complete node use, or consistent environment for execution.

- Inputs files: Provides both the static files needed for all jobs and the data specific to each job's slices.
- Output files: The expected outputs of the job.
- Resources: The size of the resources needed for executions, such as cores, memory, and disk.
- Environment: The expected environment variables used by the underlying application.
- Size: The total size of the application instance allowing precise splitting and performance analysis.
- Result: An attribute that determines is the slice has be completed, and if so if it was successful.

Having defined an application's interface, an instance of the application can be instantiated. This can be done either by directly creating an instance or by using the `FROM_DESC` to bootstrap. Each partition of the data is considered a slice, and the combination of data slices with the application and its interface create an abstract job. Abstract jobs, as will be defined below, can contain a number of slices, allowing dynamic sizing. The relationship between data, applications, abstract jobs, and the job coordinator can be seen in Figure 7.2

7.4.2 Abstract Jobs

The core bridge of the Continuously Divisible Job abstraction between applications and job coordinators are abstract jobs. Abstract jobs provide the management of higher level applications. An abstract job allows a single large slice or multiple slices to be grouped and managed without the application developer needing to handle every possible case of splitting and merging partitions. Non-mergable partitions can coexist in a single job, enabling more dynamic sizing. Executed and non-executed partitions can be passed as one or separated with no additional application handling. Application specific files and libraries can be captured with minimal user involvement, such as for bootstrapping an application remotely.

To facilitate the flexible handling of slices without pushing the handling onto the job coordinator, the abstract job layer needs to handle splitting mixed- and multi-slice jobs, joining and sorting possibly non-contiguous jobs, and provide high level mechanisms for reasoning about and classifying jobs. This provides additional functionality for the job coordinators to take advantage of in the areas of defining and tracking application results, grouping jobs based on the underlying state (unexecuted, failed, successful), and logically packing undersized slices together. This layer's consistent interface allows the job coordinator to organize and execute jobs with no knowledge of application, and likewise the application designer does not need to interact with job coordinators. On execution the application bootstraps an instance with data, creates an abstract job, and submits to the coordinator.

7.4.3 Job Coordinators

A job coordinator is the execution and policy management of the Continuously Divisible Job abstraction. Job coordinators are intended to be implemented by the execution system developers and site administrators, and are the primary component deciding on job sizing, execution, and collection. As such, the application developer (e.g., the scientist) does not implement a coordinator, but selects from existing job coordinators based on need. This could be in the form of a multicore executor, a coordinators that submits to their execution system, or a mix of job coordinators to achieve the desired configuration. Job coordinators work directly with abstract jobs to distribute and execute the specified computation. At its most basic, a job coordinator receives an abstract job and executes it, but more likely a job coordinator partitions the work to utilize many core machines. Using the `TO_DESC` and `FROM_DESC` functions in tandem allow job coordinators to adapt to a variety of resources and sites.

As job coordinators operate on abstract jobs, the job coordinator needs to rely on its own feedback and metrics to inform partitioning size and performance. This allows the job coordinator to tune for general performance, without being application specific. Tracking the time to create slices, execution time per slice, or cost of joining slices are just a few ways to measure performance. Designed properly, job coordinators can avoid bad performance in several cases. Jobs with high overhead can be scaled up as the increased size may mitigate execution overhead and improve throughput. Data can be processed in batches with time limits to avoid losing entire submissions if the resources time out or are lost, providing timed checkpoints. The same structure of timed batches can be used to load balance between fast and slow workers or highly variable slice execution. Partitions can be sized to support any number of workers, while maintaining user responsiveness. The more flexible the execution system, the more ways jobs can be tuned and resource utilization improved.

In addition to job performance tuning, job coordinators can be used to distribute work in a number of ways. The recursive nature of partitioning and joining allows several coordinators to be used in combinations to address the users needs. This can be used to achieve multi- and mixed- tiered execution models as seen in Figure 7.3. This model could be further extended to have hierarchical job coordinators that submit to a tree of resources, but allow the resources to report results directly back to the source. In cases with large input data but compact results, this model would allow for parallel distribution but centralized result collation. An example of a hierarchical model similar to what is shown in Figure 7.3.

Finally, job coordinators can be used to identify and isolate failing partitions. As is often the case, data may be malformed or corrupted causing the application to fail. In static approaches, it is left to the user to bisect the failing task and isolate the culprit. However, the dynamic sizing and result tracking of abstract jobs allows the job coordinator to automate isolation, minimizing analyzed slices.

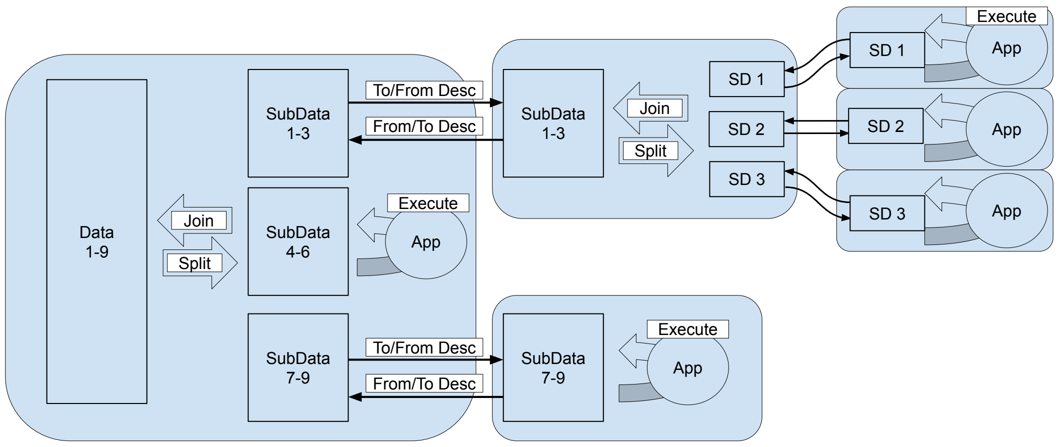


Figure 7.3. Capabilities of Continuously Divisible Job abstraction. Using the Continuously Divisible Job abstraction, jobs can be partitions and run locally. Using the same design we can also distribute to multicore workers or to other job coordinators that further distribute the work. This highlights how the Continuously Divisible Job interface relates to overall recursive design.

7.4.4 Design Considerations

In developing the application operation for Continuously Divisible Job interface, there are several design considerations that should be taken into account. These considerations include such topics as methods for file partitioning, how namespaces and job sandbox should be handled, and how result ordering can affect performance. Defined below are some of the larger considerations along with designs and methods to resolve them.

7.4.4.1 File Partitioning

File partitioning is often a crucial part of Continuously Divisible Job applications, as applications generally read in and analyze the full input dataset. As a result, each split requires new data partitions to be create, but generates redundant data in naive approaches. If each split directly partitioned the data, the remaining post-split data is repeatedly written. This leads to a multiplicative effect on the necessary storage for execution, not even accounting to the redundant file reads and writes. To prevent this, methods for just-in-time or late file realization may be needed when using the Continuously Divisible Job abstraction. Two examples of file partitioning are shown in the analysis of this chapter, the first being a naive split-on-partition where files are written when the `SPLIT` operation is called. This creates unnecessary files, but allows the application to be executed without modification similar to more static invocations. The second uses virtual files, defined in Section 7.5, to reference data slices. The virtual file abstraction allows for flexible data handling, such as the just-in-time file instantiation or direct data access.

7.4.4.2 Job Namespaces

A job's namespace consists of all the files needed to complete the job. In the base case this is simple as the namespace contains the uniquely named files used to execute. Each job is invoked the same way, so it is often tempting to use consistent generic names in the invocation, but this fails when scaling as each partition loses file name uniqueness. This leads to the more general issue of clearly defining the namespace such that any split and join results in uniquely identifiable files and names. There are several approaches to resolving this, from the easiest and often most straight forward of utilizing the partition name, generating and tracking unique identifiers for each name, or creating names based on content derived hashes. Each of these methods prevents collision within a single Continuously Divisible Job application, but with the possibility of other executions and concurrently running instances, these methods have varying success and should be considered carefully.

7.4.4.3 Execution Sandbox

Similar to job namespaces, many applications operate naively in an execution environment. Naive environment usage is common where standard data is used or when the applications relies on complex configurations of libraries and references. Common examples of this could be using hardcoded paths and file names for inputs or resolving reference databases from environment variables. In these cases and more, it is likely the application will not operate correctly when run concurrently in the same namespace, either file namespaces, process namespaces, or both. As a result it may be necessary to run an application in a sandbox to isolate both the file and process namespaces. The common nature of this problem has provides many solutions, such as using containers[74, 89], sandboxes, and wrappers[56] to isolate each application instance.

