Dynamic Sizing of Continuously Divisible Jobs for Heterogeneous Resources

Nicholas Hazekamp
University of Notre Dame
Notre Dame, Indiana 46556
nhazekam@nd.edu

Benjamin Tovar
University of Notre Dame
Notre Dame, Indiana 46556
btovar@nd.edu

Douglas Thain
University of Notre Dame
Notre Dame, Indiana 46556
dthain@nd.edu

Abstract—**Many scientific applications operate on large datasets that can be partitioned and operated on concurrently. The existing approaches for concurrent execution generally rely on statically partitioned data. This static partitioning can lock performance in a sub-optimal configuration, leading to higher execution time and an inability to respond to dynamic resources.**

We present the Continuously Divisible Job abstraction which allows statically defined applications to have their component tasks dynamically sized responding to system behaviour. The Continuously Divisible Job abstraction defines a simple interface that dictates how work can be recursively divided, executed, and merged. Implementing this abstraction allows scientific applications to leverage dynamic job coordinators for execution. We also propose the Virtual File abstraction which allows read-only subsets of large files to be treated as separate files.

In exploring the Continuously Divisible Job abstraction, two applications were implemented using the Continuously Divisible Job interface: a bioinformatics application and a high-energy physics event analysis. These were tested using an abstract job interface and several job coordinators. Comparing these against a previous static partitioning implementation we show comparable or better performance without having to make static decisions or implement complex dynamic application handling.

I. Introduction

Many scientific applications consist of large datasets that can be decomposed for concurrent execution. Fields such as bioinformatics, high energy physics, and astrophysics leverage concurrent execution to decompose and analyze data of all scales. Most of these applications have consistent analysis steps, particularly in pre-processing, which allows for clean mapping to any size data. In this context, the methods for partitioning large datasets into smaller and more manageable components are well understood, which allows data analysis to be described using common templates or frameworks. The prevalence of concurrent execution is such that new technologies are regularly developed and execution is deployed across many different compute sites.

The proliferation of concurrent approaches has lead to a wide variety of models, methods, and techniques that application developers can leverage for execution and scale. 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.

Bag-of-tasks approaches create a set of jobs that are passed to the execution system. The common approach for specifying bag-of-tasks applications predicates that the partitions are already defined when submitted. The static nature of the partitioning limits the ability of the execution system to influence the size and performance of partitions. This static definition either locks the applications performance or requires a more complex application that uses feedback to adjust the partitions. In many concurrent approaches the cost of creating partitions is high, further increasing performance overhead.

In this paper, we 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.

To show the Continuously Divisible Job abstraction we examine the performance of two applications, BWA for genome sequence alignment and a high-energy physics event analysis for detecting dimuon candidates. Using BWA, results were compared between a static bag-of-tasks approach and as a

Continuously Divisible Job application. These were run locally and using master-worker framework for distributed execution. We compare the scaling and execution time, showing that in the good configurations the Continuously Divisible Job implementation is comparable in performance, and in bad configurations the Continuously Divisible Job implementation can adjust sizing to improve performance. For the dimuon detection we compared different methods of accessing the complex input format as a virtual file.

II. Contributions

Our contributions in the paper are:

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

III. Background and 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 [1], PBS, and Torque, or high-throughput computing such as HTCondor [2], 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 [3], bulk synchronous parallel [4], 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 [5] 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 [6], [7] , 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 [8] and Pegasus [9], both of which rely on static directed acyclic graphs. This allows for simple workflow design and templating, but has limited flexibility for runtime adjustments. Similarly, workflows can be defined more generally using the Common Workflow Language [10] or the Workflow Description Language [11]. The goal of these systems is to write a workflow once and use different engines or systems to execute. This is similar to the API and job drivers that will be introduced, but lacks the ability to create a feedback loop for sizing. 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 1 we show how the total runtime of BWA is influenced by the task size on a fixed dataset.

