University of Alberta

JOLE: A LIBRARY FOR DYNAMIC JOB-LEVEL PARALLEL WORKLOADS

by

Jordan Dacey Lee Patterson

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Fall 2009
Edmonton, Alberta

Examining Committee

Paul Lu, Computing Science

Mike Carbonaro, Educational Psychology

Mario Nascimento, Computing Science

Abstract

Problems in scientific computing often consist of a workload of jobs with dependencies between them. Batch schedulers are job-oriented, and are not well-suited to executing these workloads with complex dependencies.

We introduce Jole, a Python library created to run these workloads. Jole has three contributions that allow flexibility not possible with a batch scheduler. First, dynamic job execution allows control and monitoring of jobs as they are running. Second, dynamic workload specification allows the creation of workloads that can adjust their execution while running. Lastly, dynamic infrastructure aggregation allows workloads to take advantage of additional resources as they become available.

We evaluate Jole using GAFolder, a protein structure prediction tool. We show that our contributions can be used to create GAFolder workloads that use less cluster resources, iterate on global protein structures, and take advantage of additional cluster resources to search more thoroughly.

Acknowledgements

I would like to thank my supervisor, Paul Lu, for his support and guidance during my work on this research. In addition, I would like to thank both the Proteome Analyst and Trellis research groups. Working with the Proteome Analyst group introduced me to research and motivated me to pursue further studies. Our Trellis group discussions have been enjoyable and have exposed me to a range of interesting research. As well, I would like to thank my family, for their support throughout the years. Finally, I thank my wife, Amanda, whose love and support has made this possible.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computing science has become important for research in many scientific disciplines. The vast improvement in computing hardware has enabled scientists to study problems and perform simulations in much greater detail. In biology, computers are used to predict protein structure, find similar proteins, and perform cell simulations. In physics, they can be used for simulating the effects of solar flares or fluid dynamics. In computing science, computers may be used to train machine-learned classifiers, artificial intelligence agents, or poker strategies. For many disciplines, having access to computational resources is a requirement for further research.

Many of these problem domains are well-suited to running on clusters of commodity computers. Some are naturally parallel and others may have high granularity due to a large problem size. With the capabilities of commodity hardware increasing while their associated prices decline, it has become more attractive to solve problems using many of these commodity computers, leading to capacity computing. Capacity computing focuses on maximizing the throughput of several jobs. Commodity clusters cannot solve individual problems as fast as more expensive machines, but they can solve many of them at a time. As an example, many of Google's computations are run on commodity clusters [2]. MapReduce [5] and Google File System (GFS) [6] have been created to take advantage of these commodity nodes for computation and storage.

Other problems may benefit more from running single jobs as fast as possible. Capability computing is focused on this problem. This generally means building computers with the best processors, memory, hard drives, and interconnects. While useful for many problems, capability computing resources do not offer the best price/performance ratio for problems that are well-suited to capacity computing resources, and would not benefit from Jole.

With the use of capacity computing resources, tools for running jobs have become more important. Rather than running single jobs, users generally run workloads consisting of several jobs with dependencies between them. Trying to run a user's workload interactively can be an exercise in frustration. Although it is common for users to run jobs interactively, they could save a lot of their time by using an automated tool to run their jobs for them when they are running large workloads. An automated tool can make scheduling decisions much faster than a user.

The main tool for running jobs on commodity clusters has been batch schedulers, such as TORQUE [18] or OpenPBS [20]. There are several properties of workloads in scientific computing that make them well-suited to running under batch scheduled resources. First, the jobs within a workload are non-interactive. All application options are specified by command-line arguments, environment variables, or configuration files. This means that the jobs can be run without user intervention. Second, these workloads generally consist of many jobs. For many applications, there is very little to no communication between jobs. This means that they can run efficiently over many machines in a distributed fashion. Third, the jobs within workloads consist mainly of long-running jobs. This means that scheduling overhead is less of a problem, as the jobs will have higher granularity in general.

Batch schedulers are an improvement over running jobs interactively on each node. With batch schedulers, users are able to specify large amounts of jobs, and let the scheduler decide when and where to run them. Batch schedulers are usually configured to maximize throughput, completing as much work as possible in the shortest amount of time. Users can specify the resources that their jobs need and let the scheduler handle placement of jobs on the cluster. Users just need to write scripts or workflow specifications that describe their workload, and submit them to the scheduler.

Although batch schedulers are an improvement over running jobs manually, there are some ways in which running jobs on the cluster could be improved. For example, job dependencies may not be well supported on some batch schedulers, making it hard to properly run a workflow. With batch schedulers that do support job dependencies, the job dependencies are generally static, and must be rewritten for each workflow that is submitted. For other batch schedulers, users may need to ensure that job dependencies are met by writing scripts that monitor their jobs independently of the batch scheduler.

Dependencies between jobs are common in workloads. GAFolder, a protein structure prediction tool, is an individual application, but it is possible to run multiple instances in several different ways to improve prediction results or to decrease run time. In the dynamic local and global workflow, described in Section 5.3, groups of GAFolder instances are run iteratively. The globally best protein structures are used as a basis for the GAFolder instances in each generation. This workload is shown in Figure 1.1. Note that this workflow is also shown in Figure 5.5. In the evaluation of this workflow, we are able to produce an improvement in score from -33.32 with the static workflow to -33.52 with the dynamic local and global workflow. This dynamic iteration of GAFolder instances would be difficult to implement in a static workflow language. The best scores must be determined in each generation during run time and subsequent jobs running GAFolder must be created that use the structures with the best scores.

Another improvement that could be made over batch schedulers is the ability to get more information about jobs and interact with them as they are running. Users may want to know more about a jobs execution than whether it is running or not. For example, users may want to know about the

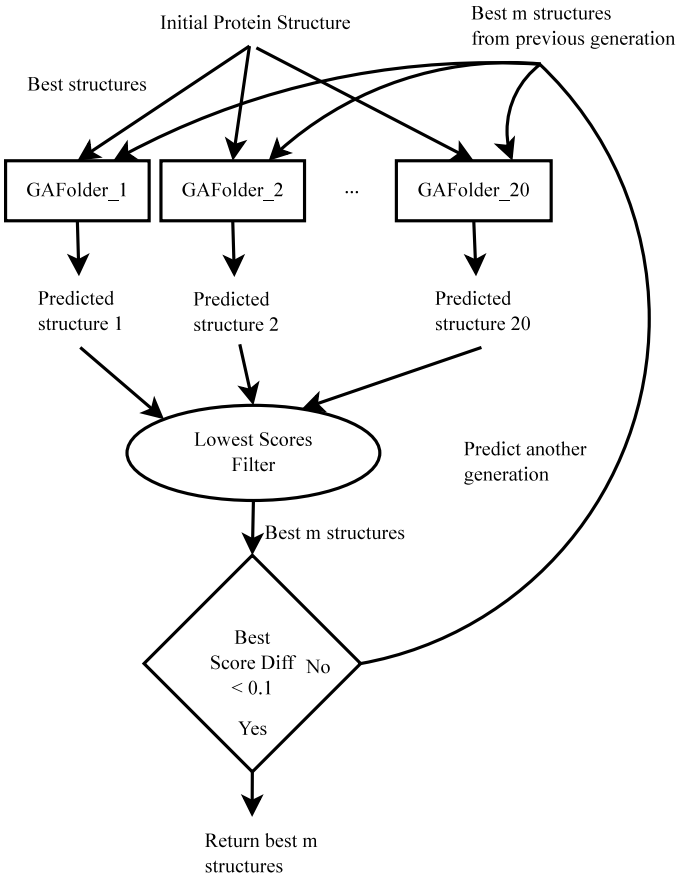

Figure 1.1: GAFolder: Dynamic Local and Global Workflow

progress being made in a job. If the job is progressing poorly or has reached a plateau in search, they may want to automatically terminate it. With batch schedulers, they must determine which node a job is executing on, login to that node, and look at output or other information in order to do this. This is too much administrative work for what may be a relatively simple task they wish to perform on their job as it is running.

Another improvement that could be made over batch schedulers is the ability for workloads to react to changes in available resources. Many clusters may exist within an organization, which may be idle much of the time. Users could benefit by taking advantage of these clusters when they are idle. For example, they may want to run additional instances of a job when there are additional resources available to run jobs on. For workloads involving search, this allows their workloads to perform a more exhaustive search when there are more available resources.

Our work is focused on creating a workflow execution library, Jole, that improves the flexibility of running workloads on clusters. With the features of our library, we can improve upon the workload support of batch schedulers.

1.1 Research Challenges and Contributions

Our main research challenges and contributions are as follows:

1. *Dynamic Job Execution*

   Not all batch jobs need to run to completion. Depending on the application or algorithm, some batch jobs should be terminated early or new batch jobs should be launched, based on partial results of other batch jobs. With a regular batch scheduler and workflow language, this level of job control is not possible. Therefore, support for dynamically controlling jobs while they are executing can be beneficial.

   We introduce monitors, which can interact with jobs as they are running. There may be some criteria by which users want to terminate jobs. A monitor can be used to interact with the job, and once it meets the criteria, terminate it.

   This is an improvement over current batch scheduling because it is now possible to interact with jobs while they are running. With a normal batch scheduler, users normally have to wait until the job is done to get their results. With monitors, it is possible to watch the output of jobs and end them early if the results are not satisfactory.

   For example, each run of GAFolder uses a random seed, producing different results on each run. Some runs will reach a local minima before reaching the standard number of iterations. It can be beneficial to end those runs early and start new ones in their place, in order to get more benefit from the computation.

2. *Dynamic Workflow*

Many workflows have complicated control and dataflow dependencies between jobs. Specifying this information in a static language can be cumbersome, especially if there is varied input data and control-flow. Some types of control-flow may not be possible, such as iterations, in systems which only support directed acyclic graphs (DAGs) for workflow specification. With a static specification language, each small difference in workflow may require a separate submission script. Therefore, a dynamic scripting-based approach to specification can be beneficial.

Data handles, job futures, and job arrays enable a dynamic scripting based-approach to workflow specification. Each of these features can be used to synchronize on job status. Combined with the use of the Python scripting language for Jole, these features allow users to synchronize on jobs, gather data globally about jobs, and adjust the workflow based on the global data.

These improvements help in several ways. First, it is possible to script a workflow which modifies itself based on global data. This allows workflows to iterate based on the results of the intermediate computations, which would be difficult to perform with a static workflow specification language. Second, the use of a scripting language allows parametrization of the workflow script, allowing reuse for different workflows. Lastly, data handles prevent the need to handle file conflicts in output files between workflows.

For example, several instances of GAFolder are run in order to produce structure predictions. New GAFolder instances can be run using the most promising structures globally in order to further refine the structure. This process can be repeated several times, in order to focus search around the most promising proteins. This cyclic graph would be hard to implement using a static language, but can be implemented using Jole.

3. *Dynamic Infrastructure*

   Within an organization, there may be several clusters that are not heavily used by their owners. These resources could be used by others when they are idle. Users with their own clusters will also benefit from this sharing, because they can now spread their jobs across other clusters when they are idle. It would be beneficial to automatically take advantage of these resources when they become available. A workflow may be modified as it is running, to better take advantage of additional resources.

   Submitters enable users to run workflows across clusters. The job manager starts and interacts with submitters on each cluster in order to run jobs. The user specifies the machines they wish to run submitters in their workflow script. A workflow can get resource information from the job manager as it is running, and use that information to decide how to continue running the workload.

   For some workloads, adjusting the workflow graph based on the available resources can help

improve results. With some search applications, it can be possible to generate better results by running more instances of the program when more resources are available. If each instance of the search is independent and searches a different space, more instances will allow a more thorough search.

For example, in the dynamic local and global workload for GAFolder, a set number of instances are run in each generation. By running additional GAFolder instances in each generation, it is possible to perform a more complete search in the protein space and produce better quality structure predictions.

Chapter 2

Background

In the previous chapter, we described the importance of workflows to scientific computing and described several improvements that could be implemented on top of batch schedulers. We introduced and described some of the contributions of Jole for running workflows.

In this chapter, we describe some work related to Jole, in the parallel language and batch scheduling fields. We then describe some concepts that are used in Jole. Finally, we describe some of the motivating applications that are used in the following chapters.

2.1 Related Work

2.1.1 Parallel and Distributed Languages

Parallel and distributed languages are targeted towards performing computation using several processor cores or machines. Data-parallel languages, such as MapReduce, are becoming more popular, as they enable a natural way of parallelizing data analysis and computation. Implementations of the Message Passing Interface (MPI), remain popular for scientific computing, as they may have communication patterns that do not map well to data-parallel languages.

While there is a wealth of research towards distributed and data-parallel languages, there is less targeted towards batch-scheduled job-level parallelism. However, there is an overlap in ideas between the two. Table 2.1 compares the main properties of each system. The types of parallelism and fault tolerance supported in each language are shown. Process placement refers to the ability to place processes near the data they will use on a distributed architecture.

System	Parallelism	Fault Tolerance	Process Placement
Jole	Task	Yes (Jobs)	Manual
MapReduce	Data	Yes (Tasks)	Automatic
Dryad	Data	Yes (Vertices)	Automatic
MPI	Task & Data	Manual	Manual

Table 2.1: Comparison of Jole to Data Parallel Languages

MapReduce

MapReduce [5] is a programming model and framework for data-parallel programming. Although Google has not released their system to the public, an open-source implementation of MapReduce called Hadoop [10] has become popular. Users specify a map function which takes key/value pairs and processes them to produce intermediate key/value pairs. These new key/value pairs are merged by a user specified reduce function. This programming model is somewhat limited, however there are many programs that are well-suited to running under this model. Some examples include counting uniform resource locator (URL) access frequencies from logs, counting term frequencies in documents, and creating inverted indexes.

Google uses MapReduce to perform computation on a variety of problems, including large machine-learning problems, clustering problems, extracting properties from web pages, and large graph computations. One of the most significant uses of MapReduce is Google's production indexing system, which produces the data structure required for Google's search engine. These computations are performed on many terabytes of data in five to ten MapReduce operations. One step performed in this process is the creation of an inverted index, which maps words to documents containing the word. This inverted index is used for web searches to find documents containing query words. Additionally, MapReduce is used for the calculations in determining PageRank [21]. Roughly, PageRank measures importance of web pages by looking on incoming links to a page and the PageRanks of those links.

The following pseudocode in Figure 2.1, from Dean and Ghemawat [5], illustrates the use of MapReduce to count word frequencies in a set of documents.

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
    Emit(AsString(result));
```

Figure 2.1: MapReduce pseudocode for counting word frequencies

The map phase uses the map function in Figure 2.1, which iterates through each word in the document and emits an intermediate key/value pair for each word with the word as the key and 1 as the value. The reduce phase uses the reduce function in Figure 2.1, which iterates through the

intermediate values for a key produced in the map phase, summing the word count for a word. This reduce phase will be performed on every key.

The MapReduce system automatically parallelizes the map phase by partitioning the input data into `M` chunks. These chunks can be processed in parallel by independent map tasks. The output from map tasks is partitioned into `R` chunks by a partitioning function, where each partition is processed by a separate reduce task. The number of partitions must be specified by the user. The partitioning function can be specified by the user, if the user prefers to partition the data differently. When the computation is close to finishing, redundant executions of in-progress tasks will be scheduled. This prevents stragglers, tasks running on slow or problematic machines, from slowing down the entire MapReduce operation.

MapReduce handles scheduling of these tasks and can recover from machine failures. Due to the use of the Google File System (GFS) [6], the data is replicated across multiple machines. If a machine fails, the input data will be on other nodes. GFS will re-replicate this data to ensure that the minimum replication criteria is met. A master process controls the execution of the MapReduce program. Workers are periodically pinged to check for failure. If a worker fails, the master process will restart its task.

MapReduce is somewhat limited in the types of dataflow programs it can run, but there are many problems that are well-suited to this model. Mapping jobs to this model may be more difficult. The simplest case for executing most jobs is to use the map phase to execute the job, while having a trivial reduce phase which merely collects the output. For example, the basic local alignment search tool (BLAST) [1] could be parallelized using MapReduce by using the set of query proteins as input to the map phase, which would be partitioned between map tasks performing BLAST. The reduce phase could simply emit the BLAST results from the map phase, or perhaps do some filtering based on score. Due to MapReduce's use of chunking to achieve parallelism, it may be necessary to adjust the chunking size in order to get sufficient concurrency. For example, GAFolder consists of long running jobs with a single input protein. The chunk size would have to be small to run a GAFolder workflow on MapReduce. MapReduce is more suited towards running large problems, where this would not be necessary.

Running jobs with MapReduce has several benefits. With sufficient input data and proper chunk size, MapReduce achieves good parallelism. Jobs run with Jole will likely achieve similar parallelism, although the user would have to split their data before creating jobs, rather than having the system do this automatically. MapReduce is able to recover from machine failures by monitoring tasks on each machine. If a machine fails, it will rerun those tasks. Our job manager is also able to recover from machine failures of placeholders. If a machine fails, the placeholder will no longer be sending messages to the job manager, so the job manager will start a new job to replace it.

Although it may be possible to map some workflows to MapReduce, it can result in a longer makespan, in addition to extra work fitting jobs into the MapReduce paradigm. MapReduce does

not support dynamic job execution, meaning that all jobs will run until finished. With Jole, jobs can be monitored and ended early if they do not meet some criteria.

Dryad

Dryad is a distributed data parallel framework from Microsoft [11]. Users specify a dataflow graph of processes and code segments within the framework. The vertices in the graphs represent the computations, while the edges specify the data channels. Dryad's job manager controls the execution of these vertices and builds an execution graph for running the workflow on the cluster. Dryad has a series of operations that must be used to create a dataflow graph. Graphs can be created by instantiating vertices, adding edges, and merging graphs.

The following example, from Isard *et al.* [11], illustrates the creation of a graph with Dryad, producing the graph shown in Figure 2.3.