7.4.4.4 Job Ordering

Continuously Divisible Job applications require the implementation of the join operation, which incorporates combining and consolidating application results. The type and structure of this output data can have dramatic affects on the overall performance of the applications. To illustrate this let us consider two different potential applications, X and Y. X is large data parallel analysis, where future pipeline steps require sorted result ordering. Joining X jobs together requires only combining contiguous jobs. Further, for performance, the application only joins from the first job upward, appending, to prevent repeatedly writing and re-writing data as out-of-order jobs are joined. Y is a complex simulation with a small input and output datasets consisting mainly of statistics. Joining Y jobs requires only combining the statistics and can be done completely out-of-order. This allows Y to quickly join results are they are completed and retrieved

7.5 Virtual File Abstraction

A virtual file defines a subset of a physical file, allowing large data files to be logically partitioned quickly. A virtual file points to a source file, keep bounds on the logical slices (logical offset and range), and can quickly resolve a slice's actual location in the large file (byte offset and range). virtual files offer a lightweight mechanism for partitioning and sub-referencing larger datasets, without the need to copy out the actual subset of data. To facilitate quick repeated resolution from a logical slice to a physical position, an index should be be constructed. Using this quick translation, a sub-set of the larger file can be realized as a physical file just prior to use. As the physical offset is only needed prior to file realization, data can be partitioned, re-partitioned, or joined with little actual computational cost. Using virtual files, an application can adapt quickly to performance feedback, without making redundant copies of data.

7.5.1 Operations on Virtual Files

Virtual files have several operations that allow for faster or more lightweight operation on data than standard files. Operations such as indexing, partitioning, location, and instantiation can be done on standard data files, but introduce considerable overhead in file system activity and redundant work. In addition to the core virtual file operations, virtual file serialization is also key to allowing for recursive partitioning expected of jobs.

7.5.1.1 Indexing

Indexing a virtual file, as with any index, parses the origin data file and tracks the location of each logical slice in the data source. If the logical slices are uniform in size, the indexing step is quick and only the size of slice needs to be tracked. If, however, the data is non-uniform in size the byte offset for each slice needed for tracking. This indexing step may be time intensive initially, but as more partitions are created the cost is amortized over execution by avoiding file accesses and redundant data copies. Though it is possible to store this information in memory, it is advisable to use a more compact persistent representation that can be distributed and recovered. Due to the speed provided by indices, many applications and formats have existing methods for creating and accessing indexes, which can be leveraged for an application's virtual file.

7.5.1.2 Partitioning

Partitioning allows for quick logical splitting of data source to virtual files. This partitioning can be done with no handling of underlying data, using only logical slices. Partitioning at the virtual file level is much faster than partitioning physical file because it handles logical slices that are resolved as needed. Resolving the byte range can be done lazily when the range is actually needed to limit accesses to the index. Operating on the virtual files allows for partitions to be merged or further split with less concern for the overhead of reading and writing the actual data. Removing the need to read and write for each partition allows job coordinators to group or further split applications with less overhead.

7.5.1.3 Index Look-up

Index look-up provides the physical location of a logical slice. This is used by partitioning and realization to find a slice location, but can also provide this information to the underlying application. An example of this can be seen later, where modifications were made to the BWA application to accept byte offsets for work. This look-up allowed BWA to jump to the relevant location prior to analysis, eliminating the need to create partition files entirely.

7.5.1.4 Instantiate

Instantiation writes a logical range of slices into a physical file. Applications that do not accept offsets and ranges of a virtual file must be instantiated as a physical file. In a local system, file instantiation simply uses index look-up on the first slice and the last slice, writing the returned byte range to a new file. As more complex job coordinators are used, there may be cases where the data source is not available, such as remote execution, and the realization needs to carry context. A virtual file instantiation not only copies the defined source data, but also creates a copy of the index and its new data offset. This allows the new offset to be quickly computed using the index look-up minus the sub-data file offset. This combination of the index and sub-data files allows for virtual files to be used in the same recursive partitioning and distribution as Continuously Divisible Job applications.

7.5.1.5 Serialization

Serialization of a virtual file allows data files to be partitioned remotely, even when the source data is unavailable. This serialization is similar to the Continuously Divisible Job operations, and should define how to reconcile the data partition within the data source. As such, a more complex hierarchy of how the partition was created is unnecessary, only the source data, index, and the partitions relative location. This information can be used to resolve the correct data offset, as inform job coordinators how to resolve the data remotely from the data source.

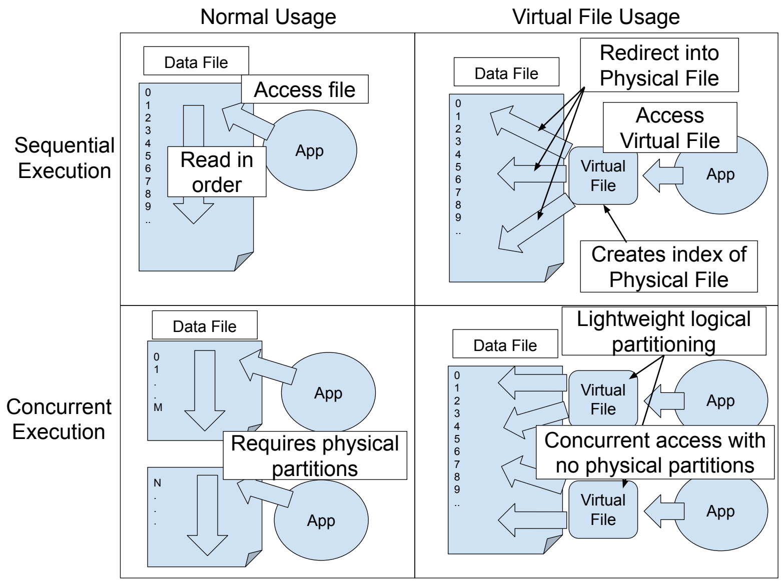

Normal Usage
Virtual File Usage
Sequential Execution
Concurrent Execution
Data File
Access file
Read in order
App
Redirect into Physical File
Access Virtual File
Virtual File
Creates index of Physical File
Requires physical partitions
Lightweight logical partitioning
Concurrent access with no physical partitions

Figure 7.4. File usage of normal and virtual file implementations. In the normal case, the application directly opens and navigates the file. In the virtual file case, the reads are directed through the virtual file using either an index or query to resolve and redirect to the read location. This approach introducing overhead of creating the index and resolving the data. However, as these application are partitioned for concurrent execution, the normal usage requires expensive physical partitions, while partitioning of virtual files is essentially free, allowing concurrent use of the data.

7.6 Implementation

This section discusses the abstract implementations of the example Continuously Divisible Job applications. The implementations for all aspects of Continuously Divisible Job were done in Python, and the results below are based on this design.

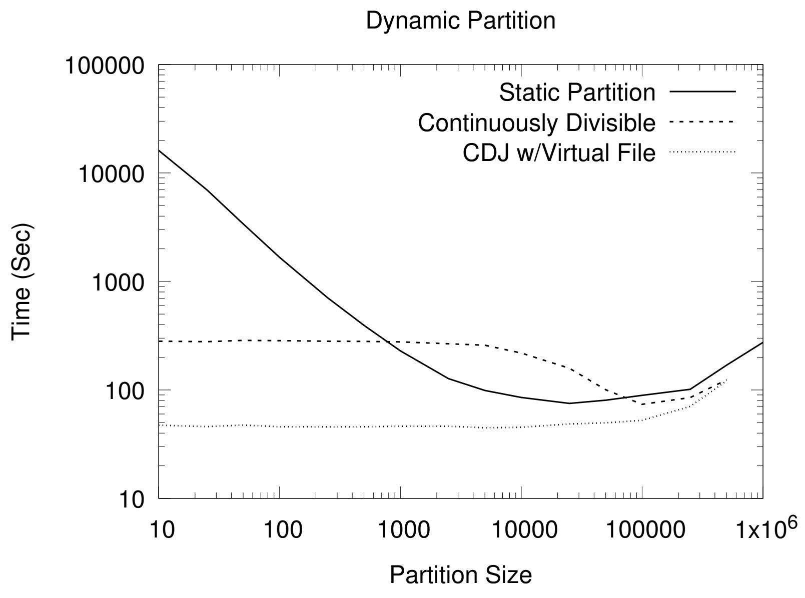


Figure 7.5. Performance of static partitions, Continuously Divisible Job, and Continuously Divisible Job using virtual files. Jobs are partitioned as previous work is finished allowing the size to dynamically find a stable partition, and all of the cores remain busy. As a combination of lightweight partitioning and direct data access, we can see the virtual file implementation performs consistently better. The base Continuously Divisible Job, however, see a bump in execution time from the added cost of redundant file writes.

In addition to the application implementations, we also further discuss several ways that virtual files could be implemented and used.

