HASTE: Serverless DAG Execution Optimizer

Avinash Arjavalingam
Aditya Parameswaran, Ed.

August 12, 2021

HASTE: Serverless DAG Execution Optimizer

by Avinash Arjavalingam

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

[illegible]

Professor Aditya Parameswaran
Research Advisor

8/10/2021

(Date)

* * * * * * *

[illegible]

Professor Joseph E. Gonzalez
Second Reader

8/ 10/ 2021

(Date)

HASTE: Serverless DAG Execution Optimizer

by

Avinash Arjavalingam

A thesis submitted in partial satisfaction of the

requirements for the degree of

Masters of Science

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Aditya Parameswaran, Chair
Professor Joseph Gonzalez

Spring 2021

HASTE: Serverless DAG Execution Optimizer

Abstract

HASTE: Serverless DAG Execution Optimizer

by

Avinash Arjavalingam

Masters of Science in Computer Science

University of California, Berkeley

Professor Aditya Parameswaran, Chair

Before the broad availability of cloud computing, algorithms were developed to generate low latency schedules for processors given an input DAG of computational tasks. These algorithms often relied on a number of simplifications, including assuming an unlimited number of processors, as well as an unlimited amount of memory per processor. With the recent explosion of cloud computing, the setting of unbounded processors is now practically viable. Our project, the HASTE execution optimizer, takes as input workloads consisting of sets of task DAGs, a style of workload that is fairly common with data-driven applications, especially machine learning. HASTE employs the Lower Bound (LWB) scheduling algorithm, a task DAG scheduling algorithm that assumes an unbounded number of processors to create schedules for DAG execution that run each task node in the DAG at its earliest possible time. HASTE utilizes the merging of common prefixes between DAGs to create a single merged DAG, eliminating duplicated execution of equivalent nodes. After passing this merged DAG through LWB to generate a schedule, HASTE determines the memory costs of the schedule, and groups scheduled task executions into VMs in order to minimize total compute time, aggregated across allocated VMs. This report outlines the ILP that groups these components together to minimize the aggregated compute time, and therefore financial cost, of HASTE. Using public Azure Function invocation data, this report also shows how the heuristics employed by HASTE deliver the low latency of LWB while staying within reasonable financial cost relative to standard DAG execution algorithms.

Contents

List of Figures

Acknowledgments

Firstly, I would like to thank my advisor, Professor Aditya Parameswaran. His support and wisdom has been extraordinarily helpful during my time as a Master's student, and his guidance with my research has been invaluable. I would also like to thank Professor Joseph Hellerstein for welcoming me into his group as an undergraduate research assistant where I learnt how to do computer science research. I am also grateful for his help with my application to the Master's program. I would like to thank Professor Joseph Gonzalez for helping me with my Master's report and for agreeing to be the second reader. I am especially grateful to Doris Xin, who introduced me to her body of research and guided me through my time carving out my own research in a similar space. I've been blessed to be able to collaborate with many great people during my time at Berkeley, including Vikram Skreekanti, Chenggang Wu, Aditya Ramkumar, Pranav Gaddamadugu, and Nathan Pemberton. Lastly, I'd like to thank my parents and sister. They have provided me will all the comfort and support needed to get through my undergraduate and graduate studies.

Chapter 1

Introduction

Data processing, especially for machine learning, is becoming increasingly popular across a wide variety of domains and industries. Anecdotally, many companies in the past few years have begun to dramatically increase their use of data analysis pipelines in order to discover actionable insights or build models from the data they produce. Data analysts or scientists are given access to commercial data that is often large in volume and complex. To gain insights or build models from the data, they apply data cleaning and pre-processing steps, often in cells in an Jupyter notebook [11]. They then use this cleaned data to train machine learning models, often by trying, in parallel, many different hyperparameter settings for the same model, as well as different machine learning model classes, which may provide different accuracy levels and behaviors. While not explicitly stated, the data analyst is constructing a directed acyclic graph of computational tasks, henceforth referred to as a DAG. The initial pre-processing cells in the notebook are the initial chain of nodes, with the training and validation of various model types and configurations representing branches of the DAG.

Many modern data-driven problems can be represented as the execution of a set of DAGs. This includes complex tasks, such as automated hyperparameter tuning [18], prediction serving [22], and neural architecture search [23, 15], as well as the most basic of data manipulation tasks, such as preprocessing data and training a simple machine learning model in a notebook as described earlier. While the latter example can usually be run quickly on a personal computer, the former examples are often run at a scale where they need large quantities of resources to run in a reasonable time frame. As a real-world example, a study of 3000 ML pipelines at Google by Xin et al. [25] uncovered pipelines with as many as 6900 nodes. Of these machine learning pipelines, over 80% contained data pre-processing nodes, and over 50% contained data and/or model analysis nodes. Production machine learning jobs of this scale and complexity are becoming increasing common, as evidenced by analysis by companies on their methods for handling their large data analysis pipelines [13, 3], as well as by the proliferation of end-to-end ML systems that handle the execution and deployment of said pipelines in the cloud [7, 9, 5].

A naive execution of a set of DAGs would sort the nodes of each DAG in topological order and then execute the nodes one in the DAG at a time, each DAG at a time, with a single thread of compute. This is the case when executing a group of cells to conduct data analysis in a Jupyter Notebook [11]. While this approach yields correct results, it misses some useful optimizations. Firstly, running the DAGs on a single thread of compute means every node will be run in sequence, a time consuming process that does not utilize the parallelism possible when executing the DAG across multiple processors. Furthermore, there are often many common nodes between the DAGs in such a set. For example, to conduct hyperparameter tuning of an ML pipeline, one must test the ML pipeline with a wide variety of hyperparameter configurations. Many of the steps, such as early data pre-processing and cleaning, will be shared or equivalent between many configurations, representing identical computation. The user will likely waste compute time, and therefore financial cost, executing equivalent nodes multiple times. Research exploring this topic by Li et al. [18] supports the prevalence of this issue. Finally, even if the user executes each DAG individually on a separate cloud instance, they will not effectively utilize the multiple processors available on many cloud instances.

To remedy these issues, we present HASTE, an execution optimizer for workloads containing sets of DAGs running in a serverless compute model. HASTE utilizes the concept of equivalent nodes and the merged DAG [18], discussed in Section 3.2, to eliminate duplicate nodes across DAGs. HASTE then uses the task graph scheduling algorithm Lower Bound (LWB) by Colin and Chretienne [14] to generate a schedule for executing the merged DAG on multiple processors. HASTE then determines the amount of on-machine memory that will be needed to execute the schedule between independent VMs. HASTE lowers the needed on-machine memory by storing intermediate results in remote storage when possible. Finally, with the execution schedules and memory allocation plans, HASTE uses simple heuristics to pack the components of schedule, called *execution sequences*, into multi-vCPU VMs with the goal of minimizing the aggregate compute time used in the processing of the workload across all allocated VMs. This in turn lowers the financial cost to the resource provider while still providing minimal latency when executing the workload.

In Chapter 2, we discuss the algorithms and concepts used by HASTE, such as merged DAGs and LWB, as well as other similar work. Then, in Chapter 3, we state our definitions of the problem and the system model we assume. In Chapter 4, we formalize the ILP associated with this optimization problem, and prove that the problem is bounded by reasonable size constraints, proofs that we use to define and explain the HASTE optimizer in Chapter 5. Finally, in Chapter 6 we show that HASTE both outpaces comparable approaches and uses a reasonable amount of compute resources by testing HASTE on microbenchmarks constructed from Microsoft Azure Functions invocation traces [8].

Chapter 2

Background and Related Work

2.1 DAG Representation and Assumptions

The input to HASTE is a workload $W = (G)$, where $g_t \in G$ is one of the task execution directed acyclic graphs (DAG) that make up the set W. Every node in each DAG represents an atomic computational task to be executed, and every edge in the DAG represents a dependency between tasks. More concretely, the DAG g_t is specified by (N_t, E_t), where node ${n^t}_i \in N_t$ is a computational task in g_t, and edge ${e^t}_{ij} \in E_t$ is an edge from ${n^t}_i$ to ${n^t}_j$ where the output of ${n^t}_i$ is an input of ${n^t}_j$. In the case where we are only considering a single DAG, including the case of applying LWB to the merged DAG of all input DAGS in W, we will omit the t.