```
GraphBuilder XSet = moduleX^N;
GraphBuilder DSet = moduleD^N;
GraphBuilder MSet = moduleM^(N*4);
GraphBuilder SSet = moduleS^(N*4);
GraphBuilder YSet = moduleY^N;
GraphBuilder HSet = moduleH^1;
GraphBuilder XInputs = (ugriz1 >= XSet) || (neighbor >= XSet);
GraphBuilder YInputs = ugriz2 >= YSet;
GraphBuilder XToY = XSet >= DSet >> MSet >= SSet;
for (i = 0; i < N*4; ++i)
{
      XToY = XToY || (SSet.GetVertex(i) >= YSet.GetVertex(i/4));
}
GraphBuilder YToH = YSet >= HSet;
GraphBuilder HOutputs = HSet >= output;
GraphBuilder fnl = XInputs || YInputs || XToY || YToH || HOutputs;
```

Figure 2.2: Dryad code for SQL query execution

The clone operation (^) creates cloned vertices, which, when executed, will have the input data split between them. For example, in Figure 2.2, `moduleX ^N` results in the `n` X vertices shown in Figure 2.2. The pointwise composition operation ($=>$) forms a pointwise composition between sets, which creates edges between the vertices. For example, in Figure 2.2, `XSet >= DSet`, results in the edges between the X and D vertices in Figure 2.3. The bipartite composition operation ($>>$) creates the complete bipartite graph between two sets. For example, in Figure 2.2, `DSet >> MSet`, results in the set of edges between the D and M vertices in Figure 2.3. Finally, the merge operation ($||$) operation merges graphs. For example, in Figure 2.2, `(ugriz1 => Xset) || (neighbor => XSet)` will create two graphs containing data dependencies into the XSet, then merge them to produce the dependencies on both sets for each X vertex shown in Figure 2.3. The complete graph for the code in Figure 2.2 is shown in Figure 2.3.

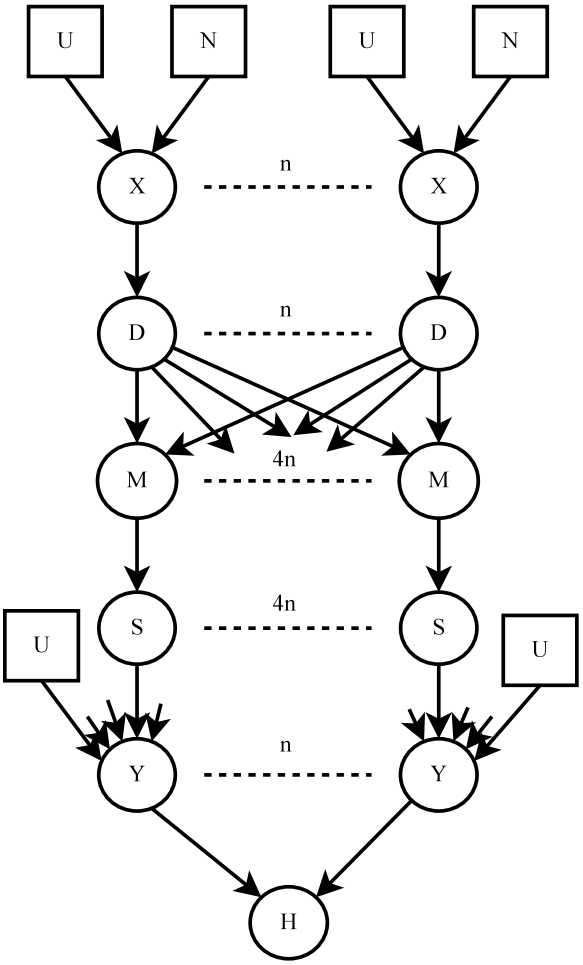

Figure 2.3: SQL Query Graph in Dryad

The job manager supports fault tolerant execution of the vertices by maintaining version numbers and execution logs for each vertex. File output from vertices are uniquely named for each version, to avoid conflicts, and when a job completes successfully, the outputs will be renamed to the final name. If a vertex execution fails, the job manager can be notified by the vertex or daemon. If the failure was due to a failed input channel, the vertex that generated that channel will be restarted. If the machine running the vertex fails, the job manager will receive a heartbeat timeout. Failed vertices are scheduled for re-execution.

Dryad can refine the execution graph to aggregate data from nodes close in the cluster topology. For example, nodes on the same rack may have a dedicated switch for communications within the rack, connected to a central switch for communications between racks. This information is not known until run-time, so it cannot be planned in advance. One use of data aggregation would be in performing a data reduction. Dryad can aggregate the data for each rack, perform the reduction, then send the data to the next jobs in the graph. Performing a reduction before sending data across racks can save bandwidth.

DryadLINQ [29] is a framework on top of Dryad to allow easier programming of data parallel applications. DryadLINQ uses Language Integrated Query (LINQ), an SQL-like language, to specify the data manipulation. LINQ's declarative style is amenable to automatic optimization and scheduling on the Dryad backend. DryadLINQ has its own job manager, which allows dynamic scheduling of the jobs on the cluster. DryadLINQ also has some useful debugging features, which allow you replicate failing jobs and their input and rerun them yourself.

Dryad requires less work to map jobs to its framework than with MapReduce. Commands cleanly map to the vertices and input and output files map to the edges in the Dryad graph. Dryad, like Jole, supports fault tolerant execution of vertices by using a ping heartbeat. If a vertex does not reply within a period of time, it is assumed to have failed, and a new job is scheduled to run. Dryad's scheduler is more advanced than ours, and will try to schedule jobs near the data they need.

Like MapReduce, Dryad does not support dynamic execution of jobs. In cases where jobs do not need to be run to completion, this will result in a longer makespan.

Message Passing Interface

Message Passing Interface (MPI) is a message-passing library specification. Several implementations exist which implement this specification, such as MPICH [9] or Open MPI [19]. MPI provides the programmer with a complete protocol specification for communicating with other processes. MPI is lower-level than the previous systems and will generally require more specification to run similar workloads.

MPI has several features that are useful for writing distributed applications:

1. *Communicator*

   Communicators are objects which are used for communicating with other processes. The

group and context of a process are contained in the communicator. Intracommunicators are for communicating with processes within the same group, while intercommunicators are for communicating with processes in other groups.

2. *Point-to-Point Communication*

   Several functions in MPI provide communication between specific processes, and may be blocking or non-blocking.

3. *Collective Operations*

   Collective operations allow communication between sets of processes and includes operations for synchronization, data movement, and collective computation. For synchronization, the `MPI_Barrier` function is used to synchronize all processes in a group, between phases of a computation. For data movement, their are several functions, including `Broadcast`, which sends the same chunk of data to all processes in the group, and `Scatter` which divides a data chunk among all processes. The computations functions include `Reduce`, which reduces a set of data to a single value, using operations such as addition or multiplication, and `Scan`, which performs partial reductions.

As a programming interface, MPI is a flexible tool for implementing parallel programs. However, implementing an application using techniques similar to Jole, MapReduce, or Dryad in MPI would take a significant amount of programming in order to achieve the same functionality. While MPI is a complete system for communication, the other systems include additional functionality, such as fault tolerance or automatic process placement, that would also need to be implemented to be competitive.

2.1.2 Batch Schedulers

Batch schedulers are systems that schedule and run a large number of jobs on a set of resources. They are useful for running large numbers of jobs, as the user can submit their jobs to the scheduler, letting it run their jobs unattended. Batch schedulers generally take computing resources into account for scheduling jobs. For example, the number of processor cores and amount of memory are attributes that are commonly set for jobs, ensuring they get enough processor time and do not run out of memory. Disk usage can be set under some batch schedulers, which can help prevent thrashing. The resources needed for a job are generally set by users, so any mistakes can lead to problems, such as slowdowns or lost jobs.

Jole is closely related to batch schedulers, as the main goal of running large numbers of jobs on a set of resources is the same. However, there are features not generally available in other batch schedulers that we feel improve the ability to run workflows. Table 2.2 compares how workflows are specified, the type of job execution supported, and the infrastructure usage possible with each

batch scheduler. Table 2.3 compares further features. It should be noted that Jole does not support all the features in Table 2.3, as they were not a focus in this work.

System	Workflow	Job Execution	Infrastructure Provisioning
Jole	Dynamic (Scriptable)	Dynamic	Dynamic
Condor	Static (DAG)	Static	Dynamic
BAD-FS	Static (DAG)	Static	Static
WaFS	Static (DAG)	Static	Static

Table 2.2: Comparison of Key Features

System	Fault Tolerance	Checkpoint/Resume	Dataflow	Capacity Aware Scheduling
Jole	Yes (Job)	No	Yes	No
Condor	Yes	Yes	No	Job
BAD-FS	Yes	No	Yes	Job
WaFS	No	No	Yes	Workflow

Table 2.3: Comparison of Additional Features

Condor

Condor [25] is a batch scheduler that, in addition to the normal functions of a batch scheduler, is able to harness idle computing resources in the execution of a workload. Condor provides libraries, which if used for an application, allow checkpointing and resumption of the application, enabling migration of jobs. In addition Condor has a meta-scheduler called Directed Acyclic Graph Manager (DAGMan). DAGMan allows users to specify their workflow by creating a directed acyclic graph. The graphs are represented by `dag` files, which contain the dependencies between jobs. These files are static, and must be rewritten to create new workflows. An example DAGman script is shown in Figure 2.4.

```
JOB CLUSTALW1 clustalw1.condor
JOB CLUSTALW2 clustalw2.condor
JOB TRAINHMM1 trainhmm1.condor
JOB TRAINHMM2 trainhmm2.condor
PARENT CLUSTALW1 CHILD TRAINHMM1
PARENT CLUSTALW2 CHILD TRAINHMM2
```

Each JOB entry is a job consisting of a name (CLUSTALW) and a script (`clustalw.condor`). Dependencies are specified using the PARENT option, where job names in the CHILD list are dependent on job names in the PARENT list.

Figure 2.4: Example Workflow Specification for DAGMan

Like Jole, Condor has the ability to automatically use idle resources. This ability in Condor is more advanced, as it will detect usage of a machine in deciding whether or not to run jobs on it. Jole relies on a global file to get this information. However, Condor does not have the ability to create a

non-static workload. Furthermore, Condor does not have the ability to interact with jobs while they are running.

BAD-FS

BAD-FS [3] is a scheduler and file system which share information with each other in order to run batch workloads more efficiently. Like in Jole, BAD-FS works on top of existing batch schedulers, rather than replacing them. This makes it easier to utilize their system on resources where root privileges are not available. BAD-FS is able to get improved performance by data-aware scheduling and caching. Intermediate data is kept on cluster nodes, where successive jobs using that data will be run. By running a sequence of jobs on the same machine, the amount of data transfer needed is minimized. BAD-FS is also aware of the data capacity requirements of workloads. By ensuring jobs have adequate storage resources, thrashing is avoided. BAD-FS achieves fault-tolerance by re-executing jobs when they fail or the resources they are using fail. Because BAD-FS avoids copying of intermediate data between jobs, several jobs may need to be run to regenerate the data needed to run the failed job.

BAD-FS uses a workflow script that is similar to those used for DAGman. The following example, from Bent *et al.* [3], illustrates an example workflow script.