7.6.1 Example Continuously Divisible Job Application

For this chapter, two application implementations were written. The first is a bioinformatics alignment tool, BWA, that is a common tool in genome annotation pipelines. BWA was selected as it compares each query sequence against a reference dataset, allowing the query dataset to be partitioned down to a single sequence. The second is a high-energy physics event analysis for detecting dimuon event candidates. This application serves as an investigation into using the complex ROOT format for

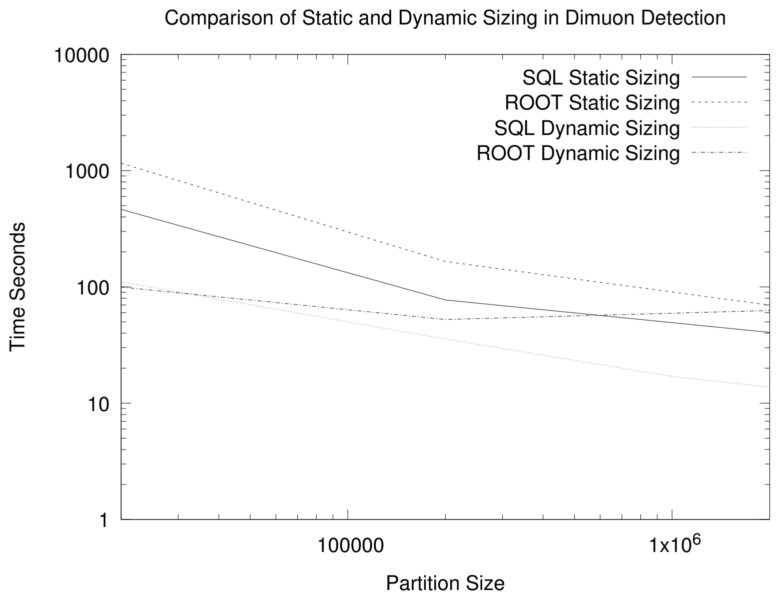


Figure 7.6. Performance of ROOT and SQL Dimuon detection methods. To evaluate both the benefit of the virtual file and dynamic sizing, each implementation was run using static partitions and then dynamic partitions. We can see in the static case, the SQL implementation provides a consistent advantage over the ROOT file. However, in the dynamic case, as the initial partitions get smaller the difference shrinks. This is due to ROOT's high overhead more quickly pushing to a better partition.

concurrent event analysis and compares with a SQL-based virtual file implementation. For both example applications this section will outline how the interface was implemented, along with design challenges that needed to be addressed.

For the BWA interface two approaches were taken, a simple direct partitioning approach and a indexed virtual file approach. For the simple direct approach, split relies on the standard fastq format which is commonly partitioned. Each split results in new jobs, each with a unique data file. When called to execute the job invokes BWA, passing its unique data slice and the intended output as arguments. After completing with results to join, the simple case checks that the joining jobs are in the same state, returning if they differ. If the data hasn't been processed the inputs are combined. If the data has been processed, then the results are combined. To increase efficiency, the outputs are combined in order from the first slice on, limiting the number of redundant appends.

The second approach relies on an indexed virtual file. When this approach is initialized, an index is created for the source data. After this, split is a lightweight call that partitions logical ranges, without handling physical data. To further exploit the indexed data, modifications were made to BWA that utilized byte ranges to quickly seek to the intended data. This implementation removed BWA's requirement for physically partitioned data, allowing minimal file manipulation. The join functionality remains the same, as the outputs created are identical.

We also use the Continuously Divisible Job abstraction to detect dimuons in a High Energy Physics (HEP) analysis. In a typical HEP task, events from a detector are analyzed one at a time collecting diverse statistics. Events are recorded in a tree-like structure in which statistics are grouped into particles, which in turn are grouped into events, luminosity sections, and so on, which are stored in ROOT files[15].

In our implementation, jobs are given a range of events to process. To feed data to the jobs, we experimented with two different virtual files implementations. In the first one, each job is submitted with the same ROOT file, and ranges of events are read using `uproot`, a tool developed by the DianaHEP project[96] that converts the tree-like structure of the ROOT file into ragged `numpy` arrays. For the second one, we flatten the root file into a `sqlite` database, in which each row encodes a particle in some event.

For each, serialization was trivial and relied on the converting dictionaries of class information into JSON using Python.

7.6.2 Virtual Files

Virtual files can be implemented in several ways such as directly in the application by physical offset and range resolved from slices (used in BWA), in an API that copies when needed, or by the filesystem redirecting virtual files to sections of larger files. An example of the first was implemented in BWA, which allows BWA to process a section of the data without having to write out a sub-data, limiting both the space needed to operate on a sub-data and the time needed to partition the data.

In the second case, a structure is needed to represent the data as partitions are defined and moved around. Using Work Queue[16], a virtual file can be specified for a job and realized at the worker without ever directly writing the copy at the master process. As the file already needs to be read for transfer, the intermediate step of writing out the sub-data is skipped.

The third case could be implemented in file system, where a new file type is created similarly to a symbolic link. In addition to the link to the origin file, an offset and range define the size. This implementation would interpose an `EOF` at the range to support normal file usage. Additionally it would be prudent to force read-only semantics on virtual files and their origin counterparts to prevent invalid offsets and ranges, as well as changing the intended data.

7.7 Results

For the analysis of the Continuously Divisible Job abstraction, we compared several of the design features discussed previously. The majority of these results were gathered using the BWA implementation, for which we had a moderately sized dataset. The results for this chapter were evaluated on a 250 MB subset of the dataset. For the static partitioning results that are compared against the Continuously Divisible Job implementations, a Makeflow BWA workflow was used. The structure of this was a simple split-join workflow that is common in bioinformatics. Makeflow was used as it supports similar execution platforms, has little overhead, and provides native multi-core execution. The following results will show:

(a) Under good configurations, execution time is similar to static partitioning.

(b) Under bad configurations dynamic sizing can find better configurations.

(c) Virtual files provide better performance even with static partitioning.

(d) In tiered, but uncoordinated configurations, resources can be under-utilized.

7.7.1 Dynamic Sizing

The first test that we wanted to investigate was the effect of dynamic sizing using Continuously Divisible Jobs. As can be seen in Figure 7.5, we are comparing static batch partitioning and on-demand dynamic job partitioning. For each data point the jobs were run concurrent on 8 cores. The value on the X axis is the initial partition size. The dynamic sizing is based on a basic hill climb algorithm that attempts to find the size with the highest throughput. As was briefly discussed earlier, the static partition's execution time is limited at the right by under-utilizing the available cores, and that the left with increased overhead of file creation and job management. For showing the Continuously Divisible Job implementations, we compare the base implementation, with file creation on split, and the indexed virtual file approach that directly accessed the data. In both cases, we see that the dynamic sizing allows the initial bad configurations can be escaped, leading to better performance to the left of the graph. For the virtual file implementation, we see consistently better performance, as it benefits from less file access and increased flexibility. In the base implementation, the middle section of the graph we see worse performance than static, with the cost of partitioning and re-partitioning data combating with the benefit of dynamic sizing, but when compared to the orders of magnitude worse behavior at the left it may be a reasonable compromise.

For evaluating the dimuon detection implementation, we used a comparison of the ROOT and SQL approaches, as well as comparing static and dynamic partitioning. Similarly to the results we saw with BWA, when looking at Figure 7.6, we can see that the dynamic sizing allows both implementations to perform consistently, avoiding the increasing execution time of smaller static partitions. Interesting, the gap between the SQL and ROOT implementation shrinks when using dynamic sizing. This is likely the result ROOT's higher overhead, causing the dynamic sizing to shift more quickly.

7.7.2 Virtual File Effectiveness

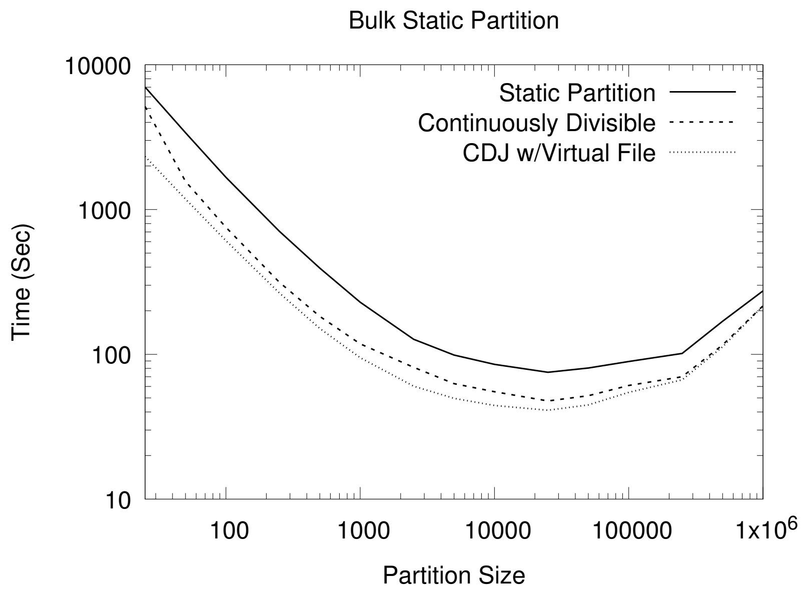


Figure 7.7. Performance of virtual files with static partitions. This shows a standard BWA bioinformatics workflows where only the size of each partition is varied. This is compared with similarly static partitioned Continuously Divisible Job implementations. We can see that lightweight partitioning and dynamic joining help them out perform the fully static case.

