Task Pipeline Specification and Scheduling

John Schulman and Arjun Singh

Abstract—Research pipelines, such as those often found in computer vision or computational biology, often consist of a large number of heterogeneous programs. This leads to brittle code that is difficult to maintain, while requiring significant effort to parallelize across multiple machines. We present a lightweight framework for specifying the execution of such pipelines. The structure imposed by our framework allows us to optimize the execution of the pipeline. Specifically, we know what resources (i.e. files) each task in the pipeline depends on and produces. We can use this information to intelligently assign jobs to different machines in the cluster. Our framework executes pipelines on clusters of machines with minimal effort (and automatic dependency packaging) while scheduling tasks to minimize time until completion (including file transfer time). We compare different heuristics used to initialize our algorithm, and show that the scheduling algorithm used can yield a significant impact on performance. We also give results on running a real-world pipeline with a single machine and multiple machines. Lastly, we compare the amount of code required to implement similar pipelines using different frameworks.

I. INTRODUCTION

Modern scientific computing pipelines often contain tens to hundreds of steps and operate on terabytes of data. The researches specifying these pipelines are often not systems engineers, but rather computational biologists, computer vision researchers, or other non-systems domain experts. In some fields, they often aren’t even trained software engineers, only learning enough programming skills to get the job done. This leads to several issues:

1) the code produced to specify these pipelines is often brittle and convoluted (i.e. spaghetti code)
2) the researchers are not easily able to parallelize their work
3) the amount of time taken to modify and run experiments goes up solely due to implementation details

To address these problems, many research groups have team members whose roles are primarily to implement the algorithms and methods designed by the researchers and for institutional clusters or grids, using tools such as MPI. These efforts make sense when the code is stable, but during the research and development process, this requirement slows things down. However, with the advent of easily-available general cloud computing in the last several years, software should be able to eliminate the need for extra development in many cases.
One other complication that often arises is that these pipelines often have a significant number of dependencies, making it difficult to run on another researcher’s computer or in the cloud. Furthermore, pipelines are often heterogeneous – they use multiple programming languages and environments, and rely on third-party libraries and binaries that the researchers have no control over. This makes it yet more difficult to easily run such code on other machines (e.g. EC2 machines).

We have three contributions:
1) a lightweight framework for specifying task pipelines
2) a framework for automatically packaging up all dependencies for a pipeline using CDE [2]
3) a scheduling framework for optimizing the execution a dependency graph of tasks in parallel

We first discuss related work, both on task pipelines and scheduling, in Section II. We then give an overview of all components in our system in Section III. We discuss the implementation of the framework in Section IV, and then discuss the problem and implementation of scheduling in Section V. Lastly, we give results and compare to other framework in Section VI and give some concluding thoughts in Section VII.

II. RELATED WORK

A. Pipeline Software

Recently, several software packages have been created for the purpose of specifying dependency graphs of computational tasks and executing them in parallel. Ruffus is a Python library for specifying a computational pipeline [3]. It uses decorators to specify dependencies between these tasks. Ruffus does not handle multi-computer parallelism. One downside of Ruffus is that it yields a specification that is difficult to maintain as it grows; it is also quite difficult to understand the control flow simply by looking at the script, due to the decorator-based dispatch.

Compmake [4] is a new Python-based tool that has make-like capabilities but enables parallel multi-computer execution of code. One limitation of Compmake is that it is Python-only. It looks great for cases in which all code is written in Python, but as soon as a step requires a third-party binary, one would have to reach for another tool to integrate it with Compmake. One interesting feature of Compmake is its integration with SGE and Multyvac. This integration allows a user to easily parallelize their work on one of these platforms. Our framework aims to give users the same ease of use while allowing for heterogeneous tasks.

There are also popular frameworks aimed at making it easy for users to run their code on clusters. Perhaps the best example of this is Hadoop [5], followed by the more recent Spark [6]. Hadoop popularized the map-reduce cluster computing paradigm, while Spark generalized this approach to allow DAG-based workflows. However, both are still essentially homogenous platforms, and one would have to build their code specifically for this platform. Our system allows a user to write their code however they normally would when developing each step in isolation, and then easily hook multiple steps together.