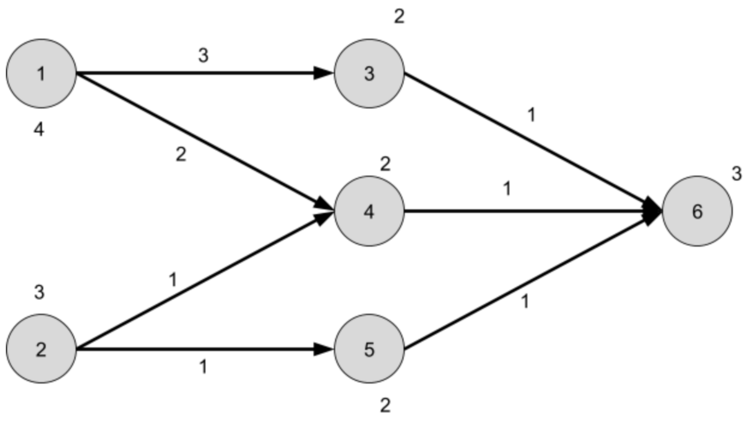

Figure 2.1: Example execution DAG, with the numbers inside nodes being the node ID i, the numbers adjacent to nodes being execution times r_i, and the numbers adjacent to edges being communication time c_{ij}

The node n_i has the properties: $[r_\mathrm{i}, x_\mathrm{i}]$, and the edge e_ij has properties: $[s_\mathrm{ij}, c_\mathrm{ij}]$. The property r_i is the processing time of the task corresponding to node n_i, and x_i is the maximum allocation of memory necessary in bytes during processing. Here, we make the simplifying assumption that these quantities are known to us. The property s_ij is the size in bytes of the output of n_i that is an input of n_j, henceforth referred to as the *message* associated with e_ij. The property c_ij is the communication time of sending e_ij between two VMs. We assume that for all $n_\mathrm{i} \in N$, x_i is less than the amount of main memory for VMs. Because all nodes will require at least enough memory to hold their inputs, we know that for all $n_\mathrm{i} \in N$:

$$x_\mathrm{i} \geq \sum s_\mathrm{ki} \; \forall k : e_\mathrm{ki} \in E$$

An example of such a DAG can be seen in Figure 2.1. The numbers inside the nodes are the Node IDs i. The numbers adjacent to the nodes are the r_i, or task processing time, of each node, and the numbers adjacent to the edges are the c_ij, or communication times, of each edge. We omit n_i and s_i in this diagram.

2.2 Lower Bound Scheduling Algorithm

The Lower Bound algorithm (LWB) is a DAG scheduling algorithm developed by J. Y. Colin and P. Chrétienne [14]. The input to LWB is a task execution DAG as previously defined. LWB assumes infinite memory per processor, and as such, when generating a schedule with LWB, we can ignore the parameters x_i (node maximum memory allocation) and s_ij (edge message memory size). The output of LWB comes in the form of a set of *execution sequences.* Each *execution sequence* consists of a linear sequence of a subset of the input DAG nodes with execution start times assigned to each node. Since each *execution sequence* is to be run on a single processor, none of the execution times of the nodes in any individual *execution sequence* overlap.

LWB is able to generate schedules with a *minimal makespan.* Minimal makespan is the shortest possible execution time of the work, with execution time being measured from the beginning of processing until no more computation is required. Schedules with a minimal makespan are henceforth referred to as *optimal schedules.* To do so, LWB, when creating a schedule, sometimes employs task duplication, which is when multiple copies of the same task are scheduled to be run independently. Also, a restriction on the input DAGs is that the DAG must have small communication times relative to task computation times. More concretely:

$$min(r_\mathrm{a} | n_\mathrm{a} \in parents(n_\mathrm{i})) \geq max(c_\mathrm{ai} | n_\mathrm{a} \in parents(n_\mathrm{i})) \forall n_\mathrm{i} \in N$$

An explanation of why this assumption is necessary can be found in Colin and Chrétienne [14]. LWB consists of two parts: computing the lower bound of the start times of each node, and then building a schedule of *execution sequences* where all copies of nodes are scheduled to start at their lower bound start times. As an example to help follow along, we will use

Figure 2.1 as an example input to LWB. The following explanation is heavily inspired by the explanation of LWB in Colin and Chrétienne [14].

Part 1: Computing the lower bound of the node start times

The first step of LWB takes the input DAG and calculates the lower bound of the start times of each node. The lower bound of the start times is the earliest time that any given node can begin execution. This depends on when all of the nodes that the given node depends on complete. As a shorthand, we will use $PRED(n_i)$ to refer to all of the immediate predecessors of node n_i. To start, LWB takes all of the "root" nodes, which have no predecessors, and assigns them a lower bound start time of 0, as they can all start immediately. Then, LWB enters a loop that exits only when all nodes have been assigned a lower bound start time. At every iteration, this loop selects a node n_i which has not been assigned a lower bound start time but all of whose $PRED(n_i)$ have been assigned lower bound start times. At this step, LWB uses the lower bound start times of these predecessors, as well as the execution times of the predecessor tasks and the communication times of predecessor outputs which input to n_i to determine and assign the lower bound start time of n_i. We first find the predecessor of n_i that will end the latest of all the predecessors, including communication time in our calculations. The LWB algorithm then assumes all copies of n_i will be run on a processor that has run a copy of the aforementioned latest predecessor. With this assumption, the LWB algorithm then determines the earliest time that n_i can be scheduled, given that the latest predecessor will not need to incur communication time, but the other predecessors will. More concretely, the algorithm is as such, with b_i being defined as the lower bound start time for n_i:

```
Calculate lower bound start times for $N$ in $G$
  **for** $n_i \in N$ **do**
    $b_i = -1$
  **end for**
  **for** $n_i \in N$ **do**
    **if** $PRED(n_i) = \emptyset$ **then**
      $b_i = 0$
    **end if**
  **end for**
  **while** $\exists n_i \in N \mid (b_i = -1 \text{ and } (b_k \geq 0 \; \forall n_k \in PRED(n_i)))$ **do**
    $C = max(b_k + r_k + c_{ki} \mid k \in PRED(n_i))$
    Define $s$ such that: $b_s + r_s + c_{si} = C$
    $b_i = max(b_s + r_s, max(b_k + r_k + c_{ki} \mid k \in PRED(n_i) - (s)))$
  **end while**
```

We see in Figure 2.2 the application of this first step to Figure 2.1, where the top row are the Node IDs, and the bottom row is the lower bound start time corresponding to each node.

1	2	3	4	5	6
0	0	4	4	3	7

Figure 2.2: Node IDs (top row) to Lower Bound time (bottom row) for the graph in Figure 2.1

We can see that the "roots" of Figure 2.1 are assigned lower bound start times of 0. As an example, Node 3 is assigned a lower bound start time of 4, because its only dependency is Node 1, which means its lower bound start time is $b_{\mathrm{s}} + r_{\mathrm{s}}$, which is $0 + 4 = 4$. Node 4's latest predecessor is Node 1, whose $b_{\mathrm{k}} + r_{\mathrm{k}} + c_{\mathrm{ki}}$ is $0 + 4 + 2 = 6$. The algorithm assumes all copies of Node 4 will be run on the same processor as a copy of Node 1, meaning we can eliminate the communication time of 2, bringing the time Node 4 needs to wait for Node 1 to complete to 4. The only other predecessor to Node 4 is Node 2, whose $b_{\mathrm{k}} + r_{\mathrm{k}} + c_{\mathrm{ki}}$ is $0 + 3 + 1 = 4$, so the lower bound start time of Node 4 is 4.