For the second test we wanted to isolate the effect of virtual files when using static batch partitioning. The static batch partitioning eliminates the dynamic sizing and just compares the benefits of virtual files. This test disadvantages the virtual file implementation as the data is still accessed and indexed, but the limited partitions do not allow the cost to be amortized. As can be seen in Figure 7.7, we compare static batch partitioning between a static workflow, the base BWA implementation, and the virtual file implementation. For each data point the jobs were run concurrent on 8 cores. The value of on the X axis is the static partition size. It is important to note that these results are shown in log scale. The static workflow approach shows the same behavior as before, and we now see similar trends in both the base and virtual file implementations. The interesting result that can be see is that for both Continuously Divisible Job implementations the results were consistently below the workflow approach, a result of the dynamic joining and lighter weight partitioning. The gap between these approaches is consistent at log scale, showing the increased benefit in poor configurations. Additionally, the performance of the virtual file approach was consistently better than the base approach, showing that by limiting the file partitions we gain a consistent performance benefit. Similarly, when comparing the dimuon implementations we can clearly see advantages to the SQL approach, with consistently better performance (Figure 7.6).

7.7.3 Tiered Sizing

The last test we wanted to explore was the affect of tiered sizing, and how it can be either beneficial or negative. In the previous dynamic sizing test, only a single layer was sizing the results. For this test, we used a master-worker framework to partition at the master level and for cores at the worker (Work Queue). The results can be see in Figure 7.5. In this graph we are comparing again against the static partitioning. Each of the subsequent lines is grouped by the initial master partition

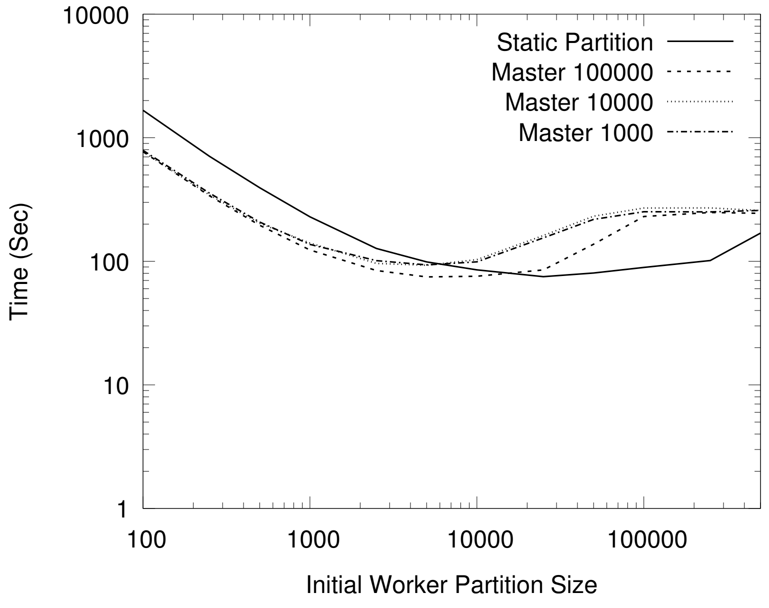


Figure 7.8. Comparison of tiered performance for several configurations. This compares static partitioning against several configurations of Continuously Divisible Job using virtual files. Each line is grouped by the initial master size and the X axis shows the initial worker size. As can be seen across each case, at the right the limited worker partitioning under-utilizes the cores. To the left, the small starting size suffers from initial splitting overhead.

size and graphed along the worker initial size. Looking at these results, we can see that though it was able to find reasonable configuration is many cases, at the edges of the search space, the combined dynamic partitioning competed with itself slowing any corrective movement. For example, to the right of the graph, performance was limited by the worker's partition being larger than the master's. This forced only a single core to be used, wasting resources. This approach shows that job coordinators can be quickly combined, but more exploration is needed to understand how to avoid negative feedback.

7.8 Conclusion

I have introduced the concept of Continuously Divisible Jobs and discussed how dynamic sizing can be used to address limitations of static partitioning. I show how implementing the Continuously Divisible Job interface allows applications to be dynamically partitioned, executed, and distributed to dynamic resources using abstract jobs and job coordinators. To further leverage this approach, we introduced virtual files, and explored how they can be leverage to provide lightweight partitioning, fast data access, and eliminate redundant reads and writes.

Continuously Divisible Jobs builds on the ideas presented throughout this dissertation. Combining the lessons learned for in Chapter 3 and Chapter 6, the API was defined allowing a static dataset to be partitioned dynamically and at will, which provides flexible responsive sizing. The job coordinator, as well as the design considerations, take into account the need for resource management as laid out in Chapter 4. Finally, the complete design of Continuously Divisible Jobs relies on the underlying sandbox model of execution, as described in Chapter 5. Further, because this is built on the sandbox model of execution, jobs executed through a job coordinator can be nested and transformed as needed for execution on a number of platforms.

CHAPTER 8

CONCLUSIONS AND BROADER IMPACT

8.1 Recapitulation

In the area of scientific computing, data is growing faster than any single machine can analyze in a reasonable time. As a result, scientists are looking for ways to scale up their computation and accelerate research. Scientist have limited experience with workflow computing and need options that are direct and straightforward to implement, as opposed to more complex and nuanced solutions such as MPI. Workflows provide a clear path for domain scientists to increase their concurrency without needing to completely re-engineer their current software.

I have argued that building abstractions into the underlying models allows for highly flexible workflows that can be adapted to new data, sites, and configurations. Workflow abstractions allow the pure scientific intent to be declared without the nuance or clutter associated with run-time execution. Using this core workflow, abstractions can be combined to provide a responsive and flexible execution. Relying on the domain scientist to describe the pure scientific intent allows applications to be used directly, without re-writing the analysis.

I first explored this using static workflows, where I described several methods for abstracting workflow structure. I achieved flexibility for different data structures (size, density, or configuration) with dynamic workflow expansion; which allows a core workflow to quickly be molded and tuned for new data. These expanded workflows now use more dynamic resources, to control this, I show how workflows can be statically analyzed for expected need and dynamically managed to execute within expects bounds. Coupling both methods, a user can leverage a large number of resources. The flexibility provided by the workflow expansion provides the use of new resources and sites. To accommodate this movement, I illustrate an algebra that allows workflows to be transformed as needed. This algebra breaks down the definition of task and shows how transformations can be quickly applied. Mapping this process across all tasks allows the workflow to be adapted as needed.

Having shown the flexibility gained with static workflows, I applied these methods to dynamic workflows. Dynamic workflows, by their nature, provide responsive sizing, but tightly couple the application to the workflow design. I showed how decoupling (abstracting) the workflow from the application allows transformations to be applied without losing the responsiveness. Taking this a step further, the decoupling allows the application to use built-in parallelism alongside the workflow's concurrency.

Though decoupling the dynamic workflow from the underlying application decreases the difficulty of implementation, the barrier to entry is still much higher than static workflows. A middle ground is achieved using Continuously Divisible Jobs, which provides an easier implementation alongside the responsive sizing of dynamic workflows. I proposed and demonstrated how Continuously Divisible Jobs allow an application to define five functions and execute in a dynamic and responsive manner. A key benefit of this approach is that it abstracts the application definition from the intricacies of execution, which are provided by a job coordinator. Ideally, this allows users to define an application, couple it with data, and use the site's provided job coordinator without knowing the underlying configuration.

Through the methods and techniques I explored workflows are able to be designed closer to the user's technical knowledge without losing flexibility and performance. Lowering the barrier to entry for users is the key to facilitating research in all fields, and the goal to be attained.

Looking at scientific computing more broadly, data and the need for computation will continue to grow as new and faster methods are discovered. Likewise, the scale of computation being produced to meet this need is becoming more complex and varied. Gone are the days that a single machine will be sufficient for a complete dataset or analysis. It is unclear if the data production or available computational power will be larger in the end, but what is becoming obvious is that as both grow the interface between them becomes increasingly complex. This complexity has outstripped the abilities of most domain scientist to have the time or experience to harness or fully utilize these emerging technologies. These domain scientist turn to distributed computing solutions, such as workflows, to bridge that gap, acting as a negotiator of resources and computation.

Solutions like Continuously Divisible Jobs offer an avenue of communication between the specifics of an individuals research and the ever increasing landscape of resources whether it be a campus cluster, a national scale supercomputer, or corporate cloud infrastructure. The flexibility of Continuously Divisible Jobs crucially provides a common interface that is powerful enough that an abstract job can be migrated, scaled, and distributed to almost any resource. The clear expectations of a job definition also make implementing a job coordinator easy, and the API provides enough flexibility that the job coordinator can adapt to performance.