B. Scheduling

Optimal scheduling problems have been considered for a long time in the operations research community. The problem we address here is most similar to job shop scheduling [7], whose formulation is that a set of identical machines must perform a set of tasks as fast as possible. Some fast, special-purpose methods have been developed for this problem. The communication aspect of our planning problem is not considered in job-shop scheduling, and it renders the special-purpose solution methods unusable. We suspect that there is a way to draw on this literature, but were unable to find a formulation that worked in time.

We also drew inspiration from the delay scheduling work used by Spark [8], in which the scheduler attempts to assign jobs close to where their data is. When working with large amounts of data, the communication often becomes as much (or more) of a bottleneck as the computation; considering data transfer can lead to huge efficiency gains.

III. OVERVIEW

We begin by discussing the terminology used throughout the paper. A pipeline is composed of multiple stages, where each stage can consist
of multiple jobs, and each job is defined by a task specification and a list of input and output resources. Every job in a stage uses the same task specification, but different sets of input and output resources. For example, in Figure 1, each box represents a pipeline stage.

A user begins by defining each task that can be part of the pipeline. We currently support two types of tasks: those written in pure Python, and those that run arbitrary binaries (which could be interpreters themselves – it’d work fine to use Python as a binary rather than pure Python). Examples of a pure Python and a binary task specification are shown in Figure 3.

The user then generates each job to be run in the pipeline. Currently, this process is explicit and imperative, though one possible extension is to develop a declarative DSL on top of the framework. An example of jobs being added to a pipeline is shown in Figure 4. This figure also shows how a user can specify parameters that are global across all jobs, as well as job-specific parameters.

The information encoded by these two pieces are enough to enable us to 1) build up the job graph, 2) package up any and all dependencies used by each task, and 3) execute the job graph locally or remotely.

IV. IMPLEMENTATION

We now discuss the implementation of the framework. First, we give a bit more detail on the task specification and packaging. We then discuss the data structures used, and lastly discuss the infrastructure required for executing the pipelines.

A. Task Specification

The task specification relies on having the user provide everything required to run each task, whether it’s the code for a pure Python task or the path and argument structure for a binary task. This is shown in Figure 3. The syntax for this specification is lightweight, and required essentially no extra effort over what would be written for a typical imperative implementation of the task (as a function or a class).

In the case of a binary task, the user specifies a single executable which can rely on an arbitrary
class DetectChessboard(Task):
    input = {'image': 'filename'}
    output = {'board': 'filename'}

def run(self):
    from scipy.misc import imread
    import pycb
    board_size = self.params['board_size']
    img = imread(self.input['image'])
    corners, chessboards = pycb.extract_chessboards(img, use_corner_thresholding=False)
    pycb.save_chessboard(self.output['board'], corners, chessboards, [board_size])

class FilterDepthDiscontinuities(BinaryTask):
    input = {'depth_map': 'filename'}
    output = {'filtered_depth_map': 'filename'}

    executable = os.path.join(PERCEPTION_DIR, "model_building", "util", "depth_discontinuity.py")

def args(self):
    args = "--in_loc={0} --out_loc={1}".format(self.input['depth_map'], self.output['filtered_depth_map'])
    return args

Fig. 3. An example of a task specification for both a pure Python task and a binary-based task. All code that is run for a pure Python task is executed in the 'run' method. A binary task simply specifies the executable and the required arguments to the executable based on the input and parameters given to the task specification. The input and output resources are accessed through self.input and self.output, while any parameters available to the job are available through self.params.

number of external dependencies (e.g. configuration files, libraries, parameters in files). We use CDE [2] to generate self-contained packages that can be executed on most Linux machines as long as the underlying CPUs support the instruction set the executable was compiled for. At a high level, CDE works by using strace to determine all files touched during an execution of the binary, and then gathering them up.