Part 2: Building the Schedule of Execution Sequences

Once the lower bound start times are calculated, LWB constructs *execution sequences* to build the minimal makespan schedule of the DAG. To do so, LWB must determine the *critical paths* in the DAG and create an *execution sequence* out of each *critical path.*

A *critical path* is a path in the task execution DAG that contains exclusively *critical edges* and is not a proper subpath of another *critical path.* A *critical edge* is any edge e_{ij} where $b_{\mathrm{i}} + r_{\mathrm{i}} + c_{\mathrm{ij}} > b_{\mathrm{j}}$. That is to say, any edge is critical if the time in which the predecessor node completes, including communication time, is greater than the lower bound start time of the descendant node. All critical edges of the DAG in Figure 2.1 are highlighted red in Figure 2.3. Intuitively, a *critical edge* means that we cannot have communication between VMs for that edge if we are to generate a schedule with minimal makespan. This means all critical edges must occur on the same VM.

Because of how we calculated lower bounds in the first part of the algorithm, at most one predecessor in $PRED(n_{\mathrm{i}})$ can satisfy $b_{\mathrm{k}} + r_{\mathrm{k}} + c_{\mathrm{ki}} > b_{\mathrm{i}}$. Because of this, we know the *critical graph* of *critical path* is a spanning forest of *out-trees*, with out-trees being trees where there is only one path between the root of the tree and any other vertex in the tree. This means each *critical path* corresponds to a single "leaf" node in the task execution DAG. This allows us to convert each *critical path* into an *execution sequence*, each of which is assigned its own processor. A proof that this process finds a minimal makespan schedule can be found in Colin and Chrétienne [14].

We can see in Figure 2.4, each row is a single *execution sequence* corresponding to a *critical path* as shown by the highlighted edges in Figure 2.3. Each box is a copy of a node,

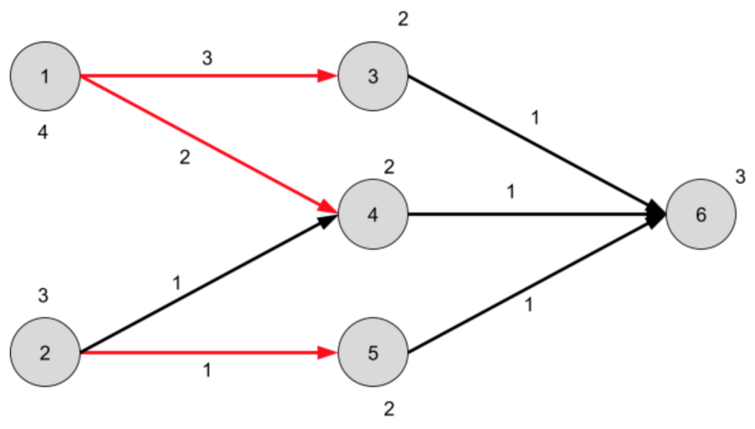

Figure 2.3: The DAG in Figure 2.1 with the critical edges highlighted red

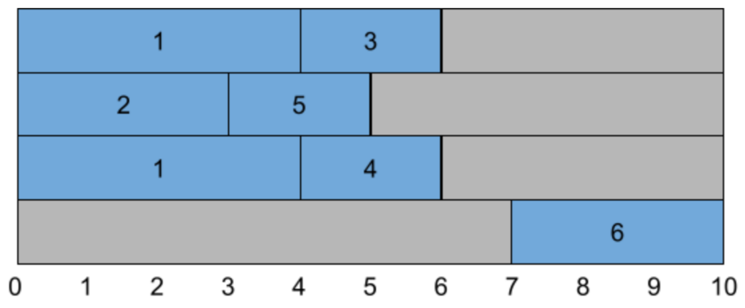

Figure 2.4: Parallel execution plan for the graph, with each row representing an execution sequence

with the number within being the Node ID. We can see that each node is scheduled at its lower bound start time as seen in Figure 2.2.

Related Task Execution DAG Scheduling Algorithms

While we chose the LWB scheduling algorithm, there are numerous DAG scheduling algorithms that have been developed over the years. Yu-Kwong Kwok and Ishfaq Ahmad [17] have compiled a list of many of them, and have formalized categories for the different types of task DAG algorithms based on their properties. This includes algorithms that require a restricted graph structure, ones that assume unit computational costs for nodes, ones that assume no communication costs, and ones that assume an unbounded number of processors. Others still will duplicate task executions in their schedules. All of these algorithms have different time complexity bounds in terms of creating the schedule, as well as disparate performance on varying types of task DAGs.

We chose LWB because its properties fit well with our problem space. LWB can handle DAGs with arbitrary structure and arbitrary computational costs for every node, although it does require perfect knowledge of node processing times in order to generate an optimal schedule. LWB can also handle arbitrary communication costs between nodes on different processors, but assumes 0 communication cost between nodes on the same processor, assumes low communication costs relative to compute costs, and again must have perfect knowledge of all communication costs in order to generate an optimal schedule. All of these assumptions to the problem space are ones we assume for the inputs to HASTE as well. The exact bounds on communication and computational cost are laid out in Section 3.1, and an example of a DAG which meets these requirements is Figure 2.1.

Importantly, LWB allows for task duplication and an unbounded number of available processors, and low communication costs. Task duplication means that two different *execution sequences* in a schedule generated by LWB can be assigned to both run the same node independently. This means nodes with multiple dependencies can be run in parallel on different processors so as to eliminate communication time and reduce overall latency. The assumption of unlimited processors means that LWB does not take as an input the number of processors it must schedule for, and instead can generate as many *execution sequences* as it needs, with each given its own processor. LWB leverages this flexibility in order to achieve optimal lower bound latency for low communication overhead DAGs [14], something most other DAG scheduling algorithms cannot.

2.3 Merged DAG

HASTE takes its definition of *equivalent nodes* and *merged DAGs* from work by Liam Li et al. [18] on optimizing the execution of hyperparameter tuning. Briefly, *equivalent nodes* are task execution nodes in different DAGs that perform the same operation on the same set of inputs, and a *merged DAG* is created by combining these different DAGs into a single DAG by assigning a common sink to the DAGs, and then eliminating duplicate *equivalent nodes*, sending the output of one copy to all the nodes that have it as a dependency. An example merging of DAGs can be found in Section 3.1. As explored in Liam Li et al. [18], hyperparamenter tuning of a single model tends to use similar preprocessing steps between different tunings, and as such there are many opportunities to merge *equivalent nodes* to reduce compute. HASTE exists in the same vein of work based around minimizing the compute necessary to complete a workload by reusing previously computed results. Helix, by Xin et al. similarly leverages reuse by storing the intermediate results of DAG executions across iterations in order to minimize execution time of iteratively chanaged DAGs [24]. HASTE, in contrast, utilizes reuse for a single execution of a workload of a general set of DAGs.

As we will show in Section 4, merging a set of DAGs and creating a schedule via LWB will yield the same latency for every node as creating a schedule via LWB for each DAG individually. This allows us to eliminate the duplicate computation between the schedules of different DAGs that might be present with the latter scheduling strategy. As a consequence, HASTE can potentially save financial cost by using less compute time than applying LWB to individual DAGs, while also not sacrificing the optimal latency of LWB.

2.4 Serverless

As discussed earlier, LWB and many other task execution DAG scheduling algorithms rely on the assuming of an unbounded number of available processor. Decades ago, at the time LWB and other similar algorithms were developed, available compute resources could not possibly fulfill this assumption. However, with the relatively modern computing model of *serverless*, we can approximate this assumption, allowing us to use LWB and similar algorithms in practice. Serverless computing is a cloud computing model that pushes the burden of allocating compute and storage resources from the application developer (as is the case with manually allocated VMs with offerings such as EC2 [21]) to the cloud provider. Serverless function-as-a-service (FaaS) offerings such as AWS Lambda and Cloudburst [22] fulfill this role by manually allocating and de-allocating containers when functions are invoked to match load. However, serverless is not limited to the FaaS model. Our system model is serverless in that it supports any workload that does not include specifications on how to allocate compute / storage resources, and instead leaves it up to the HASTE system.