Overall, abstractions are becoming increasingly important as they allow research at both ends of the spectrum, from domain scientists doing critical research to system administrators scaling with new technologies, to communicate and scale. Providing flexible interfaces that facilitates both ends, such as Continuously Divisible Jobs, is crucial to the continued growth and success of scientific computing.

8.2 Future Work

8.2.1 Continuously Divisible Jobs

In this work, I introduced the idea of Continuously Divisible jobs. The implementation presented here looks at single applications, abstracting and distributing their computation. This structure loosely translates to a simple split-join workflow, with no considerations of future analysis with the data. Compute time is spent on re-organizing and combining results, which may then be re-partitioned for future computation. Eliminating this unnecessary computation would allow for an more efficient and flexible pipeline, but a language is needed to address these relationships.

Extending the Continuously Divisible Jobs API to describe how a partition relates to the larger data set would allow for more implicit concurrency. Some computation is by its nature completely data-independent, while others look at the dataset holistically. What does a specification look like to describe this in a meaningful way? The current implementation implicitly defines the relationship between a data format and an application. To allow for more flexible data and computation handling, a clear characterization of the relationship of an applications interact with input and output formats should be explored.

If applications have a definition that explicitly states the parity of inputs to computation to outputs, applications can be combined dynamically and scalably with little user interaction, relying on the characterization to determine if the partitions are computationally valid. In its current state, much of the time developing and testing workflows has been invested into checking the validity of data manipulations which should ideally be defined by the application developer. If the Continuously Divisible Jobs API was extended to more clearly define format-application parities, abstract data-pipelines could be defined that scale to available resources and performance.

8.2.2 Nested Workflows and Transformations

A major consideration during this work was how do we consistently and safely apply transformations to workflows. I think the methods developed for applying transformations to a workflow can be harnessed to provide hierarchy to workflows. Hierarchy in workflows is intended in two ways, first by allowing for a more rich definition of workflow structure and second by allowing nested workflows. Nested workflows have been supported one way or another in Makeflow for several years, with a loose definition of how to correctly communicate between layers of execution.

The workflows used and described throughout this dissertation are by definition a flat set of rules that are translated into a DAG. However, the user writing the workflow has an implicit set of expectations for the workflow's structure, such as analysis pipelines or resource groups. In Makeflow, categories allow for rules to be grouped together by expected resource use, but any more nuanced groupings are unavailable. Extending Makeflow to support hierarchical workflows would allow for more structurally meaningful grouping and staging to be performed.

In well written JX Makeflows an analysis pipeline is mapped to all set data partitions. This hierarchical structure should be preserved and utilized to batch or pre-stage tasks to workers with relevant data. This will cut down on time spent pushing and pulling data with no computation underway. This mechanism exist in Work Queue and Makeflow to over-stage jobs, but there is little considerations to doing this based on pipelines.

If hierarchical structure can be studied and well supported within Makeflow, workflows themselves can be nested and reasoned about in the same way. Ideally, a nested workflow behaves similarly to an analysis pipeline, but with a wider set of rules and resource needs. In addition to this nested structure, a clear specification for communicating between workflow levels would be beneficial. This is by no means a trivial task, and requires careful thought of how to handle questions such as resource allocations, environment specifications, and possibly deep nested workflows.

8.3 Impact: Software and Publications

The work throughout this dissertation was peer-reviewed and presented and several different conferences and journals. The work outlined in Chapter 3 was presented as a poster at CCGrid 2014[57] where it won best poster, and at the IEEE International Conference on eScience 2015[58]. The work outlined in Chapter 4 was published in IEEE Transactions on Parallel and Distributed Computing[60]. The work outlined in Chapter 5 was presented at the IEEE International Conference on eScience 2018[56]. The work outlined in Chapter 6 was presented at the IEEE International Conference on Cloud Engineering[59]. Finally, the work outlined in Chapter 7 was presented at the IEEE International Conference on eScience 2019[61].

The software developed through the work is supported by the Cooperative Computing Lab, and distributed within the CCTools package. Additionally, example workflows for both Makeflow[55] and Work Queue are preserved in separate repositories to allow for much of this research to be reproduced.

BIBLIOGRAPHY

1. E. Afgan, D. Baker, N. Coraor, B. Chapman, A. Nekrutenko, and J. Taylor. Galaxy cloudman: delivering cloud compute clusters. *BMC bioinformatics*, 11 (Suppl 12):S4, 2010.

2. M. Albrecht, P. Donnelly, P. Bui, and D. Thain. Makeflow: A Portable Abstraction for Data Intensive Computing on Clusters, Clouds, and Grids. In *Workshop on Scalable Workflow Enactment Engines and Technologies (SWEET) at ACM SIGMOD*, 2012.

3. I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock. Kepler: an extensible system for design and execution of scientific workflows. In *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*, pages 423–424. IEEE, 2004.

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, Oct 1990.

5. P. Amstutz, M. R. Crusoe, N. Tijani, B. Chapman, J. Chilton, M. Heuer, A. Kartashov, D. Leehr, H. Mnager, M. Nedeljkovich, and et al. Common workflow language, v1.0, Jul 2016. URL `https://figshare.com/articles/Common_Workflow_Language_draft_3/3115156/2`.

6. P. Amstutz, M. R. Crusoe, N. Tijani, B. Chapman, J. Chilton, M. Heuer, A. Kartashov, D. Leehr, H. Mnager, M. Nedeljkovich, M. Scales, S. Soiland-Reyes, and L. Stojanovic. Common Workflow Language, v1.0, 7 2016.

7. T. G. Armstrong, J. M. Wozniak, M. Wilde, and I. T. Foster. Compiler techniques for massively scalable implicit task parallelism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 299–310, Piscataway, NJ, USA, 2014. IEEE Press. ISBN 978-1-4799-5500-8. doi: 10.1109/SC.2014.30. URL `https://doi.org/10.1109/SC.2014.30`.

8. Arun Ramakrishnan, Gurmeet Singh, Henan Zhao, Ewa Deelman, Rizos Sakellariou, Karan Vahi, Kent Blackburn, David Meyers and Michael Samidi. Scheduling data-intensive workflows onto storage-constrained distributed resources. *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, 2007.

9. Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. M. Wozniak, I. Foster, M. Wilde, and K. Chard. Parsl: Pervasive parallel programming in python. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '19, pages 25–36, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6670-0. doi: 10.1145/3307681.3325400. URL `http://doi.acm.org/10.1145/3307681.3325400`.

10. Y. N. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. M. Wozniak, I. T. Foster, M. Wilde, and K. Chard. Parsl: Pervasive parallel programming in python. *CoRR*, abs/1905.02158, 2019. URL `http://arxiv.org/abs/1905.02158`.

11. M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 66:1–66:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. ISBN 978-1-4673-0804-5. URL `http://dl.acm.org/citation.cfm?id=2388996.2389086`.

12. D. Blankenberg, G. V. Kuster, N. Coraor, G. Ananda, R. Lazarus, M. Mangan, A. Nekrutenko, and J. Taylor. Galaxy: A web-based genome analysis tool for experimentalists. *Current protocols in molecular biology*, pages 19–10, 2010.

13. G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science Engineering*, 15(6):36–45, Nov 2013. doi: 10.1109/MCSE.2013.98.

14. R. Brun and F. Rademakers. ROOT: An object oriented data analysis framework. *Nucl. Instrum. Meth.*, A389:81–86, 1997. doi: 10.1016/S0168-9002(97)00048-X.

15. R. Brun and F. Rademakers. Root an object oriented data analysis framework. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 389:81–86, 04 1997. doi: 10.1016/S0168-9002(97)00048-X.

16. P. Bui, D. Rajan, B. Abdul-Wahid, J. Izaguirre, and D. Thain. Work Queue + Python: A Framework For Scalable Scientific Ensemble Applications. In *Workshop on Python for High Performance and Scientific Computing (PyHPC) at the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (Supercomputing)* , 2011.

17. M. S. Campbell, C. Holt, B. Moore, and M. Yandell. Genome Annotation and Curation Using MAKER and MAKER-P. *Curr Protoc Bioinformatics*, 48:1–39, Dec 2014.

18. B. L. Cantarel, I. Korf, S. M. Robb, G. Parra, E. Ross, B. Moore, C. Holt, A. Sanchez Alvarado, and M. Yandell. MAKER: an easy-to-use annotation pipeline designed for emerging model organism genomes. *Genome Res.*, 18(1): 188–196, Jan 2008.

19. W. Chen and E. Deelman. Partitioning and scheduling workflows across multiple sites with storage constraints. In *9th International Conference on Parallel Processing and Applied Mathmatics*, 2011. URL `http://pegasus.isi.edu/publications/2011/WChen-Partitioning_and_Scheduling.pdf`. Funding Acknowledgements: NSF IIS-0905032.

20. W. Chen and E. Deelman. Workflow overhead analysis and optimizations. In *6th Workshop on Workflows in Support of Large-Scale Science (WORKS 11)*, 2011.