```
job     a  a.condor
job     b  b.condor
job     c  c.condor
job     d  d.condor
parent  a  child       b
parent  c  child       d
volume  b1 ftp://home/data 1 GB
volume  p1 scratch 50 MB
volume  p2 scratch 50 MB
mount   b1 a      /mydata
mount   b1 c      /mydata
mount   p1 a      /tmp
mount   p1 b      /tmp
mount   p2 c      /tmp
mount   p2 d      /tmp
extract p1 x      ftp://home/out.1
extract p2 x      ftp://home/out.2
```

Figure 2.5: Example Workflow Specification for BAD-FS

As with DAGman scripts, the job keyword defines jobs, and parent/child keywords are used to define dependencies between jobs. The other keywords are unique to BAD-FS. The volume keyword is used to define data sources required by the workload. The mount keyword binds a volume into a jobs namespace at the path specified. Lastly, the extract keyword indicates which files must be committed to the home server.

BAD-FS does not support dynamic workflow specification, dynamic job execution, or dynamic

infrastructure provisioning. It does, however, have some other features that are useful. It has data-aware scheduling and caching, which can prevent excess data-copying. It also implements fault tolerance similar to Jole. However, BAD-FS may have to re-execute more jobs when a node fails, as it caches intermediate data on nodes. This data has to be created again when the node fails.

WaFS

Workflow-aware File System (WaFS) [26] is a extension of a traditional file system, which provides distinct namespaces for each workflow instance, and gathers dataflow information about the workflow. The WaFS scheduler uses this information to increase inter and intra-instance concurrency of the workflow.

WaFS has three policies that were tested for handling file conflicts:

1. *Versioned Namespace (VNS)*

   WaFS implements VNS by naming files with version numbers. For example, the file `A.out` could be called `A.out.1` in workflow instance one, and `A.out.2` in workflow instance two.

   This policy is most similar to the one used with Jole. Jole adds version numbers to files only when it finds conflicts, rather than for each workflow instance.

2. *Overwrite-Safe Concurrency (OSC)*

   OSC is a policy which only overwrites files when it is safe to do so. Rather than versioning files so that each workflow instance is isolated, OSC only allows jobs from an instance to replace files when the earlier instance no longer needs the file.

3. *Hybrid*

   The hybrid policy uses a storage budget to decide how much versioning can be used to improve concurrency without incurring deadlock. If files are not needed by earlier instances, they will be overwritten by subsequent instances.

WaFS has two policies for scheduling jobs:

1. *Dataflow-based Aggregate Requests (DAR)*

   DAR attempts to maximize active storage utilization and increase inter-instance concurrency. The dataflow information is used in order to minimize the maximum claim of each instance. Resources are allocated at the workflow instance level.

2. *Dataflow-based Topological Ordering (DTO)*

   DTO attempts to maximize active storage utilization and intra-instance concurrency. Resources are allocated at the job, rather than instance level. Before running each job, DTO analyzes the storage requirements and determines whether after executing the job, there are enough resources to execute the rest of the workflow in topological order.

WaFS has more focus on dataflow gathering and scheduling of workflow instances using the dataflow information. The goals of Jole are more towards making it easier to specify workflows and dynamically control their execution. Jole focuses on user specification of dataflow information, rather than gathering it automatically. These goals are complimentary, and combining the elements of each system could produce a better system for running workloads. Jole does not schedule based on the storage resources of the workload and keeps all intermediate files from a workload. For workloads that output a significant amount of intermediate data, the hybrid policy for file conflicts would be useful. Additionally, the scheduling policies from WaFS would be useful for deadlock avoidance when running these workloads.

The use of WaFS for dataflow gathering would likely only be useful for static workflows in Jole. Using dynamic job execution or dynamic workloads in Jole would make it difficult to gather useful dataflow information, due to variable length jobs and differences in the workflow graph between instances.

TrellisDAG

TrellisDAG [7] [8] is a workload execution system developed by the Trellis Group [17], and is similar in many ways to our system. It has three main parts:

1. *Description Layer*

   At a highest level the description layer consists of groups of jobs. Jobs within a group are assumed to have a pipeline dependency graph, and will execute in serial order. If a group contains subgroups and jobs, the subgroups will execute before the jobs. Dependencies are not specified between jobs, but between groups.

   Workflows can be submitted using a flat file, makefile, or a DAG description script. For complex workflows, using a DAG description script is recommended, as it will greatly simplify the submission script. DAG description scripts rely on a Python module, which contains the interface that generates the workflow. Users specify their workflow using this interface, and the result is converted and submitted to the jobs database.

2. *Job Database*

   The job database is contained in a PostgreSQL relational database. The job database contains information about group membership and super/subgroup relationships. The database also contains information about the dependencies between groups. The job database updates job and group execution information as jobs complete.

3. *Placeholders*

   A placeholder represents a unit of potential work. In TrellisDAG, it is implemented as a shell script. Placeholders connect to a command-line server, get the next available job, execute it,

and resubmit themselves. Placeholders are one way of building an overlay metacomputer. By running placeholders across administrative domains, it is possible to have a single point of control for execution. Placeholders allow easier load balancing between resources because they claim jobs only once they are able to run. Thus, jobs will be dynamically load balanced as they only are tied to a resource once it is ready to run them.

At a high level, Jole is similar to TrellisDAG. Both Jole and TrellisDAG use placeholders, which allows them to take advantage of resources in multiple administrative domains. As a side effect of using placeholders, they both handle load balancing well, as jobs are not tied to any particular resource in advance. They will run on whatever resource asks for a job first.

Jole and TrellisDAG both contain a job manager, which contains information about jobs, their dependencies, and their status. TrellisDAG's job manager is a PostgreSQL database with command-line tools as an interface. Placeholders communicate with the database using using Secure Shell (SSH) [28]. Our job manager is a Python class, which communicates with the placeholders over TCP/IP sockets.

Workflows are created quite differently in Jole and TrellisDAG. In TrellisDAG, the user specifies a workflow by creating a Python module corresponding to the interface required. This means that the workflow is static after it is created. In Jole, the user's workflow script can respond dynamically, changing the final workflow that is executed. For example, the workflow script may use global data about job results to create new jobs and to decide whether to continue running jobs. Furthermore, we support data dependencies between jobs using data handles. When jobs are created using data handles, dependencies do not have to be explicitly specified, as they will be created automatically based on the files a job is reading and writing.

In addition to this, Jole is capable of interacting with jobs as they are running. For some types of jobs, we can determine whether the results of a job are improving at a quick enough pace, and if they are not, kill the job. New jobs could be started in their place, potentially getting better results, or no new jobs could be submitted, freeing space on the cluster for other jobs.

2.2 Concepts

2.2.1 Futures

Futures are used in the programming language Alice ML [14], and were used in earlier languages such as MultiLisp [12] and Act 1 [16]. Concurrent futures, which are used in Jole, are a way for programs to increase concurrency and synchronize on values that may take some time to compute. The future acts a proxy for the value of a computation, which is computed in parallel with the program. Concurrent futures start the computation as soon as they are created. Once the program needs to use the result of the future, it will try to access it. If the future is not done, the program will wait on the future until it finishes and returns the result. This can be used as a method of

synchronization, as the program can continue running until it actually needs the value of the future.

Alternatively, lazy futures, which can be used to prevent unneeded computation, only start a computation once the program tries to access the future. This has benefits if the result returned by the future may not be used. With eager futures, the computation would execute regardless of whether or not the result will be used. The downside of lazy futures is that the total runtime will be longer if most of the results from the futures will be needed.

2.2.2 Placeholders

Placeholders are a mechanism for creating an overlay metacomputer [22]. An overlay metacomputer allows a user to have a unified interface to a set of resources. A placeholder is defined as a unit of work. They will run tasks for the metacomputer. Placeholders may be submitted to the batch scheduler in order to complete tasks, or may be started on machines that do not have batch schedulers (zero-infrastructure).

Placeholders are not bound to a job until they are executed on a node. This late binding is useful for scheduling across multiple resource sites. Rather than submitting jobs to the batch schedulers of each site, placeholders can be submitted instead. Once the placeholder starts execution, it will request a job from the job server. This allows the load to be balanced among sites, as each site will only pull jobs from the job server when it can run them.

2.3 Motivating Applications

2.3.1 Pathway Analyst

Pathway Analyst [23] is a metabolic pathway predictor developed at the University of Alberta. A pathway is made up of a series of reactions, by which a compound may be converted into something more useful to the organism. Pathways can be represented by graphs, where the reactions are the edges and compounds are the vertices. Given a user-supplied proteome, Pathway Analyst creates a new predicted pathway, based on the proteins that are predicted to participate in the reactions specific to the pathway. A proteome is the complete set of proteins encoded by the DNA of an organism. Each protein is represented by a series of letters representing the amino acids that make up the protein.

One of the classifiers used in Pathway Analyst is the hidden Markov model (HMM) classifier. HMMs are probabilistic models that can be used to model the hidden states of a sequence. With catalysts, we are trying to model the shared structure of the catalyst proteins that is most likely the region that allows the protein to catalyze a certain reaction. To do this, we first use `clustalw` to find the most probable alignment of the proteins based on the likelihood of different mutations. With this alignment, we use `hmmbuild`, a tool in the HMMer [24] toolkit, which analyzes the alignment and produces an HMM. With the HMM, we can use `hmmsearch` to search through protein files

and find other proteins that are more likely to catalyze the reaction.

2.3.2 GAFolder

GAFolder is a protein structure energy minimizer developed at the University of Alberta. GAFolder uses cyclic coordinate decent and a genetic algorithm to perform conformational sampling. The genetic algorithm uses an energy score to evaluate structures, which is based on predicted/known secondary structure, radius of gyration, hydrogen-bond energies, and other features. During execution, GAFolder maintains a set of structures as the population within a GAFolder instance. In each iteration, the structures are mutated, changing the torsion angles of the protein. The best structures survive and will ”mate” with other structures, combining different mutations. A single instance of the program may converge to local minima, so several instances are executed in order to get the best structure prediction.

GAFolder is used within CS23D [27] and GeNMR [4] to help refine the protein structure.

2.4 Concluding Remarks

There are many methods for specifying and running batch scheduled jobs. We have shown how a variety of languages and job scheduling systems specify and execute jobs, as well as the benefits of each system. For Jole, we focus on the dynamic execution of jobs and workloads. The use of these methods can improve runtime or quality of results for several types of workloads. We have also introduced some concepts used in Jole, as well as some motivating applications.

Chapter 3

Jole: Job-Level Parallel Workloads

As capacity computing resources increase in size and performance, it has become more important than ever to be able to efficiently take advantage of these resources. Batch schedulers are good at running large numbers of independent jobs, but it may be difficult to specify dependencies or they may have to be controlled externally. The goal of Jole is to make it easier and more flexible to run these large batch workflows.

Current methods for running batch workloads can be cumbersome. For example, in some batch schedulers, even specifying job dependencies may require extra work on the part of the user. With the TORQUE Resource Manager [18], the user must get the job identification numbers of all jobs they need as dependencies, and add those dependencies to job submissions for each job they need to run. With Jole, users can specify the files that will be used by a job, allowing our job manager control job dependencies automatically.

In batch schedulers that support it, such as DAGman, dataflow information must be specified statically, by writing scheduler-specific job files that contain the dependencies for each job, for each set of input and output files. This static specification of workflow causes the need for a large amount of information that must be created for each workflow instance. With a scripting language interface, workflows can be created dynamically, based on file inputs or some other criteria. In addition, with Jole, it is not necessary for the workflow to be an acyclic graph. Users could specify that a group of commands repeat until their output meets some criteria. This can be useful for programs which refine their output on each iteration. For example, with GAFolder, several instances of the program are run in order to make protein structure predictions. The best of the predicted structures will be used as input for the next generation. This process can be repeated to further refine the protein structure, until the energy improvement does not meet some threshold.

Current methods also assume that jobs are atomic units of work that must be finished to completion. This is not true for all workloads. For example, optimization jobs may hit local minimas or maximas. Rather than letting the optimization complete, it might be beneficial to end it early once this convergence has been detected. Output of jobs may even be used to remove jobs from the queue. For example, when running a set of BLAST commands with varying E-values, one of

the commands may return no proteins. Once this is discovered, it is not necessary to run any of the BLAST commands that have a more stringent E-value, as they would not return any results.

Lastly, batch schedulers are not able to dynamically adjust workloads based on the available resources. Users may want some method of adjusting their workload based on the available resources. With the use of placeholders and submitters, we create an overlay metacomputer, aggregating multiple resources. The submitters are executed on the head nodes of each cluster the user wishes to run their jobs on. They are responsible for submitting placeholders to the batch schedulers and forwarding traffic between the placeholders and job manager. The workflow script can get resource information from the job manager in order to determine how to continue the workflow.

Jole was created in order to increase the flexibility of running workloads on clusters. In this chapter, we describe the concepts of Jole and how each concept results in an improvement over current methods.

Feature	Targeted Problem
Job Future	High latency batch jobs and synchronization
Job Array	Iteration and synchronization over multiple job futures
Data Handle	Data flow synchronization and file conflicts
Command Generator	Specification of large number of related commands
Monitor	Interaction with job while running
Submitter	Interacting with multiple batch schedulers

Table 3.1: Jole: Library Features

3.1 Pathway Analyst Example

Throughout this chapter, we include code examples to demonstrate each concept, by implementing a script to train HMM classifiers for Pathway Analyst, a metabolic pathway prediction tool created at the University of Alberta. Each iteration of the script will build upon the previous, using the concepts described. Each model is trained, as follows:

1. Align a set of proteins which catalyze a single reaction, using `clustalw` [15].
2. Train an HMM on the aligned proteins using `hmmbuild`.

The workflow for Pathway Analyst is shown in Figure 3.1. We implement the alignment of proteins using `clustalw` and training of the HMM using `hmmbuild` in the following examples.

3.2 Job Future

One of the main concepts in Jole is the idea of a job future. Job futures contain the necessary information to run a job, and submit this information to the job manager. Like futures in other languages, our job futures will execute code in another thread, immediately returning control to the

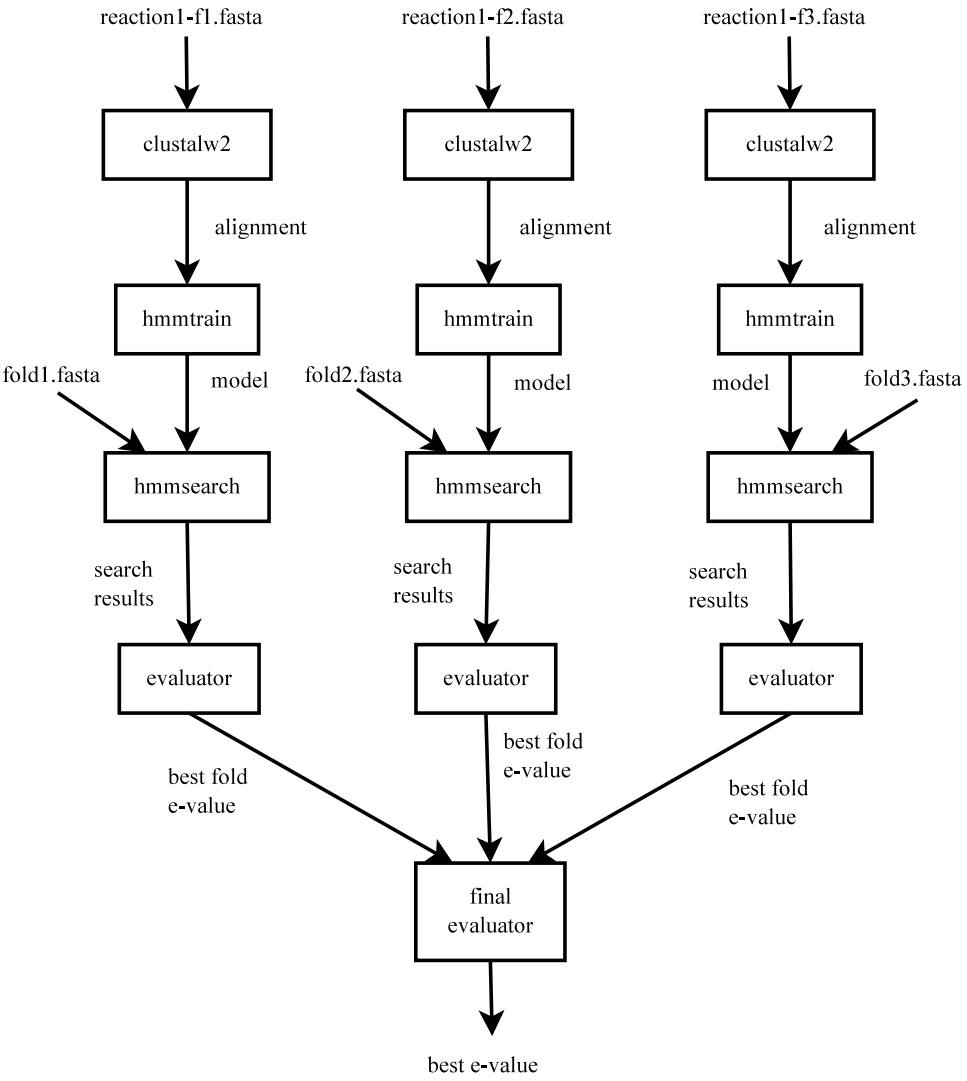

Figure 3.1: Pathway Analyst Workflow: Training an HMM Classifier

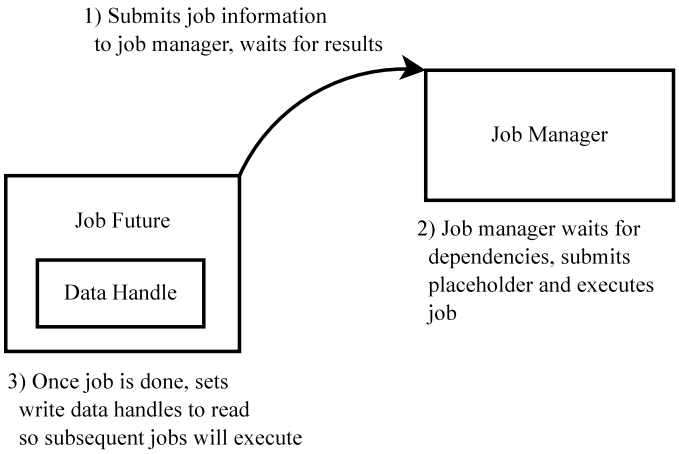

Figure 3.2: Execution of a Job Future

main program. If the main program tries to access a result from the job future, it will wait on the job until the result is available.

Job futures are a simple way for Jole to deal with dynamic job completion order. Because the job futures are executed asynchronously, they will respond to job completion with low latency, allowing dependent jobs to start as soon as the current job is finished. Job futures can handle heterogeneous cluster nodes or changes in workflow execution without blocking, due to their asynchronous nature. Job futures are illustrated in Figure 3.2

In the example coming up, we create two jobs futures, `jf` and `jf2`, then wait on `jf2`. Due to the asynchronous nature of job futures, the creation of both immediately return control to the script, which then waits on the completion of `jf2`.

The main difference between futures and job futures is the ability to run arbitrary code. Futures are generally used to execute long running methods in a program, while job futures execute programs written in any language. This difference in granularity is important as well. Job futures are tailored to running batch jobs, rather than segments of code.

Dependencies can be specified with dataflow or control flow information. Control flow information is passed using the `depends_on` argument, which is a list containing the job futures that the current job future must wait for before executing. Because job futures start executing upon creation, they must be created in order of dependencies. Dataflow information is embedded in the command argument, using data handles. These data handles must also be specified in the correct order of dependencies when creating job futures. For example, suppose a file is going to be written to by one job, then read from by a subsequent job. If the data handle for the subsequent job is created first, the job will assume that the file should already exist, since no existing data handles show the file should be written to. It will not be dependent on the job that writes that file, and will try to read it without waiting for the other job. When created in the correct order, the job that reads that file will see that another job is going to write to it, and it will wait for that job to finish before executing.

When created, job futures submit their job information to the job manager. The job manager will look at explicit dependencies and the data handles to check if the job has unmet job dependencies. If it does, the job manager will wait until they are met. Once the dependencies are met, the job manager submits a placeholder to the batch scheduler. The job future waits for notification from the job manager that its job is done.

The futures may be synchronized with the main script by waiting for them, trying to access their results, or by waiting on a collection of them. A collection of job futures allows the script to use synchronization methods, such as `wait_all`, `wait_any`, or `wait_some`.

3.2.1 Example

The first step for creating an HMM classifier for Pathway Analyst is to align the proteins and create and HMM model based on their alignment. We use `clustalw` to align the proteins and

`hmmbuild`, a tool provided with HMMer [24], to produce the HMM model. Sample code is shown in Figure 3.3.

```
# Initialization of job manager
from job_manager import JobManager
jm = JobManager()
jm.new_submitter('PBS','botha-c10')

# Create clustalw job
jf = jm.new_job_future(
       "clustalw -INFILE=32864.fasta -OUTFILE=32864.aln")
# Create hmmbuild job
jf2 = jm.new_job_future(
        "hmmbuild -F 32864.hmm 32864.aln", depends_on=[jf])
# Wait for hmmbuild job to finish
jf2.wait()
```

Figure 3.3: Creating job futures

Each job future is initialized using the `new_job_future` method of the job manager. The `clustalw` job future is initialized with only a command string. The `hmmbuild` job future is given a list argument `depends_on`, which lists other job futures that must complete before running this one.

3.3 Command Generator

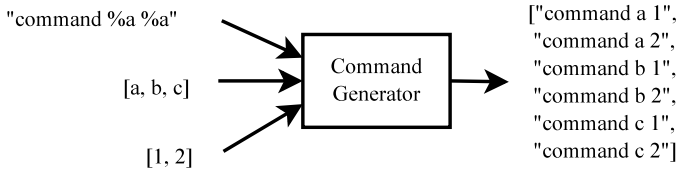

Figure 3.4: Command Generator Usage

The command generator is a useful abstraction for automatically creating sets of commands. Parameter sweeps can be easily specified, allowing the jobs to be automatically created and run, without the user having to iterate through all the parameters. Parameter sweeps are created using the Cartesian product of the parameters, but using filters on the command generators will make it easier to specify custom parameter sweeps.

Command generators help with the dynamic specification of workflows. Rather than specifying a static workflow, command generators allow the user to specify workflows dynamically based on lists of input/output files and parameters. The usage of a command generator is illustrated in Figure 3.4.

Commands are generated by specifying a skeleton string, containing printf-like variables, along with a list for each variable. The Cartesian product of the lists is taken, and each element is used

to create a command, substituting the variables in the skeleton string. To filter a command, the user specifies a function which takes the same amount of arguments. Based on these arguments, the function must be written to return true if they want to filter the command based on its variables. The commands are added to a job array to create job futures.

In the following example, a set of commands is created using two lists to build the commands.

3.3.1 Example

Training classifiers in Pathway Analyst involves running a similar set of commands over many files. Rather than iterating over all the possible arguments to produce our command strings, we can use a command generator to produce the set of commands for us. Use of a command generator is shown in Figure 3.5.