B. Pipeline Structure

We use the specification provided by the user to generate two DAGs. The first includes both resources (i.e. files) and the jobs that depend on and produce them. An example of this structure is shown in Figure 2. Resources with no parents are assumed to already exist, and resources with no children are assumed to the final output of the pipeline.

We then also condense this graph into one that only has the job dependency relationships. An example is shown in Figure 5. We do this by simply taking all grandchildren of each job node in the job + resource graph. This allows us to easily see what jobs are prerequisites for other jobs.

C. Pipeline Execution

We use a master/worker configuration, in which we assume the user’s computer has the input data and acts as the master node. We do not handle failed nodes at this time. The master has the pipeline data structures and runs a Tornado server that responds with jobs for workers.
pipeline = Pipeline()
n pipeline.set_root("~/data")

board_size = {'y': 9, 'x': 8}
reference_camera = "NP5"
turntable_square_size = 0.018923

pipeline.addGlobals(
    
    "board.size": board.size,
    "reference.camera": reference.camera,
    "turntable.square.size": turntable_square_size,
)

for object in objects:
    base_dir = os.path.join(pipeline.root_dir, dataset, object)
    image_filenames = glob.glob(os.path.join(base_dir, reference.camera + "_.jpg"))

for image_filename in image_filenames:
    output_filename = os.path.splitext(image_filename)[0] + "_corners.txt"
    input = {'image': image_filename}
    output = {'board': output_filename}
    pipeline.addTask(DetectChessboard, input, output, job_specific=42)

pipeline.run()  # Runs the pipeline locally

Fig. 4. An example of specifying global variables and adding tasks to the pipeline to be run. The addTask method also allows arbitrary keyword arguments; these get turned into parameters accessible by the tasks. The pipeline is then executed locally.

On each worker node (the master can also be a worker), we run a process that makes HTTP requests to fetch jobs and POST results. We use the cloudpickle library [9] to pickle nearly any Python object. In this case, we pickle the Task classes and instances (where each instance is a job to be run). The data that each worker receives is then simply a pickled task instance, and the data that the worker returns to the master is a flag indicating success or failure, and a pickled exception in the case that the task failed to run.

Regardless of whether the pipeline is executed locally or remotely, the overall architecture is the same. The only difference is that if the pipeline is run remotely, we have worker processes on remote machines fetching jobs. If a resource doesn’t exist on the worker node, the worker first fetches the resource prior to execution. The worker is able to assume that the resource exists somewhere (on the master or another worker), because the scheduler never dispatches jobs that don’t have all of their prerequisites met.

The simplest benefit that arises from our framework is that the execution of one pipeline stage doesn’t have to finish before another stage can start running. In a naive, ad-hoc implementation (such as the one we wrote ourselves prior to writing this framework), there is no structure that allows us to start executing jobs from the next stage before the previous one finishes. As shown in Section VI, this has a significant impact on running time.

V. SCHEDULING

A. Problem Specification

Given a set of tasks to perform, and a set of available computers, we would like to devise a plan of how to distribute the computation so as to minimize the time to completion. The communication constraints present in this problem make it different from other scheduling problems that have been considered. The computational operations produce large data files, so it is prudent to avoid transferring these files when possible because of the transfer time. In particular, there are three
limited resources of each computer: computation rate (i.e., number of CPUs), download bandwidth, and upload bandwidth. A given computer can begin downloading a file that it needs for a future task while it is performing a previous task. In this section we formally define the scheduling problem and then propose an algorithm based on simulated annealing for optimizing the allocation of tasks to computers.

The inputs to the algorithm are the following:
- Dependency graph, with a set of jobs, each of which has a set of inputs and outputs (inputs and outputs are typically files)
- Number of machines available for computation
- Specification of how long each computation and transport takes

The output is a set of tuples specifying computation and transport events.
- Computation event: (job, location, start time, finish time)
- Transport event: (from loc., to loc., start time, finish time)

Tuples obey the following constraints:
- A resource is only used after it arrives at a location
- Computation and bandwidth limits are respected by these intervals (k concurrent jobs per computer, only one upload and download at a time 1)
- Final resources arrive at the master computer.