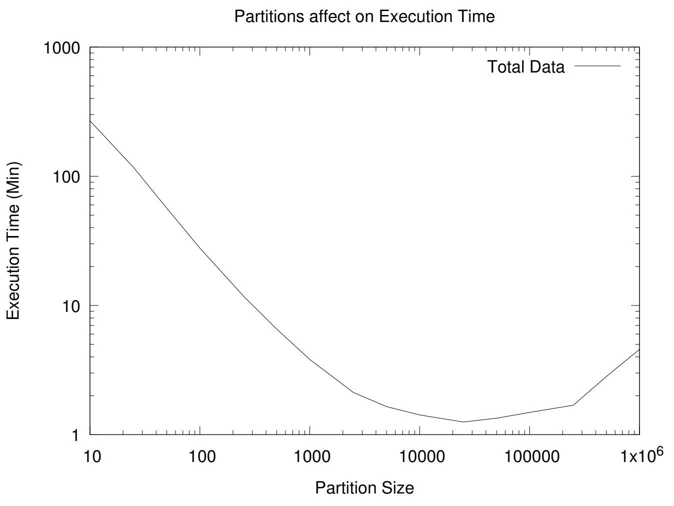

Fig. 1. A standard BWA bioinformatics workflow's performance with the input partition size varied. The partition size was varied from the dataset seize of 1,000,000 sequences down to 10. The overhead introduced when using partitions of only a single sequence was so large that the application consistently failed to complete.

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 include Work Queue [12], RADICAL Cybertools [13], Swift [14], and Parsl [15] allow for simple dynamic task definitions, but decisions about sizing and task handling are still largely user burdens. Similar to dynamic workflows are task based systems such as Charm++ [16] that allow low level control, but need more management. The challenge is that these approaches require both knowledge of the application 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.

IV. 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, greatly increasing the execution time and resource needs.

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.

A. 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 V 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.

TO_DESC(JOB)::DESCRIPTION
FROM_DESC(DESCRIPTION)::JOB
The `TO_DESC` and `FROM_DESC` define the basics for seri-

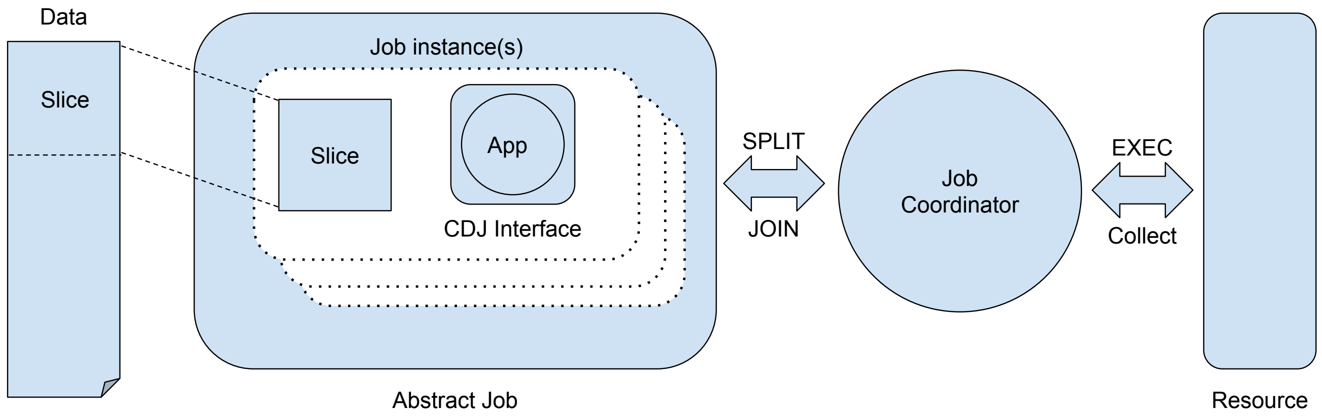

Fig. 2. This diagram outlines the relationships between the data slice, Continuously Divisible Job interface, abstract jobs, and the job coordinator. An abstract job consists of a data slice, the applications, and the interface wrapper. Abstract jobs may map a single slice, or contain several independent. 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.

alizing 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 2

B. 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 (un-executed, 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.

C. 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 drivers 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 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 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.

D. Design Considerations

In developing the application operation for Continuously Divisible Job interface, there are several design considerations