Since many serverless offerings are Function-as-a-Service systems, much effort has gone into optimizing DAG executions for serverless settings. Systems such as Wukong by Carver et al. [2] and CloudFlow by Subbaraj et al. [22] focus on optimizing single DAG execution via techniques such as data locality, operator fusion, and batching inputs. Numpywren, by Shankar et al. [12], focuses on the use case of linear algebra, and constructs a DSL to optimize matrix operations by maximizing locality. HASTE takes a slightly different path from these systems by targeting the use case of sets of DAGs with equivalent nodes between them as supposed to single DAGs. Furthermore, HASTE takes schedules generated by LWB and places them on VMs (and remote storage when possible), using heuristics to minimize the financial cost of running the workload, which is different from the latency minimizing strategies of most other serverless DAG execution systems.

For the purposes of this report, we assume the HASTE execution optimizer is used by the cloud provider and not the user running the workload. We assume theoretically infinite available compute and remote storage resources that can be brought up and shut down at any time. In terms of the calculated financial cost of a workload, we assume the spin-up and shutdown times of resources cost as much as uptime, as those clusters cannot be used elsewhere. Also, we assume the financial cost per unit time includes when a machine is idling or processing I/O and not running programs. We do not tackle questions such as periodic invocations of serverless functions to keep containers warm. If adopted in practice, we see HASTE as an offering cloud providers give to users wishing to run workloads, similar to AWS Sagemaker [7], where the cloud provider applies HASTE to the workload, allocates and runs the workload based on the output of HASTE, and charges the user for the resources that HASTE calculates to be necessary. This procedure of letting the compute provider handle resource allocation is how HASTE fits into the serverless model.

Chapter 3

Problem Definition

In this Chapter, we define what constitutes an equivalent node between DAGs in a workload. We also state our representations of the output of LWB as well as associated concepts that are used in HASTE. Finally, we define the system model we assume.

3.1 Equivalent Nodes and Merging

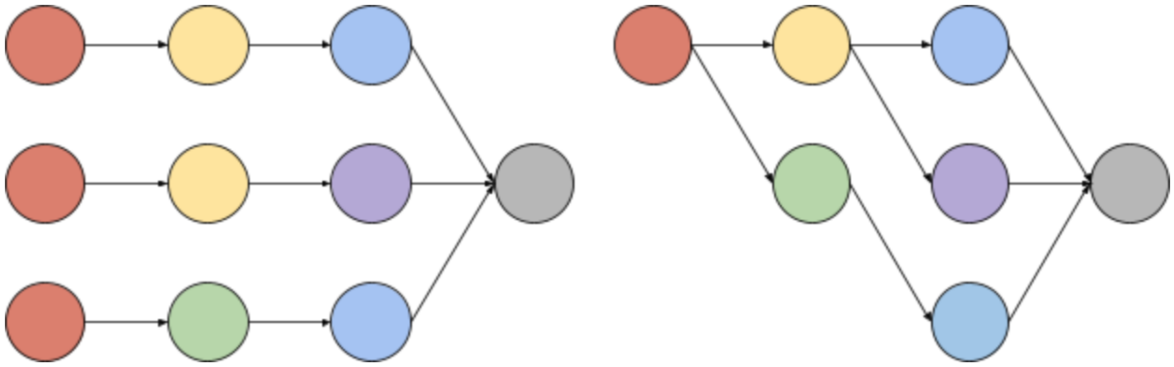

Figure 3.1: Example merged DAG from a set of DAGs

In workload W, the nodes $n^{\mathrm{t}}{}_{\mathrm{i}} \in N_{\mathrm{t}}$ and $n^{\mathrm{u}}{}_{\mathrm{j}} \in N_{\mathrm{u}}$ $(u \neq t)$ are equivalent, denoted as $n^{\mathrm{t}}{}_{\mathrm{i}} \equiv n^{\mathrm{u}}{}_{\mathrm{j}}$ if the operators corresponding to $n^{\mathrm{t}}{}_{\mathrm{i}}$ and $n^{\mathrm{u}}{}_{\mathrm{j}}$ compute identical results on the same inputs, and all the immediate predecessors of $n^{\mathrm{t}}{}_{\mathrm{i}}$ are equivalent to all the immediate predecessors of $n^{\mathrm{u}}{}_{\mathrm{j}}$. That is, there is a 1-1 equivalence mapping between the predecessors of $n^{\mathrm{t}}{}_{\mathrm{i}}$ and $n^{\mathrm{u}}{}_{\mathrm{j}}$.

Figure 3.1 below illustrates how equivalent nodes and merging work in practice, with a few linear DAGs on the left side, and a merged DAG on the right. The color of the node corresponds to the operation performed at that node, but is unrelated to the exact inputs to that node. For both the unmerged and merged DAGs, the grey node performs no operation, as it acts as a sink for all outputs. All red nodes are equivalent among the set of DAGs, as they perform the same operation and have no input. Similarly, the yellow nodes are both equivalent, as they all perform the same operation and take the same input, the red nodes. If these DAGs are part of the same HASTE workload, the red and yellow nodes are merged. However, the blue nodes are not equivalent, since even though they perform the same operation, one takes a red-yellow subtree as an input, and one takes a red-green subtree as an input. Since the input subtrees are different, it is not guaranteed that applying the blue operation on these two inputs yields the same result for both, and it is in fact likely they give different results. As such, the blues nodes in Figure 3.1 are not equivalent, and we cannot combine them into a single blue node in the merged DAG.

3.2 LWB Scheduling Algorithm Output

The application of LWB on a task dependence DAG will then output a set, Y, of *execution sequences*, where $y_{\text{i}} \in Y$ is represented as a linked list. Each element in the linked list contains the node n_{i} and its associated properties, as well as all its incoming and outgoing edges and their associated properties. Every edge is associated with the additional information of which *execution sequence* $y_{\text{j}} \in Y$ that the output of n_{i} will be sent. For some edges this will be the same *execution sequence* as n_{i}, for others it will be a different *execution sequence*, and for others still it will be no *execution sequence*, as with LWB's task duplication, another instance of n_{i} may satisfy that dependency. Each execution sequence also contains a start time and end time of execution for the vertex, as well as a pointer to the next element.

Each *execution sequence* $y_{\text{i}} \in Y$ corresponds to exactly one *memory sequence* r_{i}. A *memory sequence* r_{i} represents the memory requirements for the execution of y_{i}. The *memory sequences* are also represented as linked lists, with each node having an associated start time, as well an integer value of the amount of memory needed for that chunk of time. The execution sequence y_{i} and its corresponding memory sequence r_{i} do not necessarily contain the same amount of nodes, as y_{i} needs to account for the communication between *execution sequences* and the memory needed to accommodate that. An in depth explanation of how *memory sequences* are constructed from *execution sequences* can be found in Section 4.2, and an example memory sequence can be seen in Figure 4.1.

3.3 System Model

The atomic unit of compute in our system model is a VM . Every VM V_{i} has the same number of vCPU P, amount of RAM M, cost per cycle D, startup time U, and teardown time W. While nearly all cloud providers allow the allocation of VMs with varying amounts of vCPU, RAM, and other characteristics, for the scope of our project we will only assume the ability to allocate the same single type of VM. Since local reads from RAM are so quick, we will assume they incur a negligible latency, and do not consider them when scheduling. When determining the cost of allocating a VM, we assume startup time and teardown time cost the cloud provider the same as uptime, as the VM is still unable to service other requests during those times. Therefore, the cost, C_{t} to allocate a VM with uptime t is:

$$C_t = D(t + U + W)$$

For our project, time is measured in discrete CPU cycles, and all processors are perfectly synchronized to the same time 0 when execution begins. As such, all lengths of time are natural integers.

Every node n_{i} runs on exactly one vCPU for the processing time r_{i} and is allocated its maximum allocation x_{i} for the time it runs. If a VM has already allocated $M - x_{\mathrm{i}}$ to other processes, n_{i} cannot run until enough memory is available.