```
fasta_files = ['1.fasta', '2.fasta']
aln_files = ['1.aln', '2.aln']

# Generate clustalw commands with given argument lists
cmds = generate_commands("clustalw -INFILE=%a -OUTFILE=%a",
                         [tie_arguments(fasta_files,aln_files)])

# Resulting cmds array
# cmds = ["clustalw -INFILE=1.fasta -OUTFILE=1.aln",
#         "clustalw -INFILE=2.fasta -OUTFILE=2.aln"]
```

Figure 3.5: Generating commands

In this example, each entry in `aln_files` is the output name for the `clustalw` command, corresponding to the entry in `fasta_files` at the same index. Because we want their entries to correspond to each other, we use the `tie_arguments` function. The `%a` flags in the command string are substituted with entries from our parameter lists. Filters are not used in this example.

Filters are simple functions that return `True` if the set of parameters should be filtered from the generated commands. Example use of a filter is shown in Figure 3.6.

For this example, a filter is created which takes two arguments. Filters must have the same number of arguments as are being specified in the command generator. The filter here will return `True` if the first argument is greater than 10 and the second argument is greater than 50. Those sets that meet this criteria are filtered from the final command list.

3.4 Job Array

A collection of job futures can be contained in a job array, which allows for some useful operations to be performed. Firstly, iterators of the job array allow us to iterate over the job states of the job futures. Currently, iterators only exist for the completion states, but other iterators will be useful. For example, iterators over error states would be useful in implementing fault-tolerance. Secondly,

```
def filter(a1,a2):
  if a1 > 10 and a2 > 50:
    return True
  return False

list1 = [5,20]
list2 = [25,75]

cmds = generate_commands("command %a %a", [list1, list2],
                         filter_fun=filter)

# Resulting cmds array
# cmds = ["command 5 25", "command 5 75", "command 20 25"]
```

Figure 3.6: Filtering commands

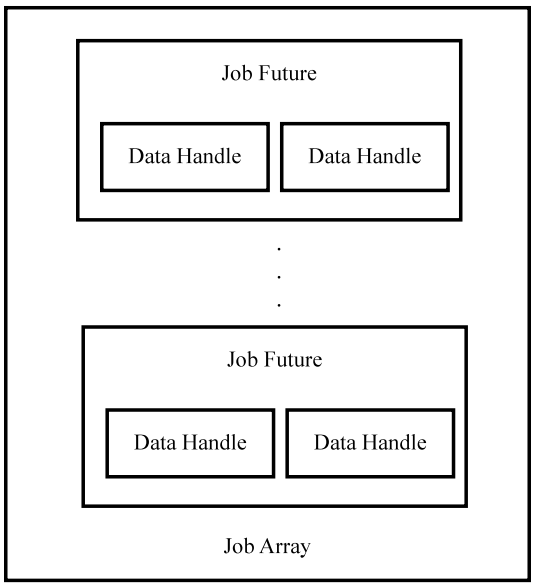

Figure 3.7: Job Array

the job array can be used for job synchronization. The standard `wait_all`, `wait_some`, and `wait_any` synchronization methods are supported.

Job arrays may also be useful for dynamic workflow creation. For example, a common workload is a parameter sweep. A static workflow could be created to run jobs with a series of parameters. With job arrays, we may iterate over the results of jobs as they finish, and create new jobs in promising areas of the sweep. Job arrays are illustrated in Figure 3.7.

Job arrays can also be used to specify monitors for groups of jobs. By adding a monitor to a job array, the monitor will be used on every job in the array.

Job arrays indirectly interact with the job manager through the job futures. Iteration and synchronization are based on interacting with the job futures in the array, which then interact with the job manager.

3.4.1 Example

In the previous example, we generated a set of commands with the command generator. It would be possible to create job futures out of the commands individually, but a job array allows us to instantiate all the job futures at once, and gives us some extra synchronization. An example using job arrays is shown in Figure 3.8

```
# Variable cmds contains commands from command generator
# cmds = ["clustalw -INFILE=1.fasta -OUTFILE=1.aln",
#         "clustalw -INFILE=2.fasta -OUTFILE=2.aln"]
# aln_files = ['1.aln', '2.aln']
hmm_files = ['1.hmm', '2.hmm']

# Create job array
aln_ja = JobArray()
# Add commands to array
aln_ja.add_commands(cmds)
# Wait for all job futures in array to finish
aln_ja.wait_all()

# Create job array for hmmbuild commands
hmm_ja = JobArray()
# Generate hmmbuild commands
cmds = generate_commands("hmmbuild -F %a %a",
                         [tie_arguments(aln_files,hmm_files)])
# cmds = ["hmmbuild -F 1.aln 1.hmm", "hmmbuild -F 2.aln 2.hmm"]

hmm_ja.add_commands(cmds)
hmm_ja.wait_all()
```

Figure 3.8: Job Array Example

The method `add_commands` is used to add a list of commands to the job array. In this example, we use the `wait_all` synchronization method to ensure all protein alignment jobs are finished

before starting the `hmmbuild` commands.

3.5 Data Handle

Data handles are a useful abstraction on files. They allow dataflow information to be handled by Jole. By specifying commands with the command generator, data handles are simplified, but provide the benefit of dataflow job synchronization. If Jole were to be combined with a dataflow scheduler, this information can be obtained automatically.

Data handles must be created in dependency order, similar to job futures. The first reference to a file by a data handle determines its initial state. For example, if a read data handle is created first, Jole will check for the file, and if it does not exist, return an exception. If a write data handle is created first, Jole assumes that the file will be a new file, and will check the filename for conflicts, automatically changing the name if there are conflicts. Subsequent jobs that have read data handles on those files will wait for the write data handles to be finished before executing.

Data handles help with the dynamic specification of workflows. Data handles allow the user to easily iterate over directory contents, creating workflows based on the input files. The job futures use these data handles to automatically create dependencies between jobs.

Data handles are also useful for fault tolerance. One feature they can handle are filename conflicts between workflows. When there is a conflict between filenames, data handles will automatically choose a new filename and use that filename in any commands which use the data handle. This filename remapping is stored in a global table. Subsequent data handles look up filenames in the global table. If a filename has been remapped, the new data handles will have that information. So, the user can refer to files by their original name in the workflow while Jole uses the renamed files behind the scenes. Data handles are also useful for file access problems. If a file is only accessible on the head node of a cluster, a data handle could automatically retrieve the file, so that it can be used by the running job.

Data handles may also be used to improve I/O performance. By copying heavily used files to local storage, job run times may be improved. It may also be possible to keep intermediate files in local storage by running subsequent jobs needing the intermediate file on the same node.

Data handles are used by the job manager in order to synchronize jobs based on dataflow information. The job manager will inspect the contents of a job for data handles, waiting for them to become available before starting the job. Once the job is finished executing, the job future will change the state of write data handles to read, allowing jobs that need to read these files to start executing.

In the following example, data handles are created automatically by specifying the `%r` and `%w` flags to `generate_commands`.

3.5.1 Example

Using data handles can let us avoid having to specify job dependencies manually through job futures or job arrays. A data handle example is shown in Figure 3.9.

```
cmds = generate_commands("clustalw -INFILE=%r -OUTFILE=%w",
                         [tie_arguments(fasta_files,aln_files)])

ja = JobArray()
ja.add_commands(cmds)
cmds = generate_commands("hmmbuild -F %r %w",
                         [tie_arguments(aln_files,hmm_files)])
ja.add_commands(cmds)
ja.wait_all()

# Basic job futures are created using the following construction:
# jf = JobFuture("filename", "w") # A file that will be written to
# jf = JobFuture("filename", "r") # A file that will be read from
```

Figure 3.9: Specifying data handles in command generator

In this example, the %a flags are replaced with %r and %w flags, which represent files that will be read or written to by the command. These flags are used so that data handles will be created for those arguments. The job futures created with these commands will automatically synchronize on the data handles, based on their read/write properties.

3.5.2 Dataflow

Data handles are used to represent the files within a workflow, and to handle filename conflicts between workflows. The state of each data handle is stored in the workflow, so that jobs which need to use those data handles can wait for them to be available by checking the workflow. By using data handles for all file input and output in a workflow, the workflow can execute efficiently based on the dataflow information.

3.5.3 Remote File Access

One problem when writing a script to run on a cluster is that files accessible on the head node may not be accessible from the compute nodes. File handles can help with this problem as well. If a file handle tries to access a non-existent path or file on a compute node, it can copy the file from the head node and modify the path in the command so that the job may execute. A similar check for writing files is done as well. If the file handle determines write access to the destination directory is not possible, it may change the path so that the job can be executed, then copy the file back to the head node in the proper path.

3.6 Job Manager

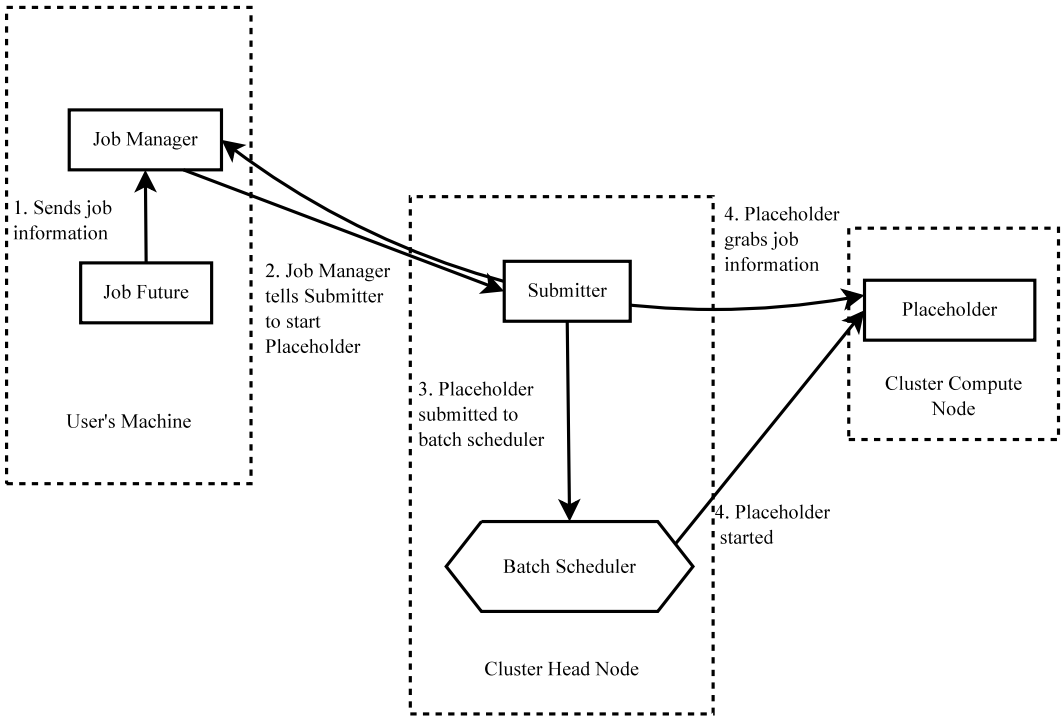

Figure 3.10: Overview of Job Execution

The job manager stores job information and status, sending job information to placeholders when they request it.

The job manager sends messages to submitters to submit placeholders to the underlying batch schedulers. A placeholder represents a unit of work, but the placeholder is not bound to a specific unit of work until it is executed. Once it is executed, the placeholder communicates with the job manager to grab a job. If there is a job available that has its dependencies met, the job will be sent to the placeholder. If there are no available jobs, the placeholder may be told to wait or terminate. By executing jobs in this way, the job manager controls job synchronization, rather than the batch scheduler.

The job manager also controls access to the placeholder for remote monitoring. Each placeholder has a communications port which the job manager stores. When a monitor wants to communicate with a job, it asks the job manager for a placeholder communicator. Once the job is grabbed by a placeholder, the job manager gives the communicator to the monitor.

Several problems can occur while running a job. Specific nodes may be missing libraries, have network issues, or have limited storage space. Error recovery for these problems can be as simple as running the job on a different node. Placeholders periodically send messages to the job manager to notify it that they are still running. If the job manager stops receiving these messages, it assumes

the machine has failed, and reschedules the job.

3.7 Submitter

The submitter is an abstraction on top of the batch scheduler. This abstraction allows Jole to have a constant interface to different batch schedulers, letting programs written in Jole work on all supported batch schedulers. Workflows specified with batch scripts will be tailored to a single batch scheduler. Workflows specified in Jole can run on any scheduler that a submitter has been created for.

Submitters are specified by the user. They must specify the hostname of the machine and the type of batch scheduler that is available. A submitter daemon will be started on each node specified by the user. Generally, they will be run on the head node of a cluster, from where they can submit placeholders to run jobs. This allows users to take advantage of multiple computing resources transparently.

Submitters also contain a socket forwarder, which acts as a proxy for communication when direct network connections are not possible. For example, on some clusters the compute nodes may only have network access to the head node, for security reasons. This means the placeholders cannot connect directly to the job manager. In cases where this happens, placeholders will instead connect to the socket forwarder which will forward their communication to the job manager.

3.8 Monitor

Monitors allow interaction with jobs as they are running. Users may want to end jobs early based on their poor results, or start new jobs based off of jobs with good results. Monitors allow users to perform these interactions while the jobs are being executed. Monitors are the key mechanism that enables dynamic job execution.

3.8.1 Remote Monitor

Remote monitors are run on the host machine and interact with a stub server that runs with the placeholder. The server has some basic functions that the monitor can call, similar to the idea of a GNU Debugger (GDB) stub program for debugging on embedded devices. These commands may ask for output from some file, ask whether a file exists, or end a job. This prevents some problems with trying to run code on the cluster nodes. Since the main program will be run on the head node, all libraries needed by the user code will already be loaded. If the user wants to use an executable to perform some task based on communication with the stub, they only have to ensure the host machine supports their code.

One common use for a monitor is to watch the output of a file. The raw output of a file may take up too much bandwidth to transfer efficiently, so we support the usage of regular expressions to

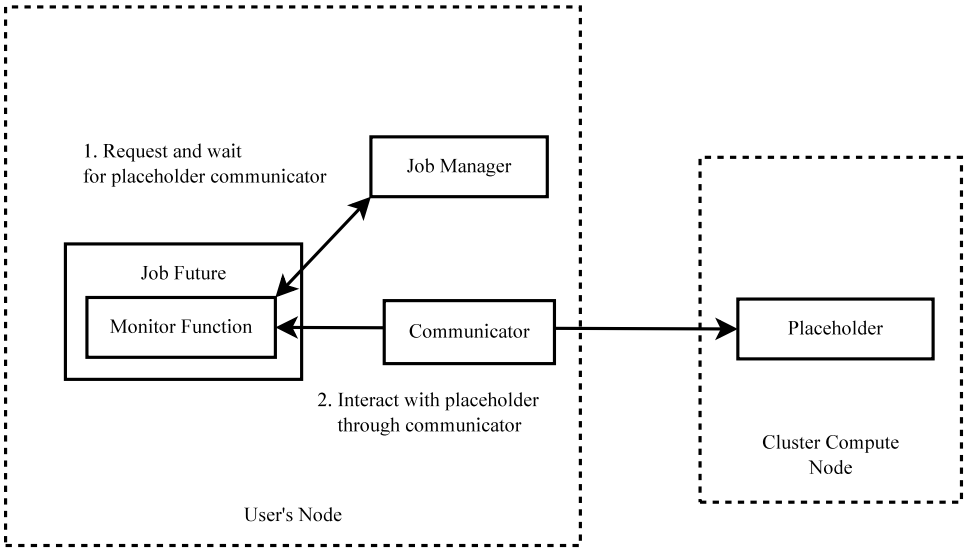

Figure 3.11: Remote Monitoring of a Job

limit the output that is sent to the monitor.

Monitoring over the network could be extended by allowing arbitrary commands to be executed by the placeholder stub server. The user may want to start a bash session on the placeholder server and submit a sequence of commands to be followed. This would allow a greater range of functionality to be performed by the stub server, without having to encode it within the server.

3.8.2 Local Monitor

We have created monitor classes to perform common functions, such as regular expression search or file existence checking. These monitor classes are passed on to the placeholder, along with the job, and will run in parallel with the job. Each monitor changes the state of the job when a positive result is returned. Monitors can depend on other monitors, so that a series of checks can be done in order. A monitor can change the state to the end state, which will kill the job. This can be useful for terminating jobs under certain circumstances.

The ability to send user specified functions along with monitor classes is being worked on. This function would be passed to the job future along with the monitor class and would be added to the placeholder when it is created. The placeholder will execute the monitor class with the user specified function while the job is running. It is possible to determine all libraries in use in the current script, so prerequisite libraries could be automatically imported in the user function. Otherwise, the function might not have the required libraries available when it is run in the placeholder. It could also be possible to specify external processes to monitor the output file. The placeholder would run this process and watch its output to determine whether to kill a job. This monitoring method may be useful to users, but may be less portable if the software across machines is not relatively

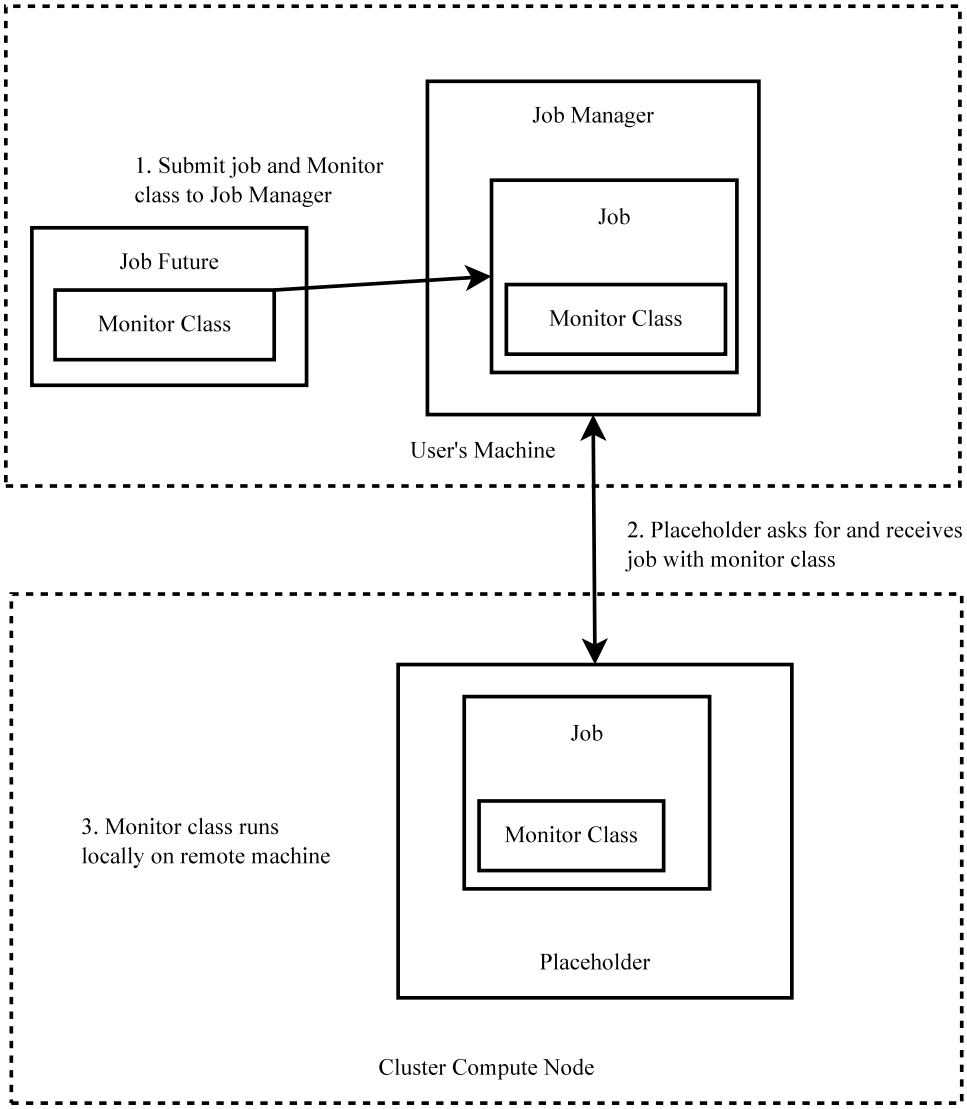

Figure 3.12: Local Monitoring of a Job

homogeneous.

Placeholder local monitoring benefits from much more efficient I/O access to output files than network monitoring. Depending on the job, running placeholder local monitors could result in a large decrease in network I/O.

Placeholder local monitoring may also be useful if users want to run a minimal version of Jole to just get the benefits of monitors. The job manager could be avoided entirely by submitting the job information and monitor with the placeholder to the batch scheduler, and letting the placeholder run the job. This allows the user to monitor certain aspects of their job without the dependencies needed by the job manager.

Local Monitor Types

We have created some basic monitors, as follows:

1. *Regular Expression Monitor*

   A regular expression monitor watches a file until the specified regular expression is matched. Basic functions can be added to the regular expression monitor to check values returned from groups in the regular expression.

2. *File Creation Monitor*

   A file creation monitor will watch for a specified filename. Once the filename has been created, the monitor is finished running.

3. *Custom Monitor*

   A custom monitor allows the user to create a custom function to execute within the monitor on the remote machine. By scanning the environment for modules and functions that are currently loaded, the user's function will be converted back into Python code along with the code necessary to load the modules and functions currently in use. Once started on the remote machine, the monitor will execute the code to create the function, then execute the function.

4. *Executable Monitor*

   An executable monitor allows a user specified executable to be run on the remote node. The executable must return zero for the monitor to return successfully.

3.9 Adapting Jole for Different Applications

We have used a workflow from Pathway Analyst in this chapter to demonstrate the features of Jole. Now, we explain the high level details of creating workflows for other applications. There are a few steps required in creating a workflow in Jole. These steps may vary slightly based on what features users take advantage of. At a minimum, users will probably want to use job futures with

dependencies specified between them, using control-flow or dataflow dependencies. The control-flow dependencies may take the form of dependencies given to each job future, or synchronizing on sets of jobs using job arrays. If they are creating many related commands, such as in a parameter sweep, command generators will allow them to specify these commands with less syntax. The Pathway Analyst workflow can take advantage of command generators to specify the commands, while the GAFolder workloads in Chapter 5 do not need command generators, as they consist of the same command being run multiple times. Additionally, these workflows can be parametrized, so that the workflow script may be run on different sets of inputs. For example, the GAFolder workloads in Chapter 5 could be parametrized to take a protein structure as input to the workflow. This reduces specification needed to run different workflow instances. In general, different workflows will not have a lot of similarity between each other and will have to be mostly written from scratch.

More advanced workflows may use custom monitors in order to enable dynamic job execution. Custom monitors will likely have to be written for different applications, as they may have different properties that must be taken into account for the monitor. Although the monitors will generally need to be tailored to each application, the mechanism behind monitor execution is general and allows Jole to run monitors specific to each application. If a user has other workloads that use the same application, they can use their custom monitors in the other workload. For example, the workflow for Pathway Analyst has no need for monitors, as all jobs need to be run until completion. However, as shown in Section 5.2, monitors can be used to terminate GAFolder job instances early and save cluster resources.

The concepts of dynamic workflow and dynamic infrastructure are general and can be used in different workflows. However, how they are specified is workflow dependent, and will be different for each workflow. For example, different workflows performing iteration over a global value will likely be looking at different job information or output to determine whether to continue iteration. Using infrastructure as it becomes available is a general concept of Jole, and this will be done regardless of what is specified in the workflow. The only necessary prerequisite for this is to run submitters on the clusters that the user wishes to use. However, modifying the workload based on the available resources is workload dependent, as different jobs may not benefit from running additional instances. This type of information must be specified in the workload. For example, these features are not needed in the Pathway Analyst workload, as the jobs must all run to completion and do not have properties that would benefit from iteration. However, the Pathway Analyst workload would be able to run jobs on additional idle resources by adding submitters. GAFolder can benefit from these features, allowing an iterative workflow on global job results to be done or for more jobs to be run when there are additional resources, as shown in Sections 5.3 and 5.4. Similar to the Pathway Analyst workload, the GAFolder workloads can run on idle resources, as long as submitters are created for each resource in the workflow script.

3.10 Concluding Remarks

In this chapter, we have described the features of Jole and why each feature is important. The Pathway Analyst example allows us to present the common usage of the various features. These features enable dynamic job execution, dynamic workloads, and dynamic infrastructure.

Chapter 4

Implementation

We have implemented Jole in Python. As a scripting language, Python requires has less verbose syntax than some other common languages, such as C++ or Java. This less verbose syntax can make it simpler to specify workflows, because less code is needed and there are fewer language concepts users need to know before implementing their workflow. Although Python is slower that C++ or Java for many CPU intensive tasks, using C++ or Java to implement Jole would likely not result in a large difference in performance. Since jobs are generally long running, the workflow script will be waiting for jobs to finish most of the time, rather than performing active computation. Improvements in the small segments of time where the script is working would likely not make a large difference in overall performance.

Other scripting languages, such as Ruby or Perl would be decent candidates for implementing Jole. Like Python, they have less verbose syntax than others, such as C++ or Java. However, we feel that Python has a good mix of clean syntax, a simple application programming interface (API), and performance when compared to other scripting languages that makes Python a good choice for Jole.

Most of Jole's functionality is contained in a library, which is used to create and execute workloads. The additional functionality is in two Python scripts. The first script is the submitter script, which is run on the head node of each batch scheduler being used. The second script is the placeholder script, which is executed by the batch scheduler and runs the job.

4.1 Job Future

Job futures are a class in our library, which uses a thread to handle asynchronous computation. When a job future is initialized, the thread handles execution of the job. Control is handed back immediately to the script after creating the thread. In the thread, the job future notes which data handles are being written to and submits the job to the job manager. The job future will then wait on the job manager until the job is done. Once it is done, all write data handles will be set to read, so that jobs dependent on reading these data handles can start.

While the job future is executing, it writes to some state variables in a synchronized fashion,

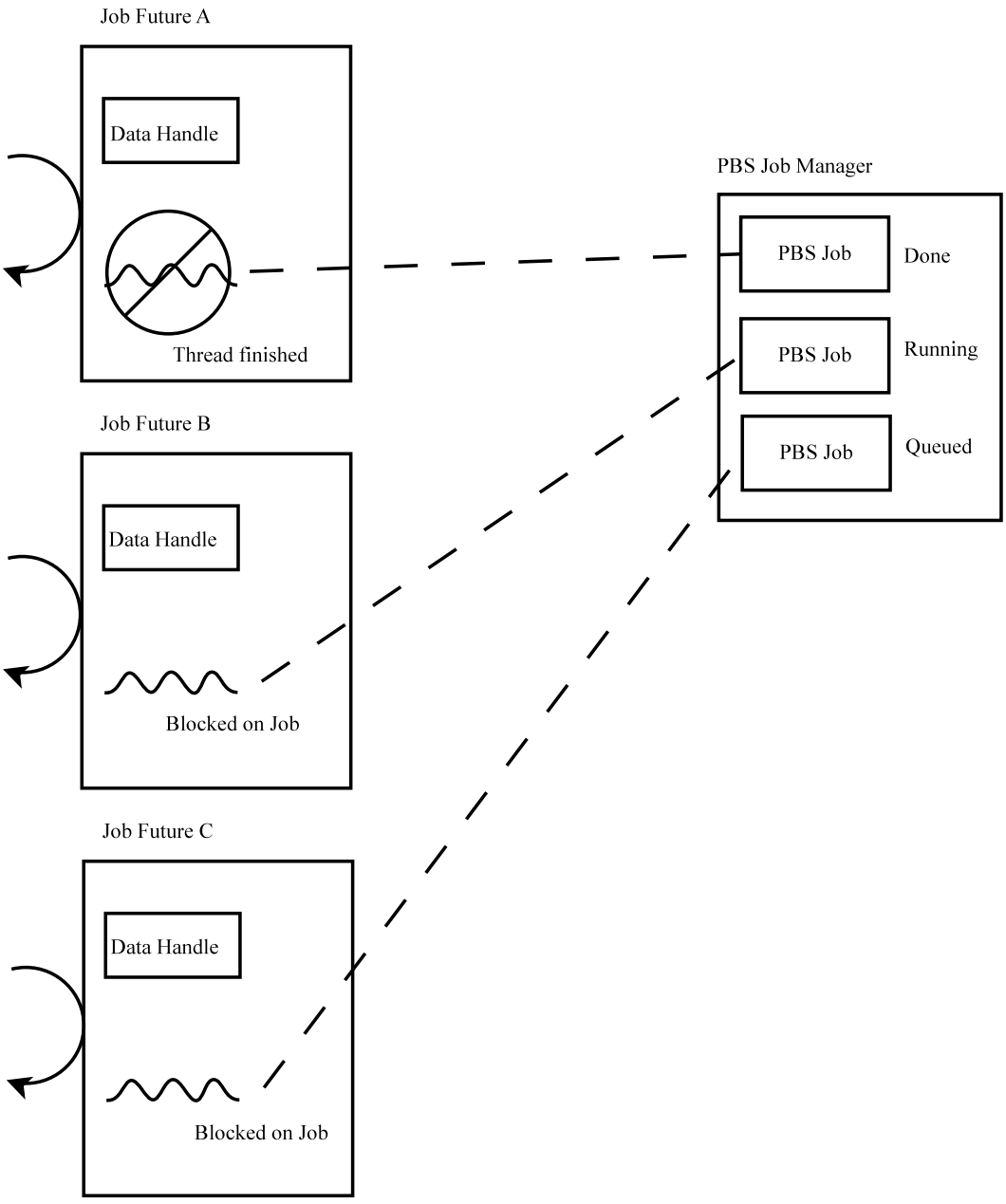

Figure 4.1: Job Future States

using a condition variable. The condition variable allows other threads or the main script to wait on the job future. Once the job future finishes, it can notify the other processes using the condition variable.

Figure 4.1 shows the possible states of a job future. Job future A has completed the job, and its thread has completed executing. Job future B has a running job, and its thread is waiting for it to finish. Job future C has a queued job, and its thread is waiting for it to finish.

4.2 Command Generator

The command generator is implemented as a set of functions. Users use the command generator by calling the `generate_commands` function. The two main functions which implement the command generator's functionality are shown in Figure 4.2. The command generator can be given a filtering function, which must have the same amount of arguments as the number of parameter lists. This function must return true for sets of parameters that should be filtered. The command generator will leave those sets out of the list of commands it creates.

To create the command list, the `generate_commands` function iterates over the Cartesian product of the parameter lists. In each iteration, it checks if the set of parameters should be filtered. If the parameters are not filtered, they are passed to the `simple_command` function along with the command string, which will parse the strings and replace the command arguments with the parameters, initializing any data handles that are specified and adding them to the final command. After this iteration is finished, the final list of commands is returned.

4.3 Job Array

The job array is a class in Jole, inherited from the Python array, with some additional features added to handle job futures. The class definition for the job array is shown in Figures 4.3 and 4.4. There are iterator methods that allow iteration over jobs in different states. As well, there is a custom iterator method, which allows the user to define a filter function that filters out job futures they do not want.

Job arrays also have synchronization commands that act on the array of job futures. The first, `wait_all`, uses the job futures synchronization commands and waits for them all to finish before returning. The second, `wait_some`, uses threads and a condition variable to synchronize on a waiting jobs variable. Each thread waits on a job future and modifies the waiting jobs variable once finished, while the main script waits on the condition variable. Once a thread reaches the proper count, it notifies the main script, and the `wait_some` method returns. `wait_any`, is a special case of `wait_some`, which waits on one job.