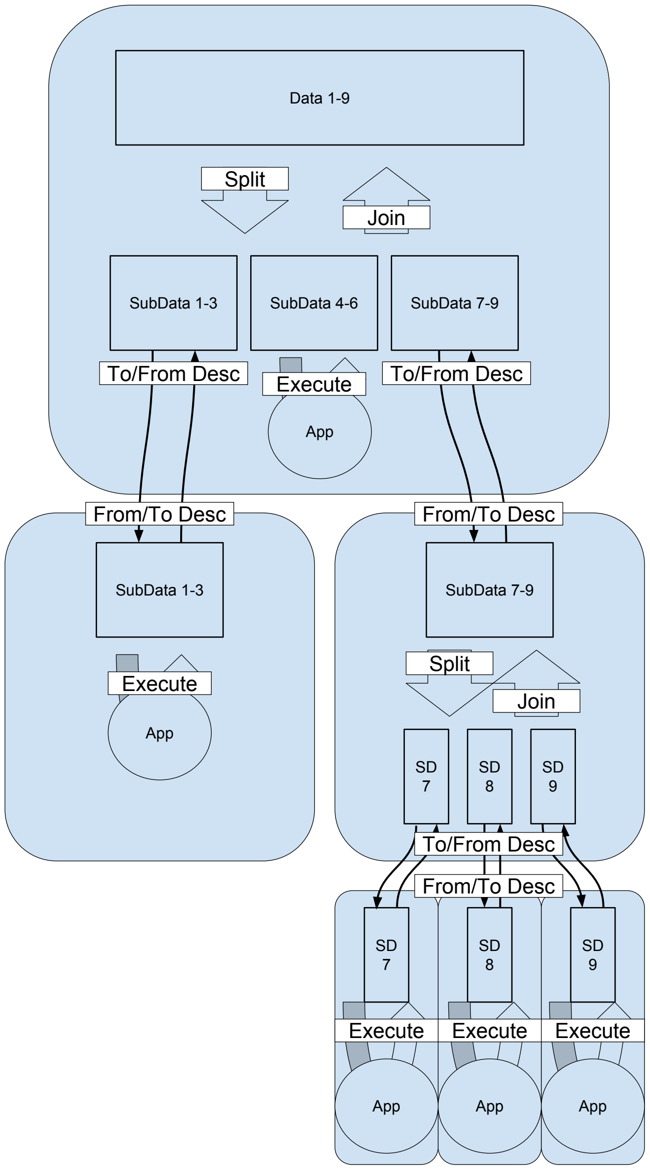

Fig. 3. Capabilities of Continuously Divisible Job abstraction. Using the Continuously Divisible Job abstraction, jobs can be partitions and run locally. Using the same design we can also distribute to multicore workers or to other job coordinators that further distribute the work. This highlights how the Continuously Divisible Job interface relates to overall recursive design.

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 larger considerations along with designs and methods to resolve them.

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 paper, 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 V, 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.

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.

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 [17], [18], sandboxes, and wrappers [19] to isolate each application instance.

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

V. 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.

A. 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.

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.

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.

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.

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.

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.

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

A. Example Continuously Divisible Job Application

For this paper, 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 concurrent event analysis and

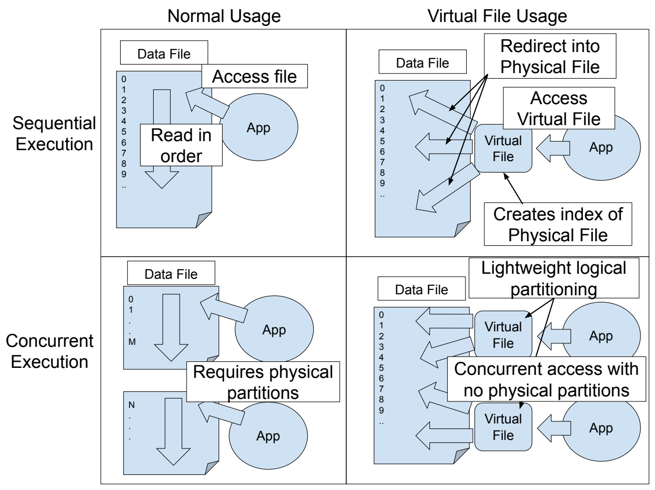
Normal Usage
Virtual File Usage
Sequential Execution
Concurrent Execution
Data File
Access file
Read in order
App
Redirect into Physical File
Access Virtual File
Virtual File
Creates index of Physical File
Requires physical partitions
Lightweight logical partitioning
Concurrent access with no physical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

Fig. 4. Comparing the file usage of a 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.