For communication edges between VMs, we assume we are not I/O bound, and as such, that all I/O occurs exactly when it is scheduled to occur. However, although we know the communication time for the entire edge e_{ij} is c_{ij}, we don't know exactly when each packet will arrive at the receiving VM. Therefore, we must make some pessimistic assumptions when determining the allocated memory at communicating VMs. The sending VM allocates s_{ij} for c_{ij} for all edges e_{ij}, representing the sending node being required to allocate the necessary memory for e_{ij} for the entire communication time of the edge. This is necessary because the rate at which the data is sent is unknown, only the total communication time is known. The receiving node will have to allocate s_{ij} for the time represented as a_{ij}:

$$a_{\mathrm{ij}} = (b_{\mathrm{j}} + r_{\mathrm{i}}) - b_{\mathrm{i}}$$

In this case, n_{i} is the sending node and n_{j} is the receiving node. This time represents the time between when n_{i} ends and n_{j} begins. This is to ensure whatever order the packets come, and when they come, there will be enough memory allocated to allow them to be written to the receiving VM. Because of the definition of the LWB algorithm we state in Section 2.2, we know that $a_{\mathrm{ij}} \geq c_{\mathrm{ij}}$. This means there is always enough time to communicate an edge across VMs.

When an *execution sequence* contains two nodes n_{i} and n_{j} which have an edge e_{ij}, we will allocate s_{ij} for a_{ij} as the memory for e_{ij} for both nodes. This is because, according to the definition of the LWB algorithm [14], all of the critical paths from which the nodes of the *execution sequences* are assigned comes from a spanning forest of out-graphs of the DAG. In practice, this means that any node n_{i} will only have to send e_{ij} to one copy of n_{j}, and if that copy is on the same *execution sequence*, then we only need to allocate s_{ij} for the time between the end of n_{i} and start of n_{j}. For the sake of constructing the *memory sequence*, when this case occurs, we assume the sending node n_{i} allocates memory for 0 time. If the time between the executions of n_{i} and n_{j} with edge e_{ij} is 0, and they both reside on the same *execution sequence*, we will not have to allocate any memory for e_{ij} for either node, as none is necessary.

Our system model also considers serverless remote storage, which we designate R. R is assumed to have infinite capacity, and is defined by M, the coefficient that multiplies c_{ij} to get the time to read or write e_{ij} to R. Because remote storage is slower to access, the time to communicate an edge to and from R is always larger than the time to communicate an edge between VMs, and therefore $M > 1$. Formally, the time it takes to read or write e_{ij} to R, represented as m_{ij}, is:

$$m_{\mathrm{ij}} = \lceil M * c_{\mathrm{ij}} \rceil$$

If the VM is reading from R, it must keep at least e_{ij} memory allocated for m_{ij} from when the read begins. However, if the VM is writing to R, it only needs to keep the memory allocated for c_{ij}, as it does not need to keep the memory around when R receiving the data and writing to itself. We choose this option whenever cost memory cost at the receiving VM would be lower when using R as supposed to allocating the memory necessary to receive the input directly from another VM, which occurs when $m_{\mathrm{ij}} \leq a_{\mathrm{ij}}$.

This storage abstraction is derived from offerings such as Amazon S3, a highly scalable object-store with near infinite storage potential, but with much higher read latencies by a VM relative to the equivalent read from RAM or communication between VMs. Because most scalable storage offerings such as S3 are so cheap per unit time relative to compute resources, we will assume storing any amount of memory in R incurs negligible financial cost.

Chapter 4

Problem Bounds and Formalization

In this Chapter, we prove that given a workload, the schedule created by LWB is bounded by the size of said workload, even though LWB allows for task duplication and assumes an unbounded number of processors. We also prove that calculating the *memory sequence* of every *execution sequence* occurs in polynomial time. Finally, we lay out the ILP for minimizing the financial cost incurred from the aggregated VM time of executing the schedule.

4.1 LWB bounds

LWB schedules the nodes in a DAG by first determining the earliest time possible it can schedule each node, known as the lower bound of the node. A rundown of the steps of the algorithm can be found in Section 2.2.

We know from Section 2.2 that the maximum number of *execution sequences* generated, and therefore the maximum number of VMs needed, is bounded by the number of nodes in the DAG [14].

Theorem 1: The maximum number of nodes that can be assigned to an *execution sequence* for DAG g is the size of N, or the number of nodes in the DAG.

Proof by Contradiction: Assume there exists an *execution sequence* y_{i} that has L nodes, where L has more nodes than exist in N. Because the *execution sequence* can only contain nodes from the DAG, there must be at least one node n_{i} that occurs more than once in the *execution sequence.* We know that *execution sequences* are assigned their nodes from the *critical graph*, as described in Section 2.2. From Section 2.2, we also know that that the *critical graph* is a spanning forest of out-trees derived from paths in the original DAG. The LWB algorithm states that each *execution sequence* is given exactly one path from the *critical graph*, with each path being given out exactly once. This means the *execution sequence* must have been assigned a single path in the original DAG that contains at least one copy of a node. This path must therefore contain a cycle, breaking the definition of a directed acyclic graph, and is therefore not possible.

Theorem 2: The maximum amount of memory that could be required at any time in any *memory sequence* made from DAG g is less than or equal to the sum of the memory requirements for all edges in g added to the memory for the largest max allocation of all the nodes in g. Mathematically, this would be: $(\sum s_{\mathbf{ij}} \forall e_{\mathbf{ij}} \in E) + (max(x_{\mathbf{i}}) \forall n_{\mathbf{i}} \in N)$.

Proof: We know from our definition of an *execution sequence* that only one node can run at a time. Therefore, in order to handle any node execution, we will need at most enough memory to handle the largest max allocation of all nodes in N. However, communication of edges can occur parallel to execution and other communication of edges. Our procedure of allocating memory for edge communication defined in Section 3.4 dictates that if the two nodes n_{i} and n_{j} of an edge e_{ij} reside on the same *execution sequence*, we will allocate s_{ij} once at the *memory sequence* corresponding to that *execution sequence*. This will be allocated for the duration between their executions. Otherwise, we allocate s_{ij} for a period of time either before n_{j} starts or after n_{i} finishes. This means that, because we only see each node at most once at every *execution sequence*, we will only have to allocate s_{ij} at most once per *memory sequence* as well. Because of this, the maximum memory needed at any time for an *execution sequence* can never exceed the sum of the memory required for all edges in E and the largest max allocation of all nodes in N, which is what Theorem 2 states.

4.2 Memory Sequence Generation Time

The *memory sequence* generation algorithm takes in as input its *execution sequence*. The *execution sequence* input takes the form of a linked list of the chronologically ordered nodes to be executed as well as their execution start and end times, and the output *memory sequence* takes the form of a linked list of chronologically ordered *allocation* vertices. Each *allocation* contains a start time and an integer value of memory necessary at that start time we will label *size*. The start time of the next *allocation* determines the implied end time of the previous *allocation*. More details on the contents of *memory sequences* and *execution sequences* can be found in Section 3.3.

There are three sectionss of memory necessary for a node n_{i} on a *memory sequence*: the max allocation, the inputs and the outputs. As described earlier, the max allocation is the memory needed to run the node. Specifically, node n_{i} needs x_{i} memory during the run time of length r_{i}. The input and output memory is the memory needed to handle the incoming and outgoing communication edges for n_{i} on this particular *memory sequence*. We use the procedure for determining memory requirements of communicating edges explained in Section 3.4.