```
# Command Generator
def generate_commands(s_command, lists, filter_fun = None):
  commands = []
  string_command = s_command.strip()
  # Iterate over Cartesian product of parameters
  for set in cartesian_product(*lists):
    if filter_fun:
      # Check if parameter set should be filtered
      if not filter_fun(*set):
        commands.append(simple_command(string_command, *set))
    else:
        commands.append(simple_command(string_command, *set))
  return commands

# Parse command
def simple_command(string_command, *args):
  p = re.compile(r'(%[arw])|[ ]')
  split_command = p.split(string_command)
  flat_args = flatten(args)
  string_arg_count = 0
  for x in split_command:
    if x == '%a' or x == '%r' or x == '%w':
      string_arg_count += 1

  if (string_arg_count != len(flat_args)):
    raise "Number of arguments doesn't match"
  count = 0
  for ind in range(0,len(split_command)):
    if split_command[ind] == '%a':
      split_command[ind] = flat_args[count]
      count += 1
    elif split_command[ind] == '%r':
      split_command[ind] = DataHandle(flat_args[count], "r")
      count += 1
    elif split_command[ind] == '%w':
      split_command[ind] = DataHandle(flat_args[count], "w")
      count += 1
    elif split_command[ind] is None:
      split_command[ind] = " "
  return split_command
```

Figure 4.2: Command Generator

```
from job_future import JobFuture
import threading
from time import sleep

class JobArray(list):
  def __init__(self, job_manager):
    super(JobArray,self).__init__()
    self.__threads = []
    self.__finish_cond = threading.Condition()
    self.job_manager = job_manager

  def __del__(self):
    for thread in self.__threads:
      thread.join()

  def custom_iterator(self, filter):
    for job_future in self:
      if not filter(job_future):
        yield(job_future)

  def add_job_future(self, job_future):
    self.append(job_future)

  def add_command(self, command):
    self.append(self.job_manager.new_job_future(command))

  def add_commands(self, command_ary):
    for command in command_ary:
      self.add_command(command)

  def wait_all(self):
    for job_future in self:
      job_future.wait()
```

Figure 4.3: Job Array Class: Part 1

```
def wait_some(self, count):
  self.__jobs_waiting = count

  for job_future in self:
    self.__threads.append(threading.Thread(
         target=self.__wait_future, args=(job_future,)))
    self.__threads[-1].start()

  self.__finish_cond.acquire()
  self.__finish_cond.wait()
  self.__finish_cond.release()

def wait_any(self):
  self.wait_some(1)

def __wait_future(self, job_future):
  job_future.wait()
  self.__finish_cond.acquire()
  self.__jobs_waiting -= 1
  if self.__jobs_waiting == 0:
    self.__finish_cond.notify()
  self.__finish_cond.release()

def finished_jobs(self):
  for job_future in self:
    if job_future.is_finished():
      yield(job_future)

def queued_jobs(self):
  for job_future in self:
    if job_future.is_queued():
      yield(job_future)

def running_jobs(self):
  for job_future in self:
    if job_future.is_running():
      yield(job_future)