21. W. Chen and E. Deelman. *Partitioning and Scheduling Workflows across Multiple Sites with Storage Constraints*, pages 11–20. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-31500-8. doi: 10.1007/978-3-642-31500-8_2. URL `http://dx.doi.org/10.1007/978-3-642-31500-8_2`.

22. W. Chen and E. Deelman. Integration of workflow partitioning and resource provisioning. In *The 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2012)*, 2012.

23. W. Chen, R. Ferreira da Silva, E. Deelman, and R. Sakellariou. Balanced task clustering in scientific workflows. In *9th IEEE International Conference on e-Science (eScience 2013)*, 2013. URL `http://pegasus.isi.edu/publications/2013/chen-clustering-escience2013.pdf`. Funding Acknowledgements: NSF IIS-0905032 and NSF FutureGrid 0910812.

24. W. Chen, R. Ferreira da Silva, E. Deelman, and R. Sakellariou. Using imbalance metrics to optimize task clustering in scientific workflow executions. *Future Generation Computer Systems*, 46:69–84, 2015. doi: 10.1016/j.future.2014.09.014. URL `http://pegasus.isi.edu/publications/2014/2014-fgcs-chen.pdf`. Funding Acknowledgements: NSF IIS-0905032 and NSF FutureGrid 0910812.

25. W. Chen, R. Ferreira da Silva, E. Deelman, and T. Fahringer. Dynamic and fault-tolerant clustering for scientific workflows. *IEEE Transactions on Cloud Computing*, 4(1):49–62, 2016. doi: 10.1109/TCC.2015.2427200. URL `http://pegasus.isi.edu/publications/2015/chen-tcc-2015.pdf`. Funding Acknowledgements: NSF IIS-0905032, NSF ACI SI2-SSI 1148515, and NSF FutureGrid 0910812.

26. A. Chervenak, E. Deelman, M. Livny, M. Su, R. Schuler, S. Bharathi, G. Mehta, and K. Vahi. Data placement for scientific applications in distributed environ-

ments. In *2007 8th IEEE/ACM International Conference on Grid Computing*, pages 267–274, Sep. 2007. doi: 10.1109/GRID.2007.4354142.

27. O. Choudhury, N. Hazekamp, D. Thain, and S. Emrich. Accelerating comparative genomics workflows in a distributed environment with optimized data partitioning. In *C4Bio Workshop at CCGrid*, 2014.

28. O. Choudhury, D. Rajan, N. Hazekamp, S. Gesing, D. Thain, and S. Emrich. Balancing Thread-level and Task-level Parallelism for Data-Intensive Workloads on Clusters and Clouds. In *IEEE Conference on Cluster Computing*, 2015.

29. O. Choudhury, D. Rajan, N. Hazekamp, S. Gesing, D. Thain, and S. Emrich. Balancing thread-level and task-level parallelism for data-intensive workloads on clusters and clouds. In *IEEE Cluster*, 2015.

30. P. J. Cock, C. J. Fields, N. Goto, M. L. Heuer, and P. M. Rice. The sanger fastq file format for sequences with quality scores, and the solexa/illumina fastq variants. *Nucleic acids research*, 38(6):1767–1771, 2010.

31. P. Couvares, T. Kosar, A. Roy, J. Weber, and K. Wenger. *Workflow Management in Condor*, pages 357–375. Springer London, London, 2007. ISBN 978-1-84628-757-2. doi: 10.1007/978-1-84628-757-2_22. URL `https://doi.org/10.1007/978-1-84628-757-2_22`.

32. J. D. de St. Germain, J. McCorquodale, S. G. Parker, and C. R. Johnson. Uintah: a massively parallel problem solving environment. In *Proceedings the Ninth International Symposium on High-Performance Distributed Computing*, pages 33–41, 2000. doi: 10.1109/HPDC.2000.868632.

33. J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1251254.1251264`.

34. E. Deelman and A. Chervenak. Data management challenges of data-intensive scientific workflows. In *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 687–692, May 2008. doi: 10.1109/CCGRID.2008.24.

35. E. Deelman, J. Blythe, Y. Gil, C. Kesselman, S. Koranda, A. Lazzarini, G. Mehta, M. A. Papa, and K. Vahi. Pegasus and the pulsar search: From metadata to execution on the grid. In R. Wyrzykowski, J. Dongarra, M. Paprzycki, and J. Waśniewski, editors, *Parallel Processing and Applied Mathematics*, pages 821–830, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24669-5.

36. E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny. Pegasus: Mapping scientific workflows onto the grid. In M. D. Dikaiakos, editor, *Grid Computing*, pages 11–20, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-28642-4.

37. E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny. *Pegasus: Mapping Scientific Workflows onto the Grid*, pages 11–20. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-28642-4. doi: 10.1007/978-3-540-28642-4_2. URL `http://dx.doi.org/10.1007/978-3-540-28642-4_2`.

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

39. E. Deelman, S. Callaghan, E. Field, H. Francoeur, R. Graves, N. Gupta, V. Gupta, T. H. Jordan, C. Kesselman, P. Maechling, J. Mehringer, G. Mehta, D. Okaya, K. Vahi, and L. Zhao. Managing large-scale workflow execution from resource provisioning to provenance tracking: The cybershake example. In *2006 Second IEEE International Conference on e-Science and Grid Computing (e-Science'06)*, pages 14–14, Dec 2006. doi: 10.1109/E-SCIENCE.2006.261098.

40. E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. Maechling, R. Mayani, W. Chen, R. F. da Silva, M. Livny, et al. Pegasus, a workflow management system for science automation. *submitted to Future Generation Computer Systems*, 2014.

41. E. Dolstra, M. de Jonge, and E. Visser. Nix: A safe and policy-free system for software deployment. In *Proceedings of the 18th USENIX Conference on System Administration*, LISA '04, pages 79–92, Berkeley, CA, USA, 2004. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1052676.1052686`.

42. E. Dolstra, A. LÖh, and N. Pierron. Nixos: A purely functional linux distribution. *J. Funct. Program.*, 20(5-6):577–615, Nov. 2010. ISSN 0956-7968. doi: 10.1017/S0956796810000195. URL `http://dx.doi.org/10.1017/S0956796810000195`.

43. M. M. Eshaghian and Y.-C. Wu. Mapping heterogeneous task graphs onto heterogeneous system graphs. In *Heterogeneous Computing Workshop, 1997.(HCW'97) Proceedings., Sixth*, pages 147–160. IEEE, 1997.

44. T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral. The spack package manager: Bringing order to hpc software chaos. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 40:1–40:12, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3723-6. doi: 10.1145/2807591.2807623. URL `http://doi.acm.org/10.1145/2807591.2807623`.

45. W. Gentzsch. Sun grid engine: Towards creating a compute power grid. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, CCGRID '01, pages 35–, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1010-8. URL `http://dl.acm.org/citation.cfm?id=560889.792378`.

46. L. Gerhardt, W. Bhimji, S. Canon, M. Fasel, D. Jacobsen, M. Mustafa, J. Porter, and V. Tsulaia. Shifter: Containers for HPC. *Journal of Physics: Conference Series*, 898:082021, oct 2017. doi: 10.1088/1742-6596/898/8/082021. URL `https://doi.org/10.1088%2F1742-6596%2F898%2F8%2F082021`.

47. W. Gerlach, W. Tang, A. Wilke, D. Olson, and F. Meyer. Container orchestration for scientific workflows. In *2015 IEEE International Conference on Cloud Engineering*, pages 377–378, March 2015. doi: 10.1109/IC2E.2015.87.

48. B. Giardine, C. Riemer, R. C. Hardison, R. Burhans, L. Elnitski, P. Shah, Y. Zhang, D. Blankenberg, I. Albert, J. Taylor, W. C. Miller, W. J. Kent, and A. Nekrutenko. Galaxy: a platform for interactive large-scale genome analysis. *Genome research*, 15(10):1451–1455, 2005.

49. Gideon Juve, Benjamin Tovar, Rafael Ferreira da Silva, Dariusz Krol, Douglas Thain, Ewa Deelman, William Allcock and Miron Livny. Practical resource monitoring for robust high throughput computing. *Workshop on Monitoring and Analysis for High Performance Computing Systems Plus Applications*, 2015.

50. J. Goecks, A. Nekrutenko, J. Taylor, and T. G. Team. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biol*, 11(8):R86, 2010.

51. T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, G. von Laszewski, C. Lee, A. Merzky, H. Rajic, and J. Shalf. Saga: A simple api for grid applications. high-level application programming on the grid. *COMPUTATIONAL METHODS IN SCIENCE AND TECHNOLOGY*, 12:7–20, 01 2006. doi: 10.12921/cmst.2006.12.01.07-20.

52. C. K. Gurmeet Singh and E. Deelman. Optimizing grid-based workflow execution. *Journal of Grid Computing*, 3(3-4):201–219, December 2005.

53. O. Harismendy, P. C. Ng, R. L. Strausberg, X. Wang, T. B. Stockwell, K. Y. Beeson, N. J. Schork, S. S. Murray, E. J. Topol, S. Levy, et al. Evaluation of next generation sequencing platforms for population targeted sequencing studies. *Genome Biol*, 10(3):R32, 2009.