Because the memory requirements of all sections of the node is fixed for a given period of time we can treat them all identically. To generate the *memory sequence*, we start at the beginning of the *execution sequence*, and iterate through every node. At every node, we determine the memory requirements for the inputs, outputs, and execution of the node as well as the start and end times of these requirements. For each section of every node, we

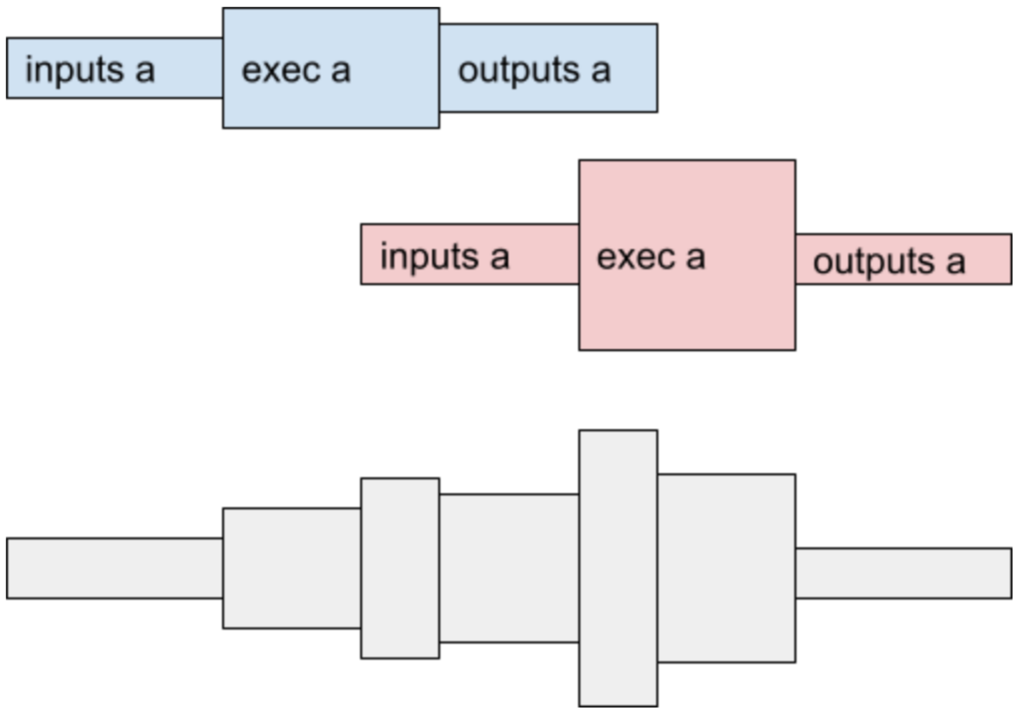

Figure 4.1: Example memory sequence

construct a *component*, which is a data structure composed of the start and end time of that section of the node, as well as the memory requirement needed for that duration. Time is measured in integer cycles, and the memory requirement is measured to the nearest larger integer. We then place an independent copy of each *component* in two min-priority queues, *startpq* and *endpq*. The *startpq* minpq is ordered by the start time of the *component*, and the *endpq* minpq is ordered by the end time of the *component*.

Once we have iterated through all vertices and constructed all *components*, we begin to build the *memory sequence*, starting with an empty linked list. To start, we compare the heads of both priority queues, and pop the head with the lower time (using the start time for *startpq* and end time for *endpq*). We then create an *allocation*, setting its start time to the *component's* time. We then determine the *size* of the previous *allocation*, using 0 if there are no previous *allocations*. If the *component* we just popped comes from *startpq*, we set the *size* of the current *allocation* to: (*size* of previous *allocation* + *component's* required memory). If the *component* we just popped comes from *endpq*, we set the *size* of the current *allocation* to: (*size* of previous *allocation* - *component's* required memory). We then add this *allocation* to the *memory sequence* linked list. We do this process until both min priority queues are empty. Because the start time of each *component* comes before its end time, we will never create an *allocation* will below 0 memory and our *memory sequence* have its last node require 0 memory which corresponds to the last *component* is processed. An example of creating a *memory sequence* can be seen in Figure 4.1.

For each individual *execution sequence* y_{i}, the number of nodes is at most the size of N as per Theorem 1. Because we can't have duplicate nodes, we see the same edge e_{ij} at most twice in y_{i}, once for node n_{i} and once for node n_{j}. Therefore, the sum of the number of edges all nodes in y_{i} have is at most the size of $(2E)$. Running the insertions and removal of all nodes from the priority queues per *execution sequence* will therefore take $O((N+E)log(N+E))$ time complexity. Running the algorithm on all *execution sequences*, of which we know there can be at most the size of N, will have $O(N(N+E)log(N+E))$ time complexity.

4.3 Cost minimization ILP

In this Section, we will define the ILP that minimizes the financial cost to allocate the VMs necessary to run the generated schedule. For the purposes of the ILP, we consider an environment of discrete time measured in cycles.

Inputs

The inputs to the ILP are the memory sequences generated as described in Section 4.2, as well as the potential VMs that they can be run on. The set of memory sequences R contain I memory sequences. Each memory sequence $r_{\mathrm{i}} \in R$ corresponds to exactly one start time for the memory sequence, h_{i}, and exactly one end time for the memory sequence, f_{i}. The memory sequence $r_{\mathrm{i}} \in R$ contains an integer vector of length T, υ_{i}. T is a constant which is the same for every, r_{i}, and equals the maximum completion time among all $r_{\mathrm{i}} \in R$. Each value $x_{\mathrm{it}} \in \upsilon_{\mathrm{i}}$ corresponds to the memory required at time t for υ_{i}. As was proven Theorem 2, the values of x_{it} are in the range $[0, K]$, where $K = (\sum s_{\mathrm{ij}})\forall e_{\mathrm{ij}} \in E + max(x_{\mathrm{i}})\forall n_{\mathrm{i}} \in N$. This vector representation is slightly different than the linked list representation we describe in Section 4.2, but both hold the same information of the memory required for the corresponding *execution sequence*. For an example of how these formats are related to each other, see Figure 4.2.

We use the vectors of the memory sequences in R to create a TxI matrix Ω which contains the vectors in of R, where the i^{th} column in the matrix corresponds to the vector of r_{i}. They take this form:

$$\begin{bmatrix} | & | & & | \\ r_0 & r_1 & \dots & r_{\text{I-1}} \\ | & | & & | \end{bmatrix}$$

As inputs, we also have a set of VMs V, where the size of V is I, and $v_{\mathrm{n}} \in V$ is a single VM. We select I VMs because we will use at most I VMs to run the *executions sequences*. The index of VMs is not related to the index of memory sequences. As laid out in Section

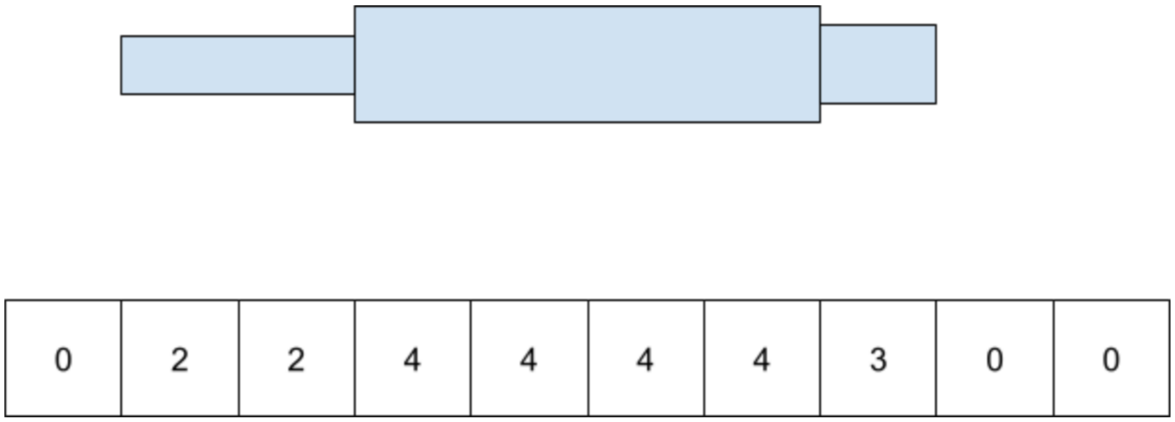

Figure 4.2: A linked list and vector representation of the same memory sequence

3.4, each VM v_n has the same number of vCPU P, amount of RAM M, cost per cycle D, startup time U, and teardown time W.