def add_monitor(self, m_func):
  for job_future in self:
    job_future.add_monitor(m_func)
```

Figure 4.4: Job Array Class: Part 2

4.4 Data Handles

Data handles are implemented as a class in the Jole library. When initialized, data handles interact with the global workflow space, in order to keep track of files. When initializing a read data handle, it will first check if the same data handle is in the global workflow space. If it is, it will return a data handle with that information. Otherwise, it will check if the file exists and return an error if it does not. We must check if the file exists in the global workflow first, because if the file was created through a write data handle, it may exist at a different path due to name conflicts.

Write data handles are initialized by finding a non-conflicting name in the same path as the original file name. Once this is done, it is returned.

4.5 Job Manager

The job manager keeps an internal list of the jobs and their properties. In addition, it keeps track of submitters for each cluster. Once a job is submitted, the job manager looks at dependencies. If they are not met, the job manager will wait to schedule the job until all its dependencies are met. Once the dependencies for the job are met, the job manager will notify the submitters that a job is ready, and the submitters will submit placeholders to their underlying batch scheduler.

The job manager also maintains a socket server, which is used to communicate with the placeholders. Each placeholder will communicate with the socket server in order to grab job information and change the state of the job. The job manager keeps track of active jobs in order to produce communicators for the placeholders. When a remote monitor asks for a placeholder communicator, the job manager will return one immediately if the job is active. Otherwise, the job manager will wait until the job becomes active. The socket server also receives messages from each placeholder as they are running, which are used to notify the job manager that the placeholders are still running. If these messages stop coming in from a placeholder, the job manager assumes that the machine has died, and will reschedule the job on another machine.

In addition, the job manager maintains information about each job as they are executed on the cluster. This information is needed by remote monitors so that they are notified once their job starts and to create a placeholder communicator for them to use to communicate with the job.

4.6 Submitters

Submitters are scripts written to interface with the batch scheduler of a cluster. They are mainly used to submit jobs, since the placeholders will still communicate with the job manager to grab jobs and return their results. The job manager communicates with these wrappers on different hosts to submit jobs.

Submitters watch a global file in order to determine whether to run jobs on their cluster. If the server becomes unavailable, the submitter must remove all jobs from the cluster. To do this,

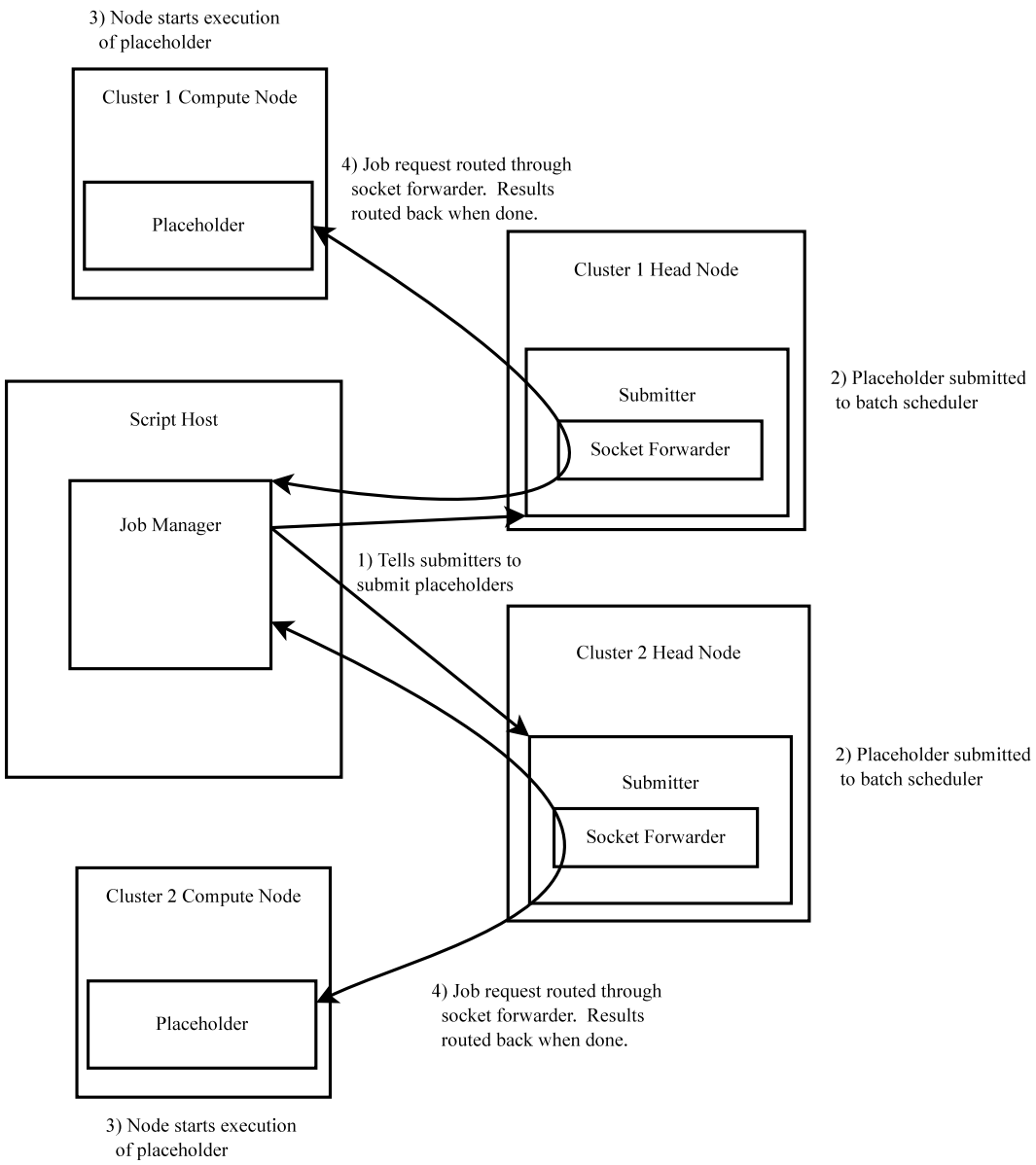

Figure 4.5: Using Multiple Batch Schedulers

submitters maintain a list of job identification numbers for submitted placeholders. When a cluster becomes unavailable, the submitter kills all jobs using the batch scheduler interface.

The submitters are executed by logging into the nodes through SSH. In order to minimize user intervention, `ssh-agent` or passwordless SSH keys should be used. Usage of the Trellis Security Infrastructure (TSI) [13] could be used to ease setup and administration of SSH keys.

On some clusters, it may not be possible for the compute nodes to communicate outside of the cluster. In this case, we need a way to forward their traffic to the job manager. This is accomplished using a socket forwarder. The socket forwarder runs in parallel with the submitter. When the submitter receives a message to submit a placeholder, it gives the placeholder the address and port number of the socket forwarder. When the placeholder communicates with the socket forwarder, the socket forwarder will open a connection to the job manager and relay traffic between them.

4.7 Placeholder

The placeholder is implemented in a script, which is executed by the batch scheduler of a cluster. The placeholder has a synchronized state condition variable, which controls the execution of the placeholder. When initialized, the placeholder starts a socket server to handle monitor requests. It then communicates with the job manager server to get the job information. The job and any placeholder local monitors are started in threads. The placeholder waits on the state variable to change to the finished state before exiting. The state will change to finished if the job finishes, or if one of the monitors decides to end execution early.

When the placeholder grabs a job from the job manager, it scans for any data handles or monitors. If a data handle is found, the placeholder verifies the data handle is accessible and has the proper permissions in order to continue. If there are file access problems, the placeholder will attempt to `scp` the input files in and write output files to an accessible directory. It will try to `scp` the output files back to the host node when the job is done. If a monitor is found, it initializes the monitor and starts it.

4.8 Remote Monitor

The remote monitor is implemented as user written function which uses a placeholder communicator to interact with the placeholder's socket server. The communicator has specific functions that allow the monitor to perform different functionality, including grabbing file output and killing jobs.

The required functionality is added to the function, then passed to the job future. The job future creates a thread to start the monitor, which first waits for the job to be submitted. Once the job is submitted, the remote monitor asks the job manager for a communicator to that job. This call will block until the job has started running and communicated with the job manager. Once this has finished, the remote monitor will run.

4.9 Local Monitor

Local monitors are implemented as a series of classes that are added to the job futures when they are created. The monitor information is formatted to send across the network, and is sent to the placeholder when it requests a job. The monitors are reinitialized at the placeholder using the `init_monitor` function, which inspects the information and reinitializes the monitor based on the information. These monitors are run in separate threads, and change the state condition variable when each succeeds.

For the most part, the monitors have fairly static functionality, and will carry it out once it is started. However, the custom monitor allows the user to submit their own function to run at the placeholder. To initialize a custom monitor, the user must add the `globals` hash to the arguments. This lets the custom monitor inspect the namespace of the main Python script in order to get the proper dependencies needed to run the code.

Figure 4.6 shows the Python code necessary to gather all the dependencies for a custom monitor function. The first step is to get the source code for the function that will be run. This is done using the Python inspect module. Then the custom monitor iterates through the `globals` hash in order to grab dependencies. For modules found in the `globals` hash, an import statement is added. User created modules only show up as strings, so we must check `sys.modules` to see if any user created modules need to be imported. Lastly, functions in the `globals` hash are looked at. If they are from the main script, their source code will be sent along with the user created method. If they are imported from other modules, a '`import x from y`' statement is added, where x is the function and y is the module.

4.10 Concluding Remarks

In this chapter, we have described the implementation of Jole and showed some of the code used to implement the functionality of Jole's components. Jole has been implemented as standard Python libraries and as two scripts, using Python.

```
# Convert to string when first creating
text_ary = inspect.getsource(function).split("\n")
# Change function name
text_ary[0] = "def mon_func(self):"

# Grabbing necessary function code
for x in my_globals.keys():
  match = re.search(r'^__.+__$',x)
  if match:
    continue
  if type(my_globals[x]) is types.ModuleType\
    and x != '__builtins__':
    # Add imports
    text_ary.insert(0, "import %s" % x)
  elif isinstance(my_globals[x], basestring)\
      and sys.modules.has_key(my_globals[x]):
    text_ary.insert(0, "import %s" % my_globals[x])
  elif type(my_globals[x]) is types.BuiltinFunctionType\
      or type(my_globals[x]) is types.FunctionType:
    try:
      f_mod = my_globals[x].__module__
      f_name = my_globals[x].__name__
      if f_name == function.__name__:
        continue
      if f_mod == "__main__":
        f_text = inspect.getsource(my_globals[x])
        text_ary.insert(0, f_text)
      else:
        text_ary.insert(0, "from %s import %s" % (f_mod, f_name))
    except Exception, e:
      text_ary.insert(0, str(e))
      text_ary.insert(0, "# %s failed" % x)