54. T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Slacker: Fast distribution with lazy docker containers. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 181–195, Santa Clara, CA, 2016. USENIX Association. ISBN 978-1-931971-28-7.

URL `https://www.usenix.org/conference/fast16/technical-sessions/presentation/harter`.

55. N. Hazekamp and D. Thain. Makeflow examples repository, 2017. URL `http://github.com/cooperative-computing-lab/makeflow-examples`.

56. N. Hazekamp and D. Thain. An Algebra for Robust Workflow Transformations. In *IEEE International Conference on e-Science*, page 12, 2018.

57. N. Hazekamp, O. Choudhury, S. Gesing, S. Emrich, and D. Thain. Poster: Expanding Tasks of Logical Workflows into Independent Workflows for Improved Scalability. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 548–549, 2014.

58. N. Hazekamp, J. Sarro, O. Choudhury, S. Gesing, S. Emrich, and D. Thain. Scaling Up Bioinformatics Workflows with Dynamic Job Expansion. In *IEEE International Conference on e-Science*, 2015.

59. N. Hazekamp, U. K. Devisetty, N. Merchant, and D. Thain. MAKER as a Service: Moving HPC applications to Jetstream Cloud. In *IEEE International Conference on Cloud Engineering*, page 6, 2018.

60. N. Hazekamp, N. Kremer-Herman, B. Tovar, H. Meng, O. Choudhury, S. Emrich, and D. Thain. Combining Static and Dynamic Storage Management for Data Intensive Scientific Workflows. *IEEE Transactions on Parallel and Distributed Systems*, 29(2):338–350, 2018.

61. N. Hazekamp, B. Tovar, and D. Thain. Dynamic Sizing of Continuously Divisible Jobs forHeterogeneous Resources. In *IEEE International Conference on e-Science*, 2019.

62. L. He, S. A. Jarvis, D. P. Spooner, and G. R. Nudd. Dynamic, capability-driven scheduling of dag-based real-time jobs in heterogeneous clusters, 2004.

63. R. L. Henderson. Job scheduling under the portable batch system. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, IPPS '95, pages 279–294, London, UK, UK, 1995. Springer-Verlag. ISBN 3-540-60153-8. URL `http://dl.acm.org/citation.cfm?id=646376.689372`.

64. B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1972457.1972488`.

65. R. Hoque, T. Herault, G. Bosilca, and J. Dongarra. Dynamic task discovery in parsec: A data-flow task-based runtime. In *Proceedings of the 8th Workshop*

on Latest Advances in Scalable Algorithms for Large-Scale Systems, ScalA '17, pages 6:1–6:8, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5125-6. doi: 10.1145/3148226.3148233. URL `http://doi.acm.org/10.1145/3148226.3148233`.

66. M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-636-3. doi: 10.1145/1272996.1273005. URL `http://doi.acm.org/10.1145/1272996.1273005`.

67. P. Ivie and D. Thain. PRUNE: A Preserving Run Environment for Reproducible Computing. In *IEEE Conference on e-Science*, 2016.

68. J. C. Jacob, D. S. Katz, G. B. Berriman, J. C. Good, A. C. Laity, E. Deelman, C. Kesselman, G. Singh, M. Su, T. A. Prince, and R. Williams. Montage&#58; a grid portal and software toolkit for science&#45;grade astronomical image mosaicking. *Int. J. Comput. Sci. Eng.*, 4(2):73–87, July 2009. ISSN 1742-7185. doi: 10.1504/IJCSE.2009.026999. URL `http://dx.doi.org/10.1504/IJCSE.2009.026999`.

69. M. A. Jette, A. B. Yoo, and M. Grondona. Slurm: Simple linux utility for resource management. In *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*, pages 44–60. Springer-Verlag, 2002.

70. G. Juve and E. Deelman. Resource provisioning options for large-scale scientific workflows. In *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, ESCIENCE '08, pages 608–613, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3535-7. doi: 10.1109/eScience.2008.160. URL `http://dx.doi.org/10.1109/eScience.2008.160`.

71. L. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

72. Q. Ke, V. Prabhakaran, Y. Xie, Y. Yu, J. Wu, and J. Yang. Optimizing data partitioning for data-parallel computing. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 13–13, Berkeley, CA, USA, 2011. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1991596.1991614`.

73. G. M. Kurtzer. Singularity 2.1.2 - Linux application and environment containers for science, Aug. 2016. URL `https://doi.org/10.5281/zenodo.60736`.

74. B. M. Kurtzer GM, Sochat V. Singularity: Scientific containers for mobility of compute. *PLoS ONE*, May 2017. URL `https://doi.org/10.1371/journal.pone.0177459`.

75. Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, Dec. 1999. ISSN 0360-0300. doi: 10.1145/344588.344618. URL `http://doi.acm.org/10.1145/344588.344618`.

76. J. Kster and S. Rahmann. Snakemakea scalable bioinformatics workflow engine. *Bioinformatics*, 28(19):2520–2522, 08 2012. ISSN 1367-4803. doi: 10.1093/bioinformatics/bts480. URL `https://doi.org/10.1093/bioinformatics/bts480`.

77. H. Li and R. Durbin. Fast and accurate short read alignment with burrows–wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

78. H. Li and R. Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, 26(5):589–595, Mar 2010.

79. H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, et al. The sequence alignment/map format and SAMtools. *Bioinformatics*, 25(16):2078–2079, 2009.

80. M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Eighth International Conference of Distributed Computing Systems*, June 1988.

81. F. Liu and J. B. Weissman. Elastic job bundling: An adaptive resource request strategy for large-scale parallel applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 33:1–33:12, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3723-6. doi: 10.1145/2807591.2807610. URL `http://doi.acm.org/10.1145/2807591.2807610`.

82. K. Liu, K. Aida, S. Yokoyama, and Y. Masatani. Flexible container-based computing platform on cloud for scientific workflows. In *2016 International Conference on Cloud Computing Research and Innovations (ICCCRI)*, pages 56–63, May 2016. doi: 10.1109/ICCCRI.2016.17.

83. T. Maeno, K. De, T. Wenaus, P. Nilsson, G. Stewart, R. Walker, A. Stradling, J. Caballero, M. Potekhin, and D. Smith. Overview of atlas panda workload management. *Journal of Physics: Conference Series*, 331, 02 2011. doi: 10.1088/1742-6596/331/7/072024.

84. T. Maeno, K. De, A. Klimentov, P. Nilsson, D. Oleynik, S. Panitkin, A. Petrosyan, J. Schovancova, A. Vaniachine, T. Wenaus, and D. Y. and. Evolution of the ATLAS PanDA workload management system for exascale computational science. *Journal of Physics: Conference Series*, 513(3):032062, jun

2014. doi: 10.1088/1742-6596/513/3/032062. URL `https://doi.org/10.1088%2F1742-6596%2F513%2F3%2F032062`.

85. K. Maheshwari, A. Rodriguez, D. Kelly, R. Madduri, J. Wozniak, M. Wilde, and I. Foster. Enabling multi-task computation on galaxy-based gateways using swift. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–3, Sept 2013. doi: 10.1109/CLUSTER.2013.6702701.

86. M. Malawski, G. Juve, E. Deelman, and J. Nabrzyski. Algorithms for cost- and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds. *Future Gener. Comput. Syst.*, 48(C):1–18, July 2015. ISSN 0167-739X. doi: 10.1016/j.future.2015.01.004. URL `http://dx.doi.org/10.1016/j.future.2015.01.004`.

87. H. Meng, R. Kommineni, Q. Pham, R. Gardner, T. Malik, and D. Thain. An invariant framework for conducting reproducible computational science. *Journal of Computational Science*, 9(0):137 – 142, 2015. ISSN 1877-7503. doi: http://dx.doi.org/10.1016/j.jocs.2015.04.012. URL `http://www.sciencedirect.com/science/article/pii/S1877750315000502`.

88. H. Meng, D. Thain, A. Vyushkov, M. Wolf, and A. Woodard. Conducting Reproducible Research with Umbrella: Tracking, Creating, and Preserving Execution Environments. In *IEEE Conference on e-Science*, 2016.

89. D. Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), Mar. 2014. ISSN 1075-3583. URL `http://dl.acm.org/citation.cfm?id=2600239.2600241`.

90. A. Merzky, M. Santcroos, M. Turilli, and S. Jha. Radical-pilot: Scalable execution of heterogeneous and dynamic workloads on supercomputers. *CoRR*, abs/1512.08194, 2015. URL `http://arxiv.org/abs/1512.08194`.

91. C. L. Monma, A. Schrijver, M. J. Todd, and V. K. Wei. Convex resource allocation problems on directed acyclic graphs: Duality, complexity, special cases, and extensions. *Mathematics of Operations Research*, 15(4):736–748, 1990. doi: 10.1287/moor.15.4.736. URL `https://doi.org/10.1287/moor.15.4.736`.