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 were 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 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,

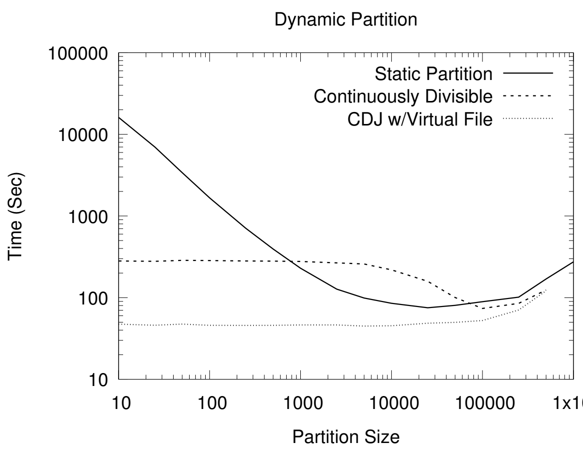

Fig. 5. Comparing the Static Makeflow partitions against both the base Continuously Divisible Job implementation and Continuously Divisible Job using Virtual Files. Jobs are partitioned as previous work is finished allowing the size to dynamically find a stable partition, and all of the cores remain 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.

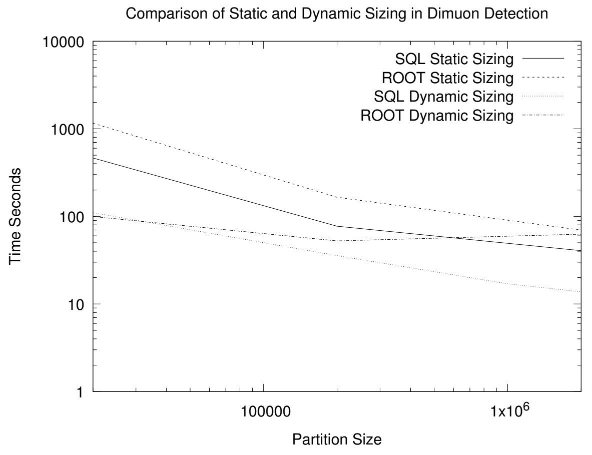

Fig. 6. Comparing the ROOT and SQL implementation of a dimuon detection analysis. To evaluate both the benefit of the virtual file and dynamic sizing, each implementation was run using static partitions and then dynamic partitions. We can see in the static case, the SQL implementation provides a consistent 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.

which in turn are grouped into events, luminosity sections, and so on, which are stored in ROOT files [20].

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 [21] 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.

B. 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 sub-data, limiting both the space needed to operate on sub-data and the time needed to partition.

In the second case, a structure is needed to represent the data as partitions are defined and moved around. Using Work Queue [12], 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.

VII. 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 paper 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.

A. 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 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 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.

B. Virtual File Effectiveness

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, 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 6).

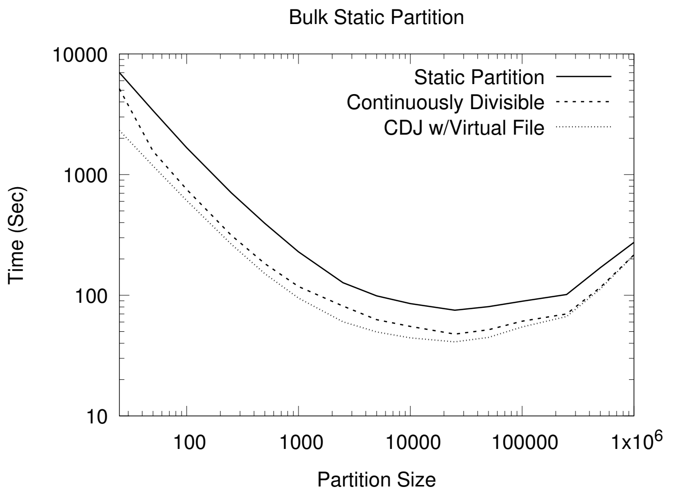

Fig. 7. This shows a standard BWA bioinformatics workflows where only the size of each partition is varied. This is compared with similarly static partitioned Continuously Divisible Job implementations. We can see that lightweight partitioning and dynamic joining help them out perform the fully static case.

C. 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 5. In this graph we are comparing again against the static partitioning. Each of the subsequent lines is grouped by the initial master partition 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 exploration is needed to understand how to avoid negative feedback.