B. Overview of Scheduling Algorithm

Our key simplifying assumption is as follows: given an assignment of jobs to computers, a near-optimal plan can be computed using a greedy controller. The greedy controller is defined to be a simple scheme for generating full plan (timing and ordering of events) given an assignment of jobs to computers. This controller can be straightforwardly be executed in real time, providing workers with jobs in response to completion messages.

Given this assumption, the planning problem is reduced to the problem of assigning jobs to computers. The timing and ordering of these jobs will be determined by the greedy controller. As we will describe below, the planning algorithm we use is hill-climbing search on the assignment vector.

The greedy controller is defined as follows. Whenever a job or transport event finishes, check to see if any new computation or transport events can be started. Computation: If all of the prerequisites for a job are present on a computer, start that job. Transport: If a future job requires a resource but that resource is created on a different computer, AND the source and target computer are free to send and receive, respectively, start transporting that resource. Can perform this check efficiently by only looking at frontier of computation graph.

C. Hill-Climbing Algorithm

Hill-climbing (along with simulated annealing and related methods) is often used in planning problems that don’t have the right structure to enable efficient exact solution methods [10]. The below we how we hill-climbing search in the task we consider—scheduling computational tasks across a network.

Let \( a \) denote an assignment vector of length \( N_{\text{jobs}} \), \( a_i \in \{1, 2, ..., N_{\text{computers}}\} \). Let \( \text{TotalTaskTime}(a) \) be the total task completion time that computed by simulating the greedy controller. The algorithm is as follows:

\[
\begin{align*}
& a_{\text{best}} \leftarrow [1, 1, \ldots] \\
& t_{\text{best}} = \text{TotalTaskTime}(a_{\text{best}}). \\
& \text{for } i_{\text{trial}} \leftarrow 1, 2, \ldots, N_{\text{trials}} \text{ do} \\
& \quad a_{\text{trial}} = \text{copy}(a_{\text{best}}) \\
& \quad \text{Pick random element of } a_{\text{trial}}, \text{ and set it to a random value.} \\
& \quad \text{Simulate executing greedy policy on } a_{\text{trial}}, \text{ and compute total task time } t_{\text{trial}}. \\
& \quad \text{if } t_{\text{trial}} < t_{\text{best}} \text{ then update } t_{\text{best}} \leftarrow t_{\text{trial}}, a_{\text{best}} \leftarrow a_{\text{trial}} \\
& \text{end if} \\
& \text{end for}
\end{align*}
\]

D. Hill-Climbing Initialization Heuristics

To initialize the Hill-Climbing procedure, we looked at two heuristics at different extremes of how much they favored data transfer.

The most naive heuristic simply assumes that data transfer has no cost, and greedily assigns jobs workers as they request them, ignoring the location of data.

---

1This constraint can be relaxed at execution time, but makes planning and simulation simpler
VI. RESULTS

A. Scheduling Results

Unfortunately, the scheduling algorithm we used ended up not having a big impact. However, the results show that scheduling can clearly play a big role...
role in determining the overall running time of the pipeline.

In Figure 6, we show the scaling behavior of our algorithm when initialized with both the greedy approach and the heuristic that minimizes data transfer. For the problem examined here, the graph had five connected components. When the number of machines is small in comparison to the number of connected components, the approach that tries to minimize data transfer is nearly twice as efficient as the approach that naively assigns jobs. However, as the number of machines scales, this approach underutilizes machines, and the naive approach wins out, as it trivially approaches full utilization.

In Figure 7, we examine the performance of the hill-climbing algorithm. We see that the hill-climbing only makes a difference in easy situations, when the number of computers is low and the cost is still high.

These results suggest that there is still significant room for improvement in the scheduling aspect.