self.__function_text = "\n".join(text_ary)
```

Figure 4.6: Gathering Modules and Functions Needed for Local Monitor

Chapter 5

Evaluation

We described the concepts and implementation of Jole in the previous two chapters. In this chapter, we evaluate Jole with respect to our three contributions using GAFolder workloads. We show:

1. Dynamic job execution allows jobs to be monitored and operations performed on them while they are running. We show that this can be used with GAFolder to terminate jobs early based on user-specified criteria. By ending some jobs early, cluster time can be freed for other jobs. In comparison, jobs run with static execution are all run until completion.

2. Dynamically changing workflows can be implemented in Jole, using standard control flow in Python and global information from the workflow. We show that Jole can be used to implement a GAFolder workload that loops based on the improvement in scores between generations. The workflow uses the best global structures found in each generation as the basis for jobs in the next. This type of workflow requires scripting, and would be difficult to implement with a static language.

3. Workloads that adapt to available resources can be implemented using Jole. This can allow a user's workload to take advantage of shared resources. We demonstrate Jole's ability to adapt to available resources, and show that this ability can be used to implement a workload which adjusts the number of jobs it runs based on the available resources. We show that adjusting the number of jobs to the available resources can be used to improve the structure prediction results of a GAFolder workload.

5.1 Evaluation Methodology

We test Jole using GAFolder, a protein structure predictor developed at the University of Alberta. GAFolder uses genetic algorithms to evolve and choose a final structure, by minimizing the energy score of the torsion angles of the predicted protein. Each instance of GAFolder uses a different random seed for the random number generator, and will search in different areas. Instances of

GAFolder may converge to local minima, so multiple instances are executed in order to perform a more thorough search.

We use GAFolder for our evaluation, because it is in production use, it is developed independently, and it serves as an external case study for Jole. GAFolder has several properties that can be improved using our system.

1. GAFolder is currently run for a set number of iterations. By monitoring GAFolder jobs as they are running, it is possible to selectively stop jobs based on the rate of progress they are making.

2. Protein structure scores can be further improved by running subsequent GAFolder instances with the most promising global structures as input. This allows the GAFolder instances to better search the space around the most promising structures.

3. Different instances of GAFolder will search different areas of the protein structure space. Therefore, running more instances of GAFolder will search a larger area of the protein structure space, and can result in higher quality structures.

We test Jole using several GAFolder workloads, created to demonstrate the contributions of Jole. Each workload runs several instances of GAFolder to refine the structure of the ubiquitin protein, one of the example proteins provided by GAFolder. For timing measurements, each workload was run five times.

All experiments are run on the test environment shown in Figure 5.1. These two test clusters have been created on separate machines within a preexisting 20-node cluster using TORQUE PBS. All machines are connected by gigabit Ethernet. Each cluster is made up of three nodes containing a total of 12 cores. Machines A1 and B1 only schedule jobs on two of the four available cores, so 10 cores are available for job scheduling in each cluster. Each compute node contains a quad-core Intel Xeon 5160 processor, with a clock speed of 3.0GHz. Each compute node also contains 12GB of RAM.

5.2 Dynamic Job Execution

Without the use of Jole, GAFolder runs for a set number of iterations, rather than using a convergence criteria. This is done for simplicity. By actively monitoring these jobs while they are running, we can stop the execution of jobs that are not improving their energy score. If the application has the exclusive use of a cluster, new jobs could be started to replace the stopped ones, since each execution is independent and searches a different structure space. This could potentially allow GAFolder to better focus its search. In the case of a shared-use cluster, stopping these jobs can free up resources for other users, which is what we show here. Recall that, in a batch scheduled environment, jobs

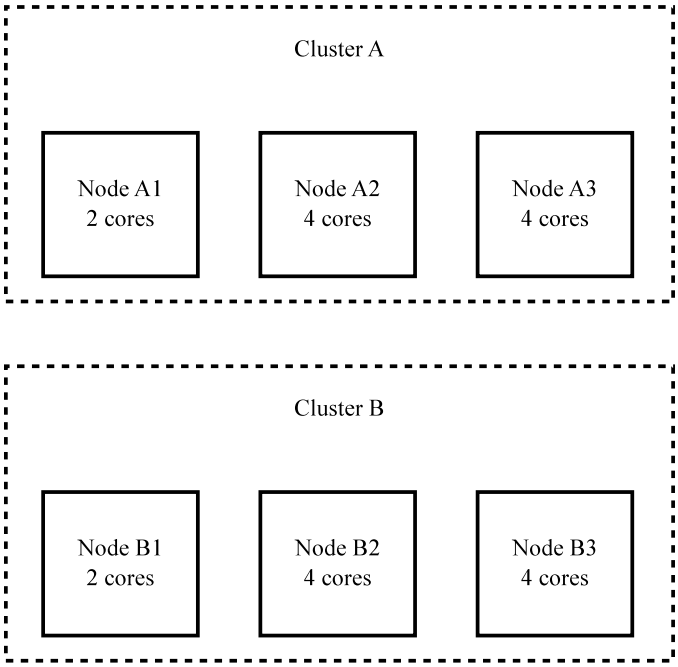

Figure 5.1: Cluster Testing Environment

are generally run to completion. While the status could be monitored by the application itself, Jole allows you to create monitors for many applications.

To test dynamic job execution, we ran two GAFolder workloads on cluster A, shown in Figure 5.1. The results are shown in Table 5.1. The workflow graph for these workloads is illustrated in Figure 5.2. Each workload consists of 20 instances of GAFolder, each given the same initial protein structure. After the workload is finished running, the best protein structures are returned, based on the evaluation score of GAFolder. GAFolder uses different random-number seeds to search the protein structure space. To ensure reproducibility, we preset the random seed for each instance, keeping them the same in both workloads.

The static workload runs standard instances of GAFolder. Each instance runs 300 iterations of the genetic algorithm, as that is a number of iterations that has been used in practice by the developers of GAFolder. This gives us a baseline from which to compare the results of the dynamic local workload.

The dynamic local workload runs standard instances of GAFolder with a monitor attached to each one. The monitor watches the output of the GAFolder job, determining the standard deviation of the last 25 iterations. By determining the standard deviation of the last 25 iterations, we can determine how much the score is changing. If the standard deviation of those iterations is less than 0.0001, the monitor will end the job, as there is little change in score happening. Note that these GAFolder instances are identical to the instances in the static workload, except for the added monitors.

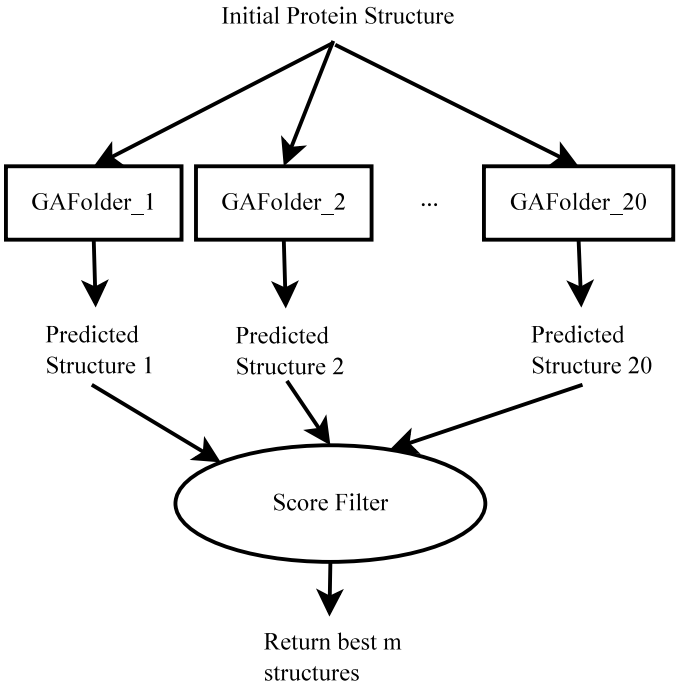

Figure 5.2: GAFolder: Workflow

From the results for the static workload and the dynamic local workload in Table 5.1, we can see that:

1. Dynamic job execution reduces the average number of iterations from 300 to 204.55, leading to shorter job times and shorter makespan. Compare the average job time of 1424.99 seconds for the static workload to 969.45 seconds for the dynamic local workload. The makespan is reduced from 2889.94 seconds to 2708.67 seconds.

2. Dynamic job execution leads to a reduction in accumulated time, which is the sum of the running times of all GAFolder instances. A reduction results in more time on the cluster for other jobs. Of course, this improvement is a direct result of shorter job times. Note that the accumulated time for the static workload is 28478.87 seconds, while the dynamic local workload uses 19375.11 seconds of accumulated time.

3. However, it should be noted that the score for the dynamic local workload is slightly worse, with a score of -32.97 versus a score of -33.32 for the static workload. Recall that lower scores are better, as GAFolder performs an energy minimization. This occurs due to ending jobs early. However, the score when using dynamic job execution can be improved by running a global workload, discussed in Section 5.3.

While there is not a large decrease in makespan, the use of dynamic job execution has resulted in a 6% reduction. The main benefit is the reduction in the accumulated time, as a result of the shorter

job times. The average job time for the workload with dynamic job execution is 32% lower, which corresponds to the 32% lower accumulated time for the workload with dynamic job execution. This is a significant reduction that could be used to allow jobs from other users to run, or to run additional GAFolder jobs to further search the protein space. The dynamic local and global workload numbers are discussed in Section 5.3.

Workload	Makespan s	Accum Time s	Avg Job Time s	Avg Iters	Score
Static (20)	2889.94 (3.11)	28478.87 (21.75)	1424.99	300	-33.32
DL (20)	2708.67 (5.37)	19375.11 (21.21)	969.45	204.55	-32.97
DLG (10)	1276.26 (11.65)	9147.47 (32.17)	303.78	60.10	-33.12
DLG (20)	2950.73 (24.14)	23566.04 (220.50)	236.14	44.86	-33.52

DL refers to the dynamic local workload and DLG refers the dynamic local and global workload.
For the workload column, the number in parentheses refers to the number of GAFolder iterations run (per generation in the case of global workloads). All timing values are from the average of five runs, and display the standard deviation in parentheses.
The score column shows the best structure score from any instance in a workload.

Table 5.1: GAFolder: Comparison of workloads

The GAFolder runtime distribution for each workload is shown in Figures 5.3 and 5.4. In Figure 5.3, we can see that there is little variation in runtimes of GAFolder for the static workload, as compared to the dynamic job execution workload. The runtime varies between 1410.75 and 1444.12 seconds. In Figure 5.4, we can see that there is more runtime variance between jobs in the dynamic workload. There are a few jobs that run almost as long as in the static workload, but for the most part, jobs were terminated significantly earlier.

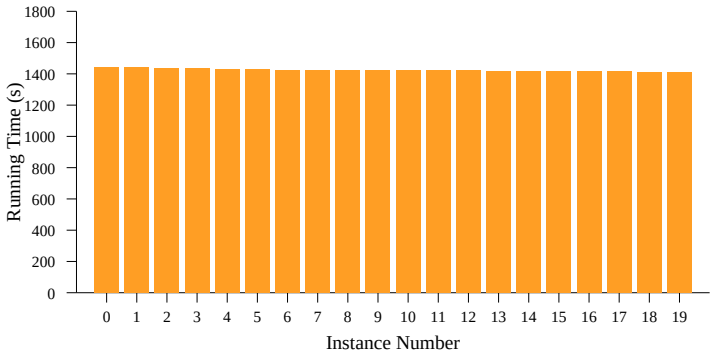

Figure 5.3: GAFolder Runtime Distribution: Static Workflow

5.3 Dynamic Workflow

As stated in Section 5.2, multiple instances of GAFolder are executed to better search the structure space of a protein. This process can be repeated on the best proteins found, allowing search to be

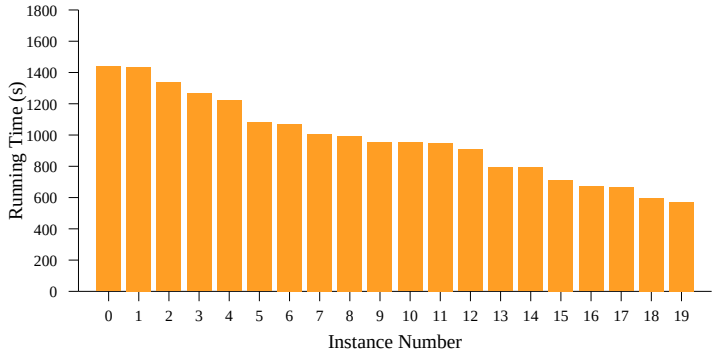

Figure 5.4: GAFolder Runtime Distribution: Dynamic Local Workflow

focused in the most promising area. A global workload, shown in Figure 5.5, repeats the workload in Section 5.2 with the best proteins from each generation. The first iteration of the workload is identical to the dynamic local workload in Section 5.2. After the first iteration is done, the best two structures are used as input for the new GAFolder instances, which will further search the structure space around the promising structures. Once the improvement in score between generations is below our threshold, the workflow will finish. Note that the dynamic local workload in Section 5.2 only alters execution locally for each instance of GAFolder, while the dynamic local and global workload looks at the results of each job in order to determine whether to continue executing additional generations.

Workflows that must use global information from job executions to decide on control flow are difficult to program in a static workflow specification language. Scripting is required in order to check the results of jobs between each generation, and decide whether to run another. If there has not been enough of an increase in score to support running more instances, the best protein found so far is returned. With Jole, this dynamic workflow can be easily specified.

We test the GAFolder dynamic local and global workload on cluster A, shown in Figure 5.1. Results are shown in Table 5.1. Each generation of the workflow is identical to the dynamic local workflow in Section 5.2, except for a more stringent cutoff for ending jobs. This means that GAFolder instances will be ended earlier when making slow progress, as compared to the dynamic local workload. The following parameters were used:

1. Two workloads are run, with one having 10 instances per generation and the other with 20. For the first test, the number of instances is less than in the dynamic local workload. This means that the search is not as exhaustive in the initial generation, but more generations will run, focusing on the best structures from each generation.

2. A standard deviation of less than 0.001 over 25 iterations ends job execution. This cutoff is more stringent than it is in the dynamic local workload. By using a more stringent termination

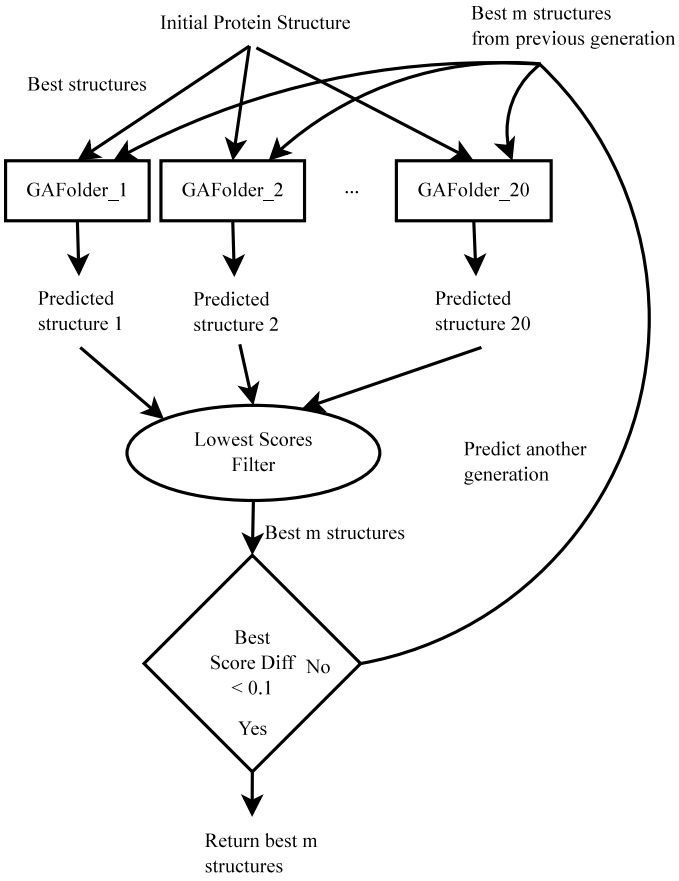

Figure 5.5: GAFolder: Dynamic Local and Global Workflow

criteria, GAFolder jobs will be ended earlier, resulting in lower initial scores. Rather than improving the score through more initial iterations, we can improve it through more generations.

3. A decrease in score between generations of less than 0.1 ends the workflow loop. For example, if the best structure score in the first generation was -16 and the best in the second generation was -16.1, than more generations would be run. If the best structure in the second generation was -16.05, than no more generations would be run.

4. The two most promising protein structures are carried over between generations, to be used as the initial structures for GAFolder instances.

From the results in Table 5.1, we can see that:

1. Dynamic workload specification allows the dynamic local and global workload to search more exhaustively around promising proteins, resulting in a better score than when using dynamic job execution alone. Using more restrictive cutoffs for GAFolder instances allows the workflow to more aggressively trim instances that are not showing improvement. Compare the score of -33.12 for the dynamic local and global workload with 10 instances to -32.97 for the dynamic local workload. The dynamic local and global workload runs in 1276.26 seconds, as compared to 2708.67 seconds for the dynamic local workload.

2. With the dynamic local and global workload for GAFolder running 20 instances per generation, an energy score better than the final one produced by the static workload is achieved, in only slightly more time. Notice a score of -33.52 for the dynamic local and global workload with 20 instances and a score of -33.32 for the static workload. The makespan for the dynamic local and global workload with 20 instances is slightly higher. However, the dynamic local and global workload with 20 instances reached a score of -33.44 after 2700 seconds, producing a better score in less time than the static workload. Also, the accumulated time of the dynamic local and global workload with 20 instances is significantly less. Compare the accumulated time of 23566.04 seconds for the dynamic local and global workload with 20 instances to 28478.87 seconds for the static workload.

When using 10 instances of GAFolder in each generation, the dynamic local and global workload finishes approximately 55% faster than the static workload, and 52% faster than the dynamic local workload. The accumulated time is also an improvement, scoring about 67% shorter than the static workload and 52% shorter than the workload with dynamic job execution. The decrease in accumulated time results in more cluster resources for other users or jobs. However, the score is slightly worse than the static workload, although it is an improvement over the dynamic local workload.

When using 20 instances of GAFolder in each generation, the dynamic local and global workload was able to produce a better quality score. The increase in instances per generation resulted in a

longer makespan, which was about 2% longer than the static workload. The accumulated time is about 21% longer than the dynamic local workload and 17% shorter than the static workflow. While this workload took slightly longer, it was able to produce a structure score that is 0.2 better than the static workload.

Figures 5.6 show 5.7 show the average number of iterations performed by GAFolder in each generation for each workload. The first generation has the largest average number of iterations by a significant margin. The next generations averages are between 20 and 30 iterations. This occurs because the GAFolder instances have a higher rate of improvement in the first generation, while in later generations improvement in score comes at a slower rate.

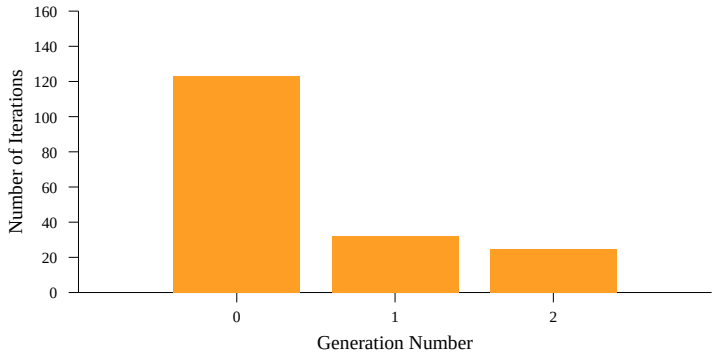

Figure 5.6: Average Number of Iterations Per Generation: Dynamic Local and Global Workload (10 instances)

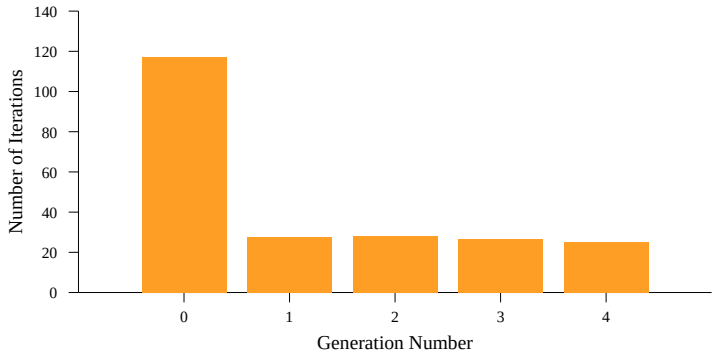

Figure 5.7: Average Number of Iterations Per Generation: Dynamic Local and Global Workload (20 instances)

5.4 Dynamic Infrastructure

In many organizations, there are resources which are often idle. For example, within an organization, several groups may have their own clusters. Some groups may only need to run large workloads occasionally. However, when they run their workloads, they want to be able to fully utilize their cluster, in order to finish running their workload as quickly as possible. When these clusters are idle, it would be beneficial to be able to execute other users jobs on them. One goal of Jole is to allow users to run workloads across cluster resources. With Jole, users can run their workloads completely on shared resources. Jole will run jobs on the shared clusters as they become available, and kill jobs when the cluster is used by the owner. The following workload demonstrates how Jole allows users to run their workload across different cluster resources.

To test dynamically running a workload across cluster resources, we use the GAFolder workflow from Section 5.3, modified to search more extensively. Specifically, we run 20 instances of GAFolder per generation and end execution when the difference in score between generations is less than 0.02. After 15 minutes, the submitter running on cluster B is notified to stop running jobs. 60 minutes later, the same submitter is notified that it can resume running jobs. We are running the workload on the two test clusters, cluster A and cluster B, shown in Figure 5.1. Our submitters are configured to remove all jobs from the queue, and kill any running jobs, immediately after receiving a notification to stop running jobs. For example, if job 1 is running on cluster B and the submitter for B receives a notification to stop running jobs, job 1 will be killed. It will then be put into the failed jobs queue, which has priority over jobs that have not run yet. Job 1 will run on cluster A as soon as it reaches the front of the failed jobs queue.

Figure 5.8 illustrates the usage of each cluster throughout the workload. Cluster A has relatively stable usage throughout the execution of the workload. Reductions in the number of instances running on both clusters occur mainly due to barriers between generations of the GAFolder calculations, when only a few GAFolder instances are running. Cluster B has 0 instances running on it, from 15 minutes to 75 minutes. This is due to the submitter on cluster B stopping jobs after receiving stop notification. At 75 minutes the submitter receives a notification to continue running jobs, and jobs start running on cluster B again.

Figure 5.9 shows the number of completed instances on clusters A and B. We can see that cluster A has a mostly linear graph, finishing jobs at a stable pace. Cluster B, however, reaches a plateau after 15 minutes. Again, this happens because the submitter for B was told to stop running jobs. Once the submitter is allowed to run jobs again, after 60 minutes, the number of completed instances rises relatively linearly.

5.4.1 Adjusting Workflow Based On Total Resources

Distributing jobs among available resources as they become available is useful, but with Jole it is also possible to dynamically alter the workload based on the available resources. Suppose a user

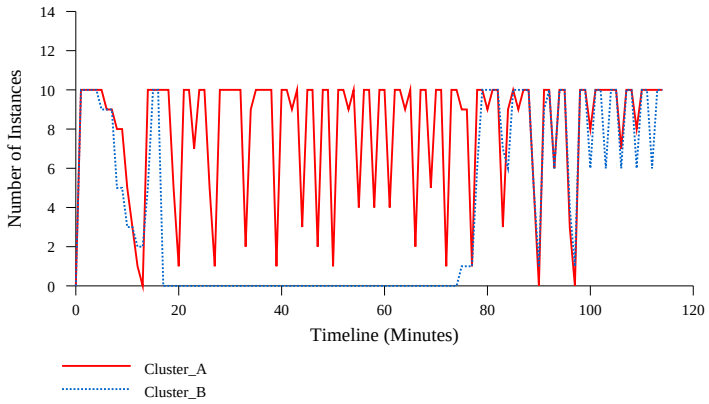

Figure 5.8: Dynamic Infrastructure: Number of Instances Running on Each Cluster

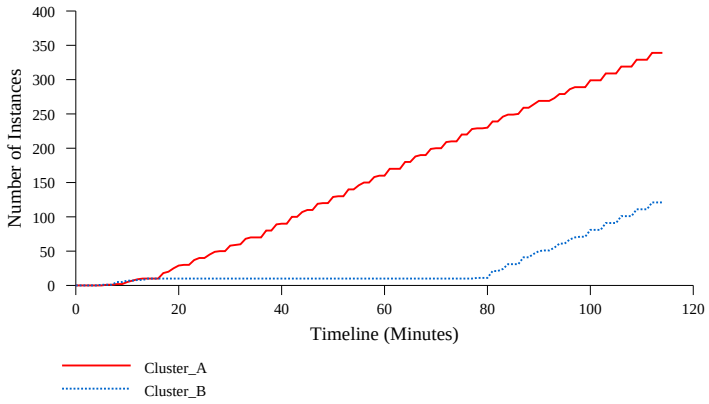

Figure 5.9: Dynamic Infrastructure: Instances Finished on Each Cluster

wants to run a workload that finishes within a certain period of time. If they are running a workload on heterogeneous resources, they could roughly estimate running time based on the number of jobs. If more resources are available, they could run more jobs in the same amount of time.

We test this using the same workflow from earlier in this section, except for a small change. Rather than running a set amount of GAFolder instances in each generation, we now check the total number of processors available, setting the number of instances to double the amount of processors. This allows our workflow to do a more exhaustive search in each generation if there are more resources available. The dynamic local and global workflows in Section 5.3 are focused on improving score with less resources. In this section, we show that score can be improved with more resources.

We run the workload twice, with the same base infrastructure available, cluster A. The constant workload has no additional resources available to it and will run 20 instances per generation on cluster A until the workload has finished. The adjusting workload has opportunities to use cluster B for the first 15 minutes and after 75 minutes into the workload. Remember that both cluster A and B have 10 cores available. This results in twice as much computing power to boost throughput for the adjusting workload when cluster B is available.

Figures 5.10 and 5.11 show the number of instances running and completed on cluster A over time for the constant workload. Cluster A has a stable number of instances running on it throughout the workload. The number of instances finishing in the first 23 minutes increases at a much slower rate than later generations, as each instance is running for more iterations than instances in later generations. After the first 23 minutes, the number of instances finished increases at a quicker rate.

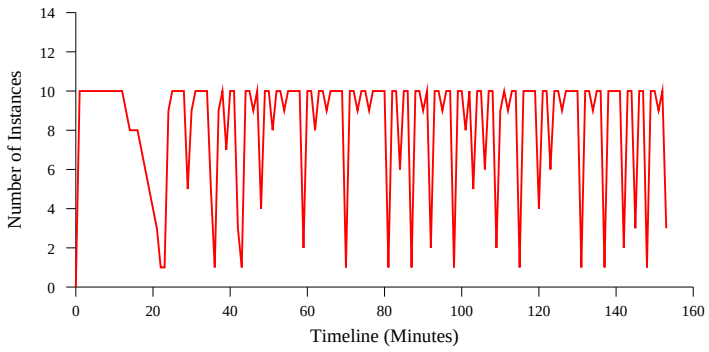

Figure 5.10: Constant Workload: Instances Running on Cluster A

Figures 5.12 through 5.13 show the number of instances running and finished on clusters A and B for the adjusting workload. Cluster A has a similar slope for finished instances over the first 23 minutes, which then increases as instances are run for less iterations. From the figures for cluster B, we can see that the number of finished instances increases in between 0 and 15 minutes and after 75 minutes into the workload.

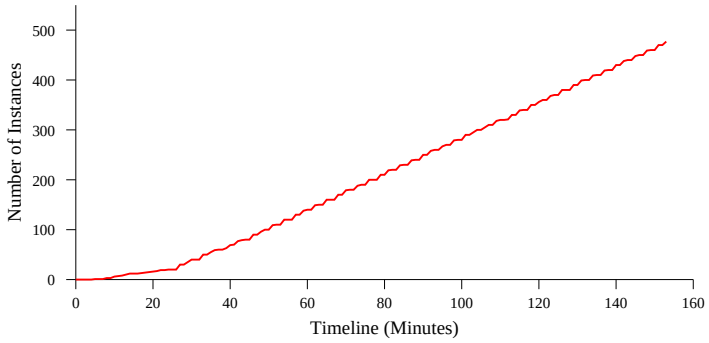

Figure 5.11: Constant Workload: Instances Finished on Cluster A

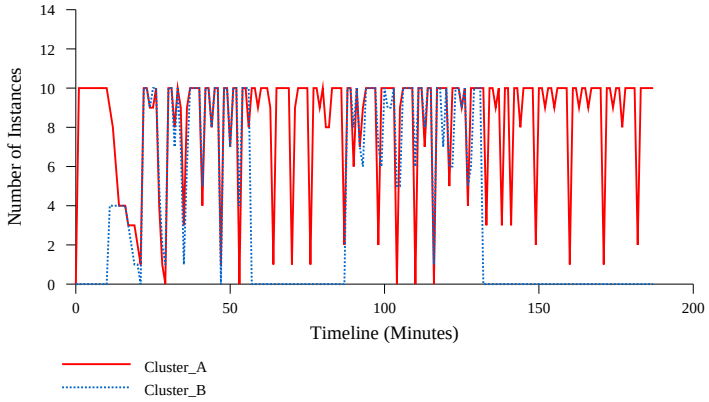

Figure 5.12: Adjusting Workload: Instances Running on Each Cluster

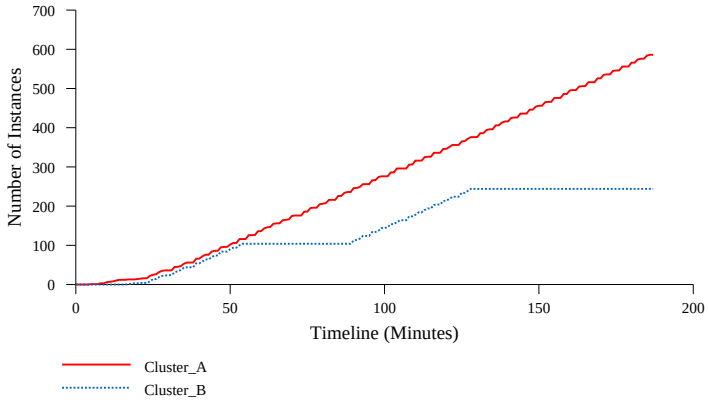

Figure 5.13: Adjusting Workload: Instances Finished on Each Cluster

The best GAFolder score over time for each workload is shown in Figure 5.14. These scores are gathered by finding the globally best structure score every minute. The scores from the adjusting workload and constant workload trade places over the first 82 minutes. However, after 75 minutes, the adjusting workload has more resources available and starts running more instances. The adjusting workload maintains a lead over the constant workload until both are finished. The constant workload stops running after 153 minutes, because the score improvement criteria for running more generations was not met. The adjusting workloads runs for slightly longer, as it meets the minimum score improvement criteria for more generations.

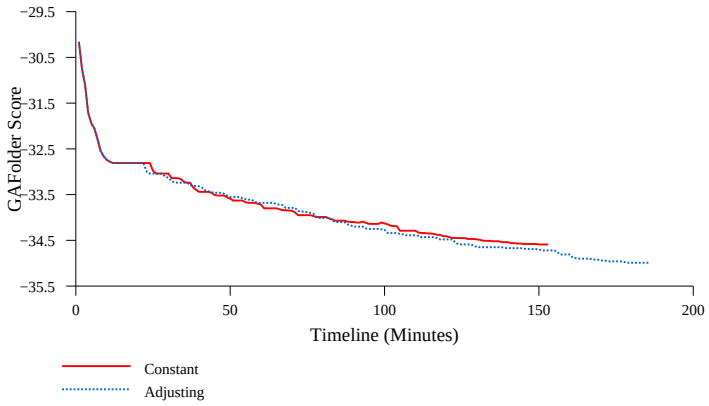

Figure 5.14: GAFolder Scores Over Time in the Constant and Adjusting Workloads

Table 5.2 compares the best scores between workloads at the end of each generation. From the table, we can see that each workload achieves the same score after the first generation. This happens because each workload runs the same instances in the first generation, producing the same results. Subsequent generations show the adjusting workload gaining a lead in the first generation and losing it in the second generation. The adjusting workload regains this lead in the ninth generation. The adjusting workload maintains this lead for the all remaining generations. Note that generation times are relatively close for the first six generations. During generation 6, cluster B becomes unavailable, so any jobs that were running on it needed to be rerun. The same situation occurs in generation 18. The constant workload only ran 24 generations, as the threshold for continuing was not met in generation 23.

5.5 Concluding Remarks

In evaluating Jole with GAFolder workloads, we have shown that the contributions of Jole can be used to improve upon the standard GAFolder workflow. Specifically:

1. *Dynamic Job Execution*

Generation	Constant Workload			Adjusting Workload		
	Score	Time (m)	Instances	Score	Time (m)	Instances
0	-32.81	23	20	-32.81	21	20
1	-33.04	29	20	-33.11	29	40
2	-33.24	36	20	-33.24	35	40
3	-33.44	43	20	-33.35	41	40
4	-33.52	48	20	-33.46	47	40
5	-33.63	53	20	-33.55	52	40
6	-33.70	59	20	-33.68	64	40
7	-33.80	64	20	-33.79	70	20
8	-33.85	70	20	-33.90	76	20
9	-33.95	75	20	-34.01	81	20
10	-33.99	81	20	-34.10	86	20
11	-34.07	86	20	-34.20	92	40
12	-34.11	92	20	-34.25	98	40
13	-34.14	97	20	-34.34	104	40
14	-34.19	103	20	-34.39	110	40
15	-34.29	109	20	-34.43	115	40
16	-34.35	114	20	-34.48	121	40
17	-34.41	120	20	-34.59	127	40
18	-34.45	125	20	-34.65	138	40
19	-34.49	131	20	-34.67	143	20
20	-34.52	136	20	-34.69	149	20
21	-34.56	142	20	-34.72	154	20
22	-34.58	147	20	-34.81	160	20
23	-34.59	153	20	-34.90	165	20
24	-	-	-	-34.94	171	20
25	-	-	-	-34.96	176	20
26	-	-	-	-34.99	181	20
27	-	-	-	-34.99	187	20

Table 5.2: GAFolder Scores After Each Generation in the Constant and Adjusting Workloads

In Section 5.2, we show that using monitors to dynamically terminate GAFolder instances showing a slow rate of improvement, rather that waiting for the standard number of iterations to finish, can free up significant time on the cluster. However, this happens at the expense of some score quality.

2. *Dynamic Workflow*

   In Section 5.3, we show that by using global information from jobs in a workload running 10 instances per generation, it is possible to get decent scores in much less time than the static workload. In addition, we also show that by using the global information from a workload running 20 instances per generation, it is possible to get better scores than the static workload, while not using as much cluster resources.

3. *Dynamic Infrastructure*

   In Section 5.4, we show that Jole is able to take advantage of additional resources as they become available. Furthermore, we show that these additional resources may be used to run extra GAFolder instances in each generation, generating better scores.

Chapter 6

Concluding Remarks

As scientific computing problems grow larger, methods for executing these problems become more important. Batch schedulers are often used to schedule and execute these scientific workloads. Generally, batch schedulers are static with respect to three properties: interacting with a job while it is running, modifying a workload while it is running, and changing the workload infrastructure while it is running. Many workloads can benefit from allowing dynamic changes in one or several of these properties.

We introduce Jole, a library that enables dynamic changes in these three properties. Monitors allow interaction with jobs as they are running. Job futures, job arrays and the Python base of Jole allow modifying workloads as they are running. The architecture of the job manager, submitters, and placeholders enables dynamic changes to infrastructure.

We evaluate Jole using a set of workloads for GAFolder, to test each of Jole's contributions.

1. *Dynamic Job Execution*

   The dynamic local workload demonstrates the usefulness of dynamically interacting with jobs as they are running. By using monitors to end GAFolder jobs that reach plateaus, space can be freed up on the cluster to run other jobs, at the expense of some score quality.

2. *Dynamic Workflow*

   The dynamic local and global workloads demonstrate the usefulness of modifying workloads as they are running, using global information. By monitoring improvements in score between generations, our workload script will run while a fast enough rate of progress is being made. Combining the dynamic workload abilities with more restrictive cutoffs allows the global workload to get better score, while using less cluster resources.

3. *Dynamic Infrastructure*

   The adjusting workload demonstrates the usefulness of dynamic changes in the execution infrastructure. As more resources become available, Jole will run jobs on them. By adjusting

the workload to run more jobs when there is more available resources, it is possible to find better protein structures with GAFolder.

6.1 Future Work

Jole would benefit from more closely integrating the running of jobs in virtual machines (VMs) with the scheduler. While it is currently possible to run VM jobs, Jole would benefit from being able to perform low-level interactions with the VM. Specifically, the ability to suspend/resume VMs would be a great benefit in dealing with dynamic infrastructure. If a cluster becomes unavailable, a simple policy would be to suspend all the VMs, waiting for the cluster to become available again. If any of the suspended jobs still need to be run when the cluster is available again, they would be resumed. However, this policy depends on the availability of enough disk space to suspend the VMs on each node, and still run other jobs. A more advanced policy might migrate machines off the cluster, choosing to move the VMs with most computation performed or by some other criteria.

Jole may also benefit from the use of some of the Trellis infrastructure. Specifically, the Trellis Security Infrastructure (TSI) and Trellis NFS (TNFS) would be useful for running jobs with Jole. TSI is a complete solution for maintaining an SSH overlay, so that user processes can access any other remote host in the overlay. Currently in Jole, users must setup their own `ssh-agent` process on the host node, which must be done to allow submitters to start on the remote nodes. Processes cannot access other remote hosts unless the user has setup passwordless SSH keys. TSI would simply the infrastructure needed to run jobs and move data between nodes in Jole. The use of TNFS would be more secure than the current setup required to automatically copy data to placeholders when it is not available. With the use of TSI, TNFS securely authenticates with other hosts and copies files.

Bibliography

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–10, 1990.

[2] L. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[3] J. Bent, D. Thain, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Explicit control in a batch-aware distributed file system. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, 2004.

[4] M. V. Berjanskii, P. Tang, J. Liang, J. A. Cruz, J. Zhou, Y. Zhou, E. Bassett, C. Macdonell, P. Lu, G. Lin, and D. S. Wishart. Genmr: a web server for rapid nmr-based protein structure determination. *Nucleic Acids Research*, 37(Web-Server-Issue):670–677, 2009.

[5] 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*, 2004.

[6] S. Ghemawat, H. Gobioff, and S. Leung. The google file system. *SIGOPS Operating Systems Review*, 37(5):29–43, 2003.

[7] M. Goldenberg. Trellisdag: A system for structured dag scheduling. Master's thesis, University of Alberta, 2003.

[8] M. Goldenberg, P. Lu, and J. Schaeffer. Trellisdag: A system for structured dag scheduling. In *9th Workshop on Job Scheduling Strategies for Parallel Processing*, pages 21–43, 2003.

[9] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996.

[10] Hadoop. `http://hadoop.apache.org/`.

[11] 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*, pages 59–72, 2007.

[12] R. H. Halstead Jr. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[13] M. Kan, D. Ngo, M. Lee, P. Lu, N. Bard, M. Closson, M. Ding, M. Goldenberg, N. Lamb, R. Senda, E. Sumbar, and Y. Wang. The Trellis Security Infrastructure: A Layered Approach to Overlay Metacomputers. In *The 18th International Symposium on High Performance Computing Systems and Applications*, 2004.

[14] L. Kornstaedt. Alice in the land of oz: An interoperability-based implementation of a functional language on top of a relational language. In *Workshop on Multi-Language Infrastructure and Interoperability*, 2001.

[15] M. A. Larkin, G. Blackshields, N. P. Brown, R. Chenna, P. A. McGettigan, H. McWilliam, F. Valentin, I. M. Wallace, A. Wilm, R. Lopez, J. D. Thompson, T. J. Gibson, and D. G. Higgins. Clustal w and clustal x version 2.0. *Bioinformatics*, 23(21):2947–2948, 2007.

[16] H. Lieberman. Concurrent object-oriented programming in act 1. *Object-oriented concurrent programming*, pages 9–36, 1987.

[17] P. Lu. The Trellis Project. http://www.cs.ualberta.ca/~paullu/Trellis/.

[18] TORQUE Resource Manager. http://www.clusterresources.com.

[19] Open MPI. http://www.open-mpi.org.

[20] OpenPBS. http://www.openpbs.org.

[21] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, 1999.

[22] C. Pinchak. Placeholder scheduling for overlay metacomputing. Master's thesis, University of Alberta, 2003.

[23] L. Pireddu, D. Szafron, P. Lu, and R. Greiner. The path-a metabolic pathway prediction web server. *Nucleic Acids Research*, 34:W714–W719, 2006.

[24] S. Eddy *et al.* HMMer: Biosequence Analysis Using Profile Hidden Markov Models. http://hmmer.janelia.org/.

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

[26] Y. Wang. *Transparent Dataflow Detection and Use in Workflow Scheduling: Concurrency and Deadlock Avoidance*. PhD thesis, University of Alberta, 2008.

[27] D. S. Wishart, D. Arndt, M. V. Berjanskii, P. Tang, J. Zhou, and G. Lin. Cs23d: a web server for rapid protein structure generation using nmr chemical shifts and sequence data. *Nucleic Acids Research*, 36(Web-Server-Issue):496–502, 2008.

[28] T. Ylonen. SSH - Secure login connections over the internet. *Proceedings of the 6th Security Symposium*, 1996.

[29] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th conference on Symposium on Operating Systems Design & Implementation*, 2008.

Appendix A

Code Listings

A.1 Workflow Script for GAFolder Local and Global Workflow