Setup

In order to assign each r_i to a VM, we will create an $I \times I$ matrix Δ. The nth column vector in Δ, Δ_n, corresponds to v_n. We define Δ as the following:

$$\Delta_{ij} = \begin{cases} 1 & \text{if we assign } \mathrm{r_i} \text{ to } \mathrm{v_j} \\ 0 & \text{otherwise} \end{cases}$$

Each r_n must be assigned to at least one VM to ensure that it is run. Also, each V_j can be assigned no more than P *memory sequences* to ensure there are enough processors to run every assigned *memory sequences*. We will let A be defined as:

$$A = \Omega \cdot \Delta$$

As such, A have dimensions $T \times I$, and A_{ij} represents the memory needed at VM v_i at time step j. A will be used to ensure that no VM is allocated *memory sequences* which it cannot hold in memory.

Once we construct Δ, for each v_n we define a set μ_n. μ_n contains: $[m|(\Delta_{nm} = 1) \in \Delta_n]\forall\Delta_{nm} \in \Delta_m$. In other words, the set contains the indexes of all of the *memory sequences* to be assigned to v_n. Once each μ_n is constructed, it will have a size in the range $[0, I]$. We will denote this size per μ_n as y_n. Once y_n is determined, we will assign an indicator variable, z_n, for each v_n. z_n will have values in the range $[0, 1]$, and will indicate whether or not there are any *memory sequences* assigned to v_n. Specifically, we define z_n to be:

$$z_{\mathrm{n}} = \begin{cases} 1 & \text{if } y_{\mathrm{n}} \geq 1 \\ 0 & \text{if } y_{\mathrm{n}} = 0 \end{cases}$$

Each VM v_{n} is given a VM start time σ_{n} and VM end time ϵ_{n}. σ_{n} must be less than or equal to h_{n} for every r_{n} assigned to v_{n}, and similarly, ϵ_{n} must be greater than or equal to f_{n} for every r_{n} assigned to v_{n}. Both all σ_{n} and ϵ_{n} are within the range $[0, T]$, as no VM can start before time step 0, and no VM will still be executing after all memory sequences are completed at T.

Execution Sequence Placement Problem

Definition 1: Execution Sequence Placement

The goal is to find the execution assignment policy Δ, as well as start times σ_n and end times ϵ_n for every VM v_n that minimize the following expression:

$$\min_{\epsilon,\sigma,z} \sum_{n=1}^{I} (\epsilon_{\mathrm{n}} - \sigma_{\mathrm{n}}) + z_{\mathrm{n}} * (U + W)$$

subject to:

$$\Delta_{\mathrm{ij}} \in \mathbb{Z} \qquad \forall \Delta_{\mathrm{ij}} \in \Delta$$

$$0 \leq \Delta_{\mathrm{ij}} \leq 1 \qquad \forall \Delta_{\mathrm{ij}} \in \Delta$$

$$\sum_{k=0}^{I} \Delta_{\mathrm{kl}} = 1 \qquad \forall k \in [0, I)$$

$$0 \leq y_{\mathrm{n}} \leq P \qquad \forall y_{\mathrm{n}}$$

$$z_{\mathrm{n}} = \begin{cases} 1 & \text{if } y_{\mathrm{n}} \geq 1 \\ 0 & \text{if } y_{\mathrm{n}} = 0 \end{cases}$$

$$A = \Omega \cdot \Delta$$

$$0 \leq A_{\mathrm{ij}} \leq M \qquad \forall A_{\mathrm{ij}} \in A$$

$$\sigma_{\mathrm{n}}, \epsilon_{\mathrm{n}} \in \mathbb{Z} \qquad \forall n \in [0, I)$$

$$0 \leq \sigma_{\mathrm{n}} \leq h_{\mathrm{i}} \qquad \forall i \in \mu_{\mathrm{n}}, \forall n \in [0, I)$$

$$f_{\mathrm{i}} \leq \epsilon_{\mathrm{n}} \leq T \qquad \forall i \in \mu_{\mathrm{n}}, \forall n \in [0, I)$$

In other words, we want to minimize the sum of all the time VMs are allocated, including startup and teardown time if a VM v_{i} is ever assigned a *memory sequence*. This is subject to the constraints that every *memory sequence* has to be assigned to exactly one VM and each VM can hold no more than P *memory sequences*. Also, at no time step should the combined memory needed for all *memory sequences* exceed M, the amount of memory that every VM has. Furthermore, for VMs assigned at least one *memory sequence*, we assign them times it begins executing its assigned *memory sequences*, and a time after that it stops executing its assigned *memory sequences*. That range of time for every VM must be able to fit the start and end times of all *memory sequences* assigned to it.

Chapter 5

HASTE overview

The HASTE execution optimizer takes in as an input workload W, which contains a set of task DAGs to be executed. It outputs a set of tuples I, with tuples $(b_\mathrm{i}, f_\mathrm{i}, Y) \in I$ each representing the responsibilities of a single VM V_i. The variable (b_i represents the time when V_i is instructed to be brought up, and f_i represents the time when V_i is instructed to be torn down. Notably, the time these instructions are sent precedes the completion of the instruction by startup time U and teardown time W, respectively. Y is the set of *execution sequences* y_i that are assigned to run on V_i. The size of Y is within the range $[1, P]$, and each *execution sequence* is assigned to run on exactly one V_i.

The procedure by which HASTE maps inputs to outputs can be divided into four components:

1. Merging the individual DAGs into a *merged DAG* to eliminate equivalent nodes
2. Applying the Lower Bound (LWB) scheduling algorithm to the *merged DAG*, getting the set E of *execution sequences*
3. Using communication and execution times, generating the *memory sequence* r_i for every *execution sequence* y_i
4. Grouping the *execution sequences* together to be executed the groups in parallel on the same VM and determining the uptime of each VM assigned to a group so as to minimize the total compute time / cost for all VMs

In this Section, we describe the components of the HASTE execution optimizer and the reasoning behind the why each component was chosen.

5.1 Merging the DAGs

Using our definition of equivalence in Section 3.2, we can define our DAG merging algorithm. We will use the simple merging algorithm used in Liam Li et al. [18]. We take our set of

DAGs, and iterate through the every pair of nodes, merging two nodes if they are equivalent. Then, we create an edge from the sinks of each of the individual DAGs to a universal sink we create. This sink has no computational time, and simply returns the values of all of its inputs. After merging all equivalent nodes and connecting the sinks, we are left with our merged DAG. This has $O(N^2)$ time complexity, with N being the total number of nodes across all DAGs in W, trivially. Figure 3.1 shows the conversion of a few simple DAGs into a merged DAG.

5.2 Applying LWB

Once the merged DAG is constructed, it can be processed using LWB. LWB will then output the set E of *execution sequences*, the properties of which are described in Section 3.3. The time complexity of this algorithm is $O(N^2)$ [17].

5.3 Creating memory sequences

Once the *execution sequences* are created by LWB, we must determine how much memory each *execution sequence* will need. We use the algorithm described in Section 4.2. Creating *memory sequences* for every *execution sequence*, of which we know there can be at most size of N, will have $O(N(N+E)\log(N+E))$ time complexity.

5.4 Grouping executions

Once the *memory sequence* for every *execution sequence* is finished, HASTE groups executions together on VMs. We define the ILP for minimizing financial cost of allocating VMs to run the schedule produced by HASTE in Section 4.3. While HASTE is still bound by the ILP's restrictions on placing *memory sequences* on VMs in terms of the maximum memory and available vCPUs of the VM, we cannot directly solve the ILP, as it is well known that solving the ILP is NP-Hard [20]. In order to approximate the solution to the ILP, we use some intuitions about the properties of *memory sequences* and the VMs. First, if there exist b *memory sequences* where $b \leq P$, where every *memory sequence* has a maximum amount of memory needed at any time below: M/b, then all of these memory sequences can be scheduled on the same VM without any chance of running out of memory. Secondly, if a set of *memory sequences* all end (meaning they all require 0 memory) at similar times, then assigning them all to the same VM means that, when the VM is torn down after the last *memory sequence* finishes, there will be little wasted time for the processors running the other *execution sequences*.

Using these intuitions, we formulate our heuristic to pack *execution sequences* into VMs. We first initialize an array of P arrays, with each subarray initially containing no elements. Then, for every *memory sequence*, we determine its maximum memory needed at any time,

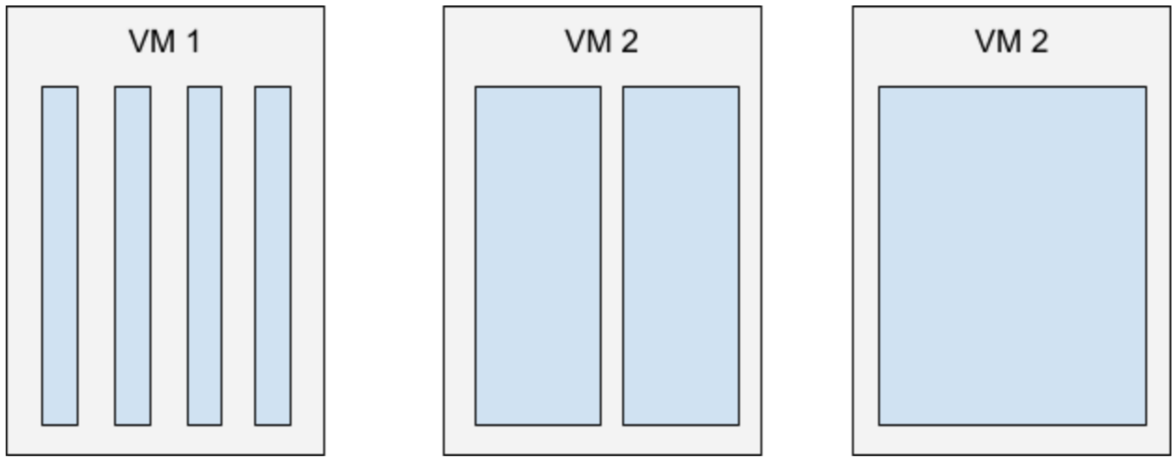

Figure 5.1: VM packing

which we will denote as F. We then iterate through the P arrays, and, at every array P_{i}, we determine if: $F \leq (M/(P-i))$. When we find an array that meets this criteria, we append the *memory sequence* to that subarray. We assume that every VM has at least enough memory handle the largest F among all *memory sequences*, so every *memory sequence* will be placed in a subarray. Then, for each subarray, we sort its *memory sequences* by their time they end. Finally, for each sorted subarray, we pop $P-i$ *memory sequences* from the front. We then allocate a single VM, give it a startup time of 0 and a teardown time of the maximum end time of the $P-i$ popped *memory sequences*, and place each corresponding *execution sequence* on that VM. Because the number of *memory sequences* never exceeded the number of vCPU for that VM, and the total memory needed will never exceed the total available memory for that VM, all placements are correct. Figure 5.1 shows what this could result in in practice, with each blue column representing the maximum allocation required by *memory sequences*. Because we are sorting all *memory sequences*, of which there can be at most size of N items via Theorem 1, the time complexity of this stage is $O(N\log(N))$.

Chapter 6

Empirical Evalution

In order to demonstrate that HASTE achieves the low latency of LWB without an undue financial cost burden from allocating VMs, we created some microbenchmarks by generating workloads of DAGs from sample Azure Function traces. These Azure Function traces come from work by Cortez et al. [8] that documented a subset of all invocations of Azure functions in July of 2019. This data includes logs of function execution time and memory allocations. We sample this data to create the nodes of the DAGs in our workloads. We simulate running each generated workload with HASTE, and compare this to simulated executions of other algorithms. We sample our functions from these Azure traces to ensure that the DAGs we construct have a basis in production function executions. However, determine the size and "shape" of the DAGs ourselves in order to be able to finely control certain DAG parameters, such as number of nodes or edges, so that we can determine how HASTE performs relative to comparable algorithms for a variety of workload types.

6.1 Algorithms and Baselines for Comparison

We consider a few baseline algorithms to compare HASTE to when running workloads. We chose these algorithms because they are execution strategies that we have decided would be common for users running data-driven applications without specific optimizations. These algorithms include:

- **Single Threaded Execution (STE):** Single threaded execution is what occurs when a user runs a set of DAGs naively, likely on their own personal computer, e.g., executing all cells in a Jupyter Notebook from top to bottom on your personal computer. This entails running each DAG serially and running each node in the DAG in topological order. This has the benefit of requiring only a single machine and no communication, but will likely have very high latency.

- **Naive Parallel Execution (NPE):** Naive parallel execution occurs when each DAG runs, single threaded, on its own VM, with each DAG executing in parallel with all other DAGs. This is a likely option if a user has access to a large cluster but does not use any algorithm to analyze and schedule the nodes in the DAG onto the cluster.

6.2 Workload Generation

When creating the population of Azure Function traces to sample from for our workloads, we take each invoked function, its average execution time, and its average allocated memory to be our node n_{i}, its processing time r_{i}, and its max allocation x_{i}, respectively. We only sample from Azure Functions that have over 1000 ms of runtime in order to meet our low relative communication cost requirement for LWB valid DAGs. To this end, because the Azure Function traces do not come with communication times of the results of the functions, we use the intuition that they are likely to be proportional to the max allocations of the functions. We construct artificial communication times by scaling the max allocations of each function in order to be compliant with the low communication cost, the exact requirements of which are described in Section 3.1. We use this method to determine the node and edge properties of all nodes and edges in all DAGs.

For every workload, we manually determine certain key variables. Those are the number of DAGs in the workload, the percent of total nodes in the workload that are duplicates, the number of edges per DAG, and the number of "layers" per DAG. The percent of duplicate nodes in the workloads use a total number of nodes that counts replicas across DAGs for every time they appear, and uses a number of duplicate nodes that does not count the first encountered replica of that node. For example, if there are 4 replicas of node A, this will only add 3 to the count of replicas, as without the extra 3, node A would not be considered a replica. Layers of a DAG is defined in depth in Iverson et al. [19].Briefly, Iverson states that the nodes in DAGs can be arranged into discrete layers. At the index 0, all nodes have no parents, and at each subsequent layer, all the nodes in said layer only have ancestors in lower index layers, and only have descendants in higher index layers. The number of layers a DAG roughly determines the amount of parallelism that is achievable when execution the DAG, and is an important variable to control when testing the scalability of a parallel scheduler like LWB. In order to enforce the number of edges, each DAG g_{t} has at least size of N_{t} edges, or at least as many edges as nodes, and so any number of edges input is always greater than or equal to the number of nodes.

6.3 Latency Testing

In order to achieve its low latency execution, HASTE leverages the optimal scheduling of LWB. It also removes equivalent nodes via merging, which prevents duplicated execution. This provides a great deal of speedup relative to STE, as STE executes all nodes in sequence. Since NPE executes every DAG in parallel, the speedup of HASTE relative to NPE is slightly less than compared to STE. Still, NPE does not allow for task duplication, and every VM running an individual DAG is limited by its finite processors, so NPE will still be slow relative to HASTE.

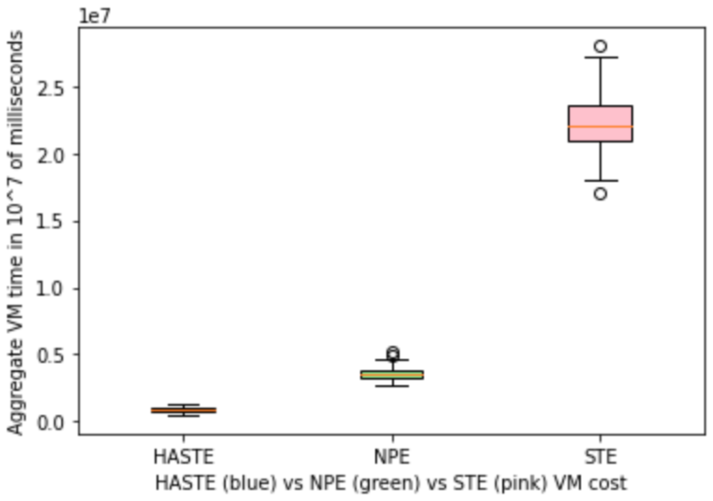

Figure 6.1: Latency comparison of 100 samples of HASTE, STE, and NPE on 100 node DAGs