VIII. Conclusion

In this paper we introduced the concept of Continuously Divisible Jobs and discussed how dynamic sizing can be used to address limitations of static partitioning. We discuss how implementing the Continuously Divisible Job interface allows applications to be dynamically partitioned, executed, and distributed to dynamic resource 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

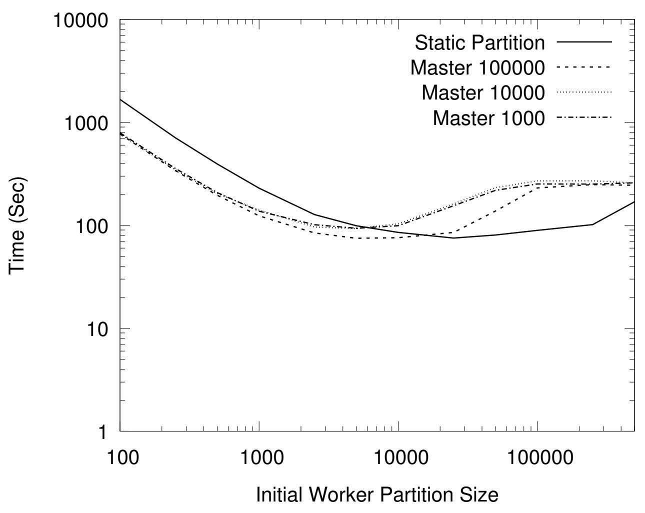

Fig. 8. This compares the Static partitions against several configuration of Continuously Divisible Job using Virtual Files. Each line is grouped by the initial master size and the X axis shows the initial worker size. As can be seen across each case, at the right the limited worker partitioning under-utilizes the cores. To the left, the small starting size suffers from initial splitting overhead.

redundant reads and writes. Combining these concepts, we implement Continuously Divisible Jobs using Python, and provided two example applications. This implementation was executed, comparing the results against static partitioning approach to show in good configuration similar performance and in bad configuration a significant improvement. Using these results, this work will be further explored to better understand applications that can benefit from Continuously Divisible Job and how this approach can be improved.

IX. Reproducibility Data

In an effort to provide consistent, reproducible results here are outlined the resources utilized in this paper and where they can be found. If specific commits are mentioned to provide the exact version that was used. All of these repositories are open source and contain Makefiles and instructions on how to build and run them.

The paper, prototypes of Continuously Divisible Job, and the example application implementations: `https://github.com/nhazekam/partitioning-scalability`

BWA, which was forked and modified for the use in this paper: `https://github.com/nhazekam/bwa/tree/add_offset_limits`

Makeflow and Work Queue, which were used as the baseline and for remote execution respectively: `https://github.com/cooperative-computing-lab/cctools`

References

[1] 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*. Springer-Verlag, 2002, pp. 44–60.

[2] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the condor experience." *Concurrency - Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005.

[3] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251264

[4] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990. [Online]. Available: http://doi.acm.org/10.1145/79173.79181

[5] 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)*, ser. MSST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/MSST.2010.5496972

[6] 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*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113

[7] 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*, vol. 59, no. 11, pp. 56–65, Oct. 2016. [Online]. Available: http://doi.acm.org/10.1145/2934664

[8] 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.

[9] 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. Kat, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming Journal*, vol. 13, no. 3, 200.

[10] 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. [Online]. Available: https://figshare.com/articles/Common_Workflow_Language_draft_3/3115156/2

[11] G. J. Voss K and V. der Auwera G, "Full-stack genomics pipelining with gatk4 + wdl + cromwell," 2017. [Online]. Available: https://doi.org/10.7490/f1000research.1114631.1

[12] 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.

[13] A. Merzky, M. Santcroos, M. Turilli, and S. Jha, "Radical-pilot: Scalable execution of heterogeneous and dynamic workloads on supercomputers," *CoRR*, vol. abs/1512.08194, 2015. [Online]. Available: http://arxiv.org/abs/1512.08194

[14] 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.

[15] Y. Babuji, A. Brizius, K. Chard, I. Foster, D. S. Katz, M. Wilde, and J. Wozniak, "Introducing Parsl: A Python Parallel Scripting Library," Aug. 2017. [Online]. Available: https://doi.org/10.5281/zenodo.891533

[16] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of OOPSLA'93*, A. Paepcke, Ed. ACM Press, September 1993, pp. 91–108.

[17] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241

[18] B. M. Kurtzer GM, Sochat V, "Singularity: Scientific containers for mobility of compute," *PLoS ONE*, May 2017. [Online]. Available: https://doi.org/10.1371/journal.pone.0177459

[19] N. Hazekamp and D. Thain, "An Algebra for Robust Workflow Transformations," in *IEEE International Conference on e-Science*, 2018, p. 12.

[20] 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*, vol. 389, pp. 81–86, 04 1997.

[21] "uproot," https://github.com/scikit-hep/uproot.