```
#!/usr/bin/env python
# Jole Libraries
from job_manager import *
from job_future import *
from job_array import *
from data_handle import *
from monitor import *

from gafolder_lib import *

import operator, threading, os, re, sys
from uuid import uuid4
import time

def main():
  jm = JobManager()
  # Head nodes of two test clusters
  jm.new_submitter('PBS', 'botha-c12')
  jm.new_submitter('PBS', 'botha-c10')

  sthread = threading.Thread(target=score_thread)
  sthread.setDaemon(True)
  sthread.start()

  # Generate uuid for job
```

```
uuid = str(uuid4())
# GAFolder parameters
num_instances = 20
num_carryover = 2
loop_diff = 0.02

loop_count = 0
count = 0

gafolder_dir =\
  "/usr/botha10b/jordan/job_demo/gafolder_dyn_infra/gafolder"

inst_dirs = []
ja = jm.new_job_array()
for inst_num in range(0, num_instances):
  # Uniquely identify each instance
  inst_dir = "gafolder-%s-0-%d" % ( uuid, inst_num)
  inst_dirs.append(inst_dir)
  command = ["./gafolder -it 300 -r %d -pdb " % count,
    DataHandle("gafolder/example/ubiquitin/1ubq.pdb", "r"),
    " -cs ",
    DataHandle("gafolder/example/ubiquitin/bmr5387.str", "r"),
    " > gafolder_log"]
  count += 1
  os.mkdir(inst_dir)
  os.chdir(inst_dir)
  cm = CustomMonitor(job_mon, my_globals=globals())
  jf = GAJobFuture(jm, command,
                   gafolder_dir=gafolder_dir, monitors=[cm])
  ja.add_job_future(jf)
  os.chdir("..")

ja.wait_all() # Wait for GAFolder jobs to finish

prev_best_score = None
loop_count += 1
```

```
while True:
  best_dirs_and_scores =\
    best_score_dirs(inst_dirs, num_carryover)
  cur_best_score = best_dirs_and_scores[0][1]
  dirs = [x[0] for x in best_dirs_and_scores]
  best_pdb_files = best_pdbs(dirs)
  if not prev_best_score is None:
    score_diff = prev_best_score - cur_best_score
    # End iterations if improvement in score below threshold
    if prev_best_score - cur_best_score < loop_diff:
      break
  prev_best_score = cur_best_score
  inst_dirs = []

  ja2 = jm.new_job_array()
  for inst_num in range(0, num_instances):
    inst_dir = "gafolder-%s-%d-%d"\
               % ( uuid, loop_count, inst_num)
    inst_dirs.append(inst_dir)
    command = ["./gafolder -it 300 -r %d -pdb " % count,
     DataHandle(best_pdb_files[inst_num%len(best_pdb_files)],\
                "r"),
     " -cs ",
     DataHandle("gafolder/example/ubiquitin/bmr5387.str", "r"),
     " > gafolder_log"]
    count += 1
    os.mkdir(inst_dir)
    os.chdir(inst_dir)
    cm = CustomMonitor(job_mon, my_globals=globals())
    jf = GAJobFuture(jm, command,
                     gafolder_dir=gafolder_dir, monitors=[cm])
    ja.add_job_future(jf)
    os.chdir("..")

  loop_count += 1
  ja2.wait_all() # Wait for GAFolder jobs to finish
```

```
if __name__ == "__main__":
  main()
```

A.2 Custom Local Monitor Used for GAFolder Instances

```
def job_mon():
  header_hsh = {}

  # Regular expressions for various output
  hl_re = re.compile('^-+\n$')
  output_re = re.compile\
              ('Best score after initializing all genomes')
  output_re2 = re.compile('To set version\=124')
  output_end_re = re.compile\
                  ('To set version\=123') # Output is done

  # Track where we are in file
  begin = False
  begin2 = False
  first_hl = True
  nlh = False

  scores = []
  # Iterate through log file as it is added to
  for line in tail_f_retry("gafolder_log"):
    # Break once output is done
    if begin2 and output_end_re.match(line):
      break
    # Check score of current output
    if begin2:
      sp_line = line.split()
      try:
        iter = int(sp_line[0])
        score = float(sp_line[header_hsh['Total']])
        scores.append(score)
        # Check standard deviation of last 25 scores
        if std_dev_last(scores, 25, 0.0001):
```

```
      # Notify placeholder job is done
      self.comp_state.change_state("Log Finish", True)
      break
  except Exception as e:
    pass
# Split header to get column names
if nlh:
  sp_header = line.split()
  for x in range(0,len(sp_header)):
    header_hsh[sp_header[x]] = x+1
  nlh = False
# Find first line previous to header
if first_hl and hl_re.match(line):
  nlh = True
  first_hl = False
# First beginning regex
if output_re.match(line):
  begin = True
# Second beginning regex
if begin and output_re2.match(line):
  begin2 = True
```