B. Execution Results

We ran a real-world pipeline based on a pipeline from our own research. The goal of this pipeline is to take 1200 images and 600 point clouds and generate high-quality 3D models of objects. This process is run on a thousand objects, though we only consider 5 objects here. We ran this pipeline on a variety of scenarios, as shown in Table I. Clearly, the framework is able to significantly speed up execution of a real-world pipeline by
from ruffus import *

@transform(starting_files,
suffix(“.fasta”),
”.sam”)  
def map_dna_sequence(input_file,
output_file):
    ii = open(input_file)
    oo = open(output_file, "w")

@transform(map_dna_sequence,
suffix(“.sam”),
”.bam”)  
def compress_sam_file(input_file,
output_file):
    ii = open(input_file)
    oo = open(output_file, "w")

pipeline_run()

# Input = starting files  
# suffix = .fasta  
# Output suffix = .sam

# Input = previous stage  
# suffix = .sam  
# Output suffix = .bam

Fig. 8. An example of a pipeline specification in Ruffus. Although a simple pipeline is straightforward, the decorator-based dispatch gets messy for larger pipelines.

adding more machines to the cluster; however, it also seems to be the case that the naive scheduling is quite inefficient.

Another interesting result is that even with no paralellization, the pipeline runs significantly faster than our old, ad-hoc implementation of the pipeline, which would have to wait for each stage to finish prior to executing the next stage. This is because the old implementation did not have the structure that enabled it to run the next set of jobs before the first set finished, while a DAG encodes this information.

The lack of linear scaling is likely due to the communication overhead. Better scheduling would alleviate much of this. It would also be interesting to investigate using a distributed file system to possibly speed up access to the resources.

C. Pipeline Specification Results

We compared the old implementation of the pipeline with our new implementation, and found that we were able to cut down the lines of code by about 1/6.

We looked at implementing the pipeline in other frameworks, such as Oozie (based on Hadoop), but found it difficult to even specify the entire pipeline in a reasonable amount of time (we spent roughly 2 hours reading Oozie documentation trying to get everything in place to be able to run it, but then ran into trouble).

An example of a Ruffus-based pipeline is shown in Figure 8. The primary downside of specifying a pipeline in Ruffus is that the control flow is specified completely through decorators, so it’s hard to figure out how a large pipeline works when coming back to it after a few days. Furthermore, the implementation doesn’t currently let you parallelize across machines, though the same techniques we use could be added to Ruffus easily enough.

VII. CONCLUSIONS AND FUTURE WORK

We thought the project was overall successful, though we’d have liked to see the scheduling work out better. We tried other approaches for scheduling, based on graph search and integer linear programming, but couldn’t quite find the

<table>
<thead>
<tr>
<th>Version</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old pipeline</td>
<td>642</td>
</tr>
<tr>
<td>New pipeline</td>
<td>545</td>
</tr>
</tbody>
</table>

TABLE II

LINES OF CODE IN TWO IMPLEMENTATIONS OF THE SAME PIPELINE.
right approach in time. However, by all metrics, we have already improved on what was already easily available to us (namely Ruffus and our own code). We can now trivially run our pipeline on a cluster of machines, with a smaller amount of clearer code. It’s also nice to see that there’s significant room for further speedups through more intelligent scheduling.

We plan to continue using the framework we built, and to improve it further. Possible areas of further work include:

1) Visualizing the state of the running pipeline to understand how much is complete and pending
2) More intelligent scheduling, based off of existing job-shop scheduling algorithms
3) Dynamic rescheduling based on the completion of each job to account for slow workers
4) Handling failed worker nodes by reallocating and rerunning jobs where the resources have been lost
5) Integrate with a distributed file system
6) Only rerun parts of the pipeline that have had their code or input changed
7) Automatically manage the execution of the pipeline with different parameters over the same data while minimizing data transfer (common use case for machine learning and vision)

ACKNOWLEDGEMENTS

We thank Ben Kehoe, Jie Tang, Ziang Xie, Karthik Narayan, and James Sha for their feedback that helped guide the goals of this project.

REFERENCES