92. S. C. Müller, G. Alonso, A. Amara, and A. Csillaghy. Pydron: Semi-automatic parallelization for multi-core and the cloud. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 645–659, Broomfield, CO, Oct. 2014. USENIX Association. ISBN 978-1-931971-16-4. URL `https://www.usenix.org/conference/osdi14/technical-sessions/presentation/muller`.

93. H. Packard. Lustre: A scalable, high-performance file system. Technical report, Cluster File Systems, Inc., Novemeber 2002. URL `http://www.cse.buffalo.edu/faculty/tkosar/cse710/papers/lustre-whitepaper.pdf`.

94. R. Poplin, V. Ruano-Rubio, M. A. DePristo, T. J. Fennell, M. O. Carneiro, G. A. Van der Auwera, D. E. Kling, L. D. Gauthier, A. Levy-Moonshine, D. Roazen, K. Shakir, J. Thibault, S. Chandran, C. Whelan, M. Lek, S. Gabriel, M. J. Daly, B. Neale, D. G. MacArthur, and E. Banks. Scaling accurate genetic variant discovery to tens of thousands of samples. *bioRxiv*, 2018. doi: 10.1101/201178. URL `https://www.biorxiv.org/content/early/2018/07/24/201178`.

95. R. Priedhorsky and T. 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, pages 36:1–36:10, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5114-0. doi: 10.1145/3126908.3126925. URL `http://doi.acm.org/10.1145/3126908.3126925`.

96. U. Project. uproot. `https://github.com/scikit-hep/uproot`, 2018.

97. R. Raman, M. Livny, and M. Solomon. Matchmaking: distributed resource management for high throughput computing. In *Proceedings. The Seventh International Symposium on High Performance Distributed Computing (Cat. No.98TB100244)*, pages 140–146, Jul 1998. doi: 10.1109/HPDC.1998.709966.

98. I. Redhat. Ansible, 2012. URL `https://www.ansible.com/`.

99. R. Sakellariou and H. Zhao. A hybrid heuristic for dag scheduling on heterogeneous systems. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 111–, April 2004. doi: 10.1109/IPDPS.2004.1303065.

100. T. Shaffer, N. Kremer-Herman, and D. Thain. Flexible Partitioning of Scientific Workflows Using the JX Workflow Language. In *Practice and Experience in Advanced Research Computing (PEARC)*, 2019.

101. K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-7152-2. doi: 10.1109/MSST.2010.5496972. URL `http://dx.doi.org/10.1109/MSST.2010.5496972`.

102. G. Singh, C. Kesselman, and E. Deelman. Application-level resource provisioning on the grid. In *2006 Second IEEE International Conference on e-Science and Grid Computing (e-Science'06)*, pages 83–83, Dec 2006. doi: 10.1109/E-SCIENCE.2006.261167.

103. E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken. Regent: A high-productivity programming language for hpc with logical regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 81:1–81:12, New York, NY, USA, 2015.

ACM. ISBN 978-1-4503-3723-6. doi: 10.1145/2807591.2807629. URL `http://doi.acm.org/10.1145/2807591.2807629`.

104. S. Srinivasan, G. Juve, R. F. da Silva, K. Vahi, and E. Deelman. A cleanup algorithm for implementing storage constraints in scientific workflow executions. *9th Workshop on Workflows in Support of Large-Scale Science (WORKS)*, 2014.

105. C. A. Stewart, T. M. Cockerill, I. Foster, D. Hancock, N. Merchant, E. Skidmore, D. Stanzione, J. Taylor, S. Tuecke, G. Turner, M. Vaughn, and N. I. Gaffney. Jetstream: A self-provisioned, scalable science and engineering cloud environment. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, XSEDE '15, pages 29:1–29:8, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3720-5. doi: 10.1145/2792745.2792774. URL `http://doi.acm.org/10.1145/2792745.2792774`.

106. K. Sweeney and D. Thain. Efficient Integration of Containers into Scientific Workflows. In *Science Cloud Workshop at HPDC*, 2018.

107. D. Thain, T. Tannenbaum, and M. Livny. Condor and the grid. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley, 2003.

108. D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

109. A. Thrasher, Z. Musgrave, B. Kachmark, D. Thain, and S. Emrich. Scaling Up Genome Annotation with MAKER and Work Queue. *International Journal of Bioinformatics Research and Applications*, 10(4-5):447–460, 2014.

110. H. Topcuouglu, S. Hariri, and M.-y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):260–274, Mar. 2002. ISSN 1045-9219. doi: 10.1109/71.993206. URL `http://dx.doi.org/10.1109/71.993206`.

111. B. Tovar, N. Hazekamp, N. Kremer-Herman, and D. Thain. Automatic Dependency Management for Scientific Applications on Clusters. In *IEEE International Conference on Cloud Engineering (IC2E)* , 2018.

112. J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, R. Roskies, J. R. Scott, and N. Wilkins-Diehr. Xsede: Accelerating scientific discovery. *Computing in Science Engineering*, 16(5):62–74, Sept 2014. ISSN 1521-9615. doi: 10.1109/MCSE.2014.80.

113. R. Tudoran, A. Costan, and G. Antoniu. Overflow: Multi-site aware big data management for scientific workflows on clouds. *IEEE Transactions on Cloud Computing*, 4(1):76–89, Jan 2016. doi: 10.1109/TCC.2015.2440254.

114. D. Turi, P. Missier, C. Goble, D. D. Roure, and T. Oinn. Taverna workflows: Syntax and semantics. In *Third IEEE International Conference on e-Science and Grid Computing (e-Science 2007)*, pages 441–448, Dec 2007. doi: 10.1109/E-SCIENCE.2007.71.

115. K. Vahi, M. Rynge, G. Juve, R. Mayani, and E. Deelman. Rethinking data management for big data scientific workflows. In *2013 IEEE International Conference on Big Data*, pages 27–35, Oct 2013. doi: 10.1109/BigData.2013.6691724.

116. L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990. ISSN 0001-0782. doi: 10.1145/79173.79181. URL `http://doi.acm.org/10.1145/79173.79181`.

117. G. J. Voss K and V. der Auwera G. Full-stack genomics pipelining with gatk4 + wdl + cromwell, 2017. URL `https://doi.org/10.7490/f1000research.1114631.1`.

118. J. Wang, D. Crawl, I. Altintas, and W. Li. Big data applications using workflows for data parallel computing. *Computing in Science Engineering*, 16(4):11–21, July 2014. ISSN 1521-9615. doi: 10.1109/MCSE.2014.50.

119. S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1. URL `http://dl.acm.org/citation.cfm?id=1298455.1298485`.

120. B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 2:1–2:17, Berkeley, CA, USA, 2008. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1364813.1364815`.

121. M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9): 633–652, 2011.

122. K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, J. Bhagat, K. Belhajjame, F. Bacall, A. Hardisty, A. Nieva de la Hidalga, M. P. Balcazar Vargas, S. Sufi, and C. Goble. The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic Acids Research*, 41 (W1):W557, 2013. doi: 10.1093/nar/gkt328. URL `+http://dx.doi.org/10.1093/nar/gkt328`.

123. K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, J. Bhagat, K. Belhaj-

jame, F. Bacall, A. Hardisty, A. Nieva de la Hidalga, M. P. Balcazar Vargas, S. Sufi, and C. Goble. The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic Acids Research*, 41(W1):W557–W561, 2013. doi: 10.1093/nar/gkt328. URL `http://nar.oxfordjournals.org/content/41/W1/W557.abstract`.

124. A. Woodard, M. Wolf, C. Mueller, N. Valls, B. Tovar, P. Donnelly, P. Ivie, K. H. Anampa, P. Brenner, D. Thain, K. Lannon, and M. Hildreth. Scaling Data Intensive Physics Applications to 10k Cores on Non-Dedicated Clusters with Lobster. In *IEEE Conference on Cluster Computing*, 2015.

125. W. Wu, A. Bouteiller, G. Bosilca, M. Faverge, and J. Dongarra. Hierarchical dag scheduling for hybrid distributed systems. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 156–165, May 2015. doi: 10.1109/IPDPS.2015.56.

126. M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1863103.1863113`.

127. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

128. M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, Oct. 2016. ISSN 0001-0782. doi: 10.1145/2934664. URL `http://doi.acm.org/10.1145/2934664`.

129. Y. Zhao, J. Dobson, L. Moreau, I. Foster, and M. Wilde. A notation and system for expressing and executing cleanly typed workflows on messy scientific data. In *SIGMOD*, 2005.

130. C. C. Zheng and D. Thain. Integrating Containers into Workflows: A Case Study Using Makeflow, Work Queue, and Docker. In *Workshop on Virtualization Technologies in Distributed Computing (VTDC)*, 2015.

This document was prepared & typeset with pdfLaTeX*, and formatted with* NDdiss2$_\varepsilon$ *classfile (v3.2017.2[2017/05/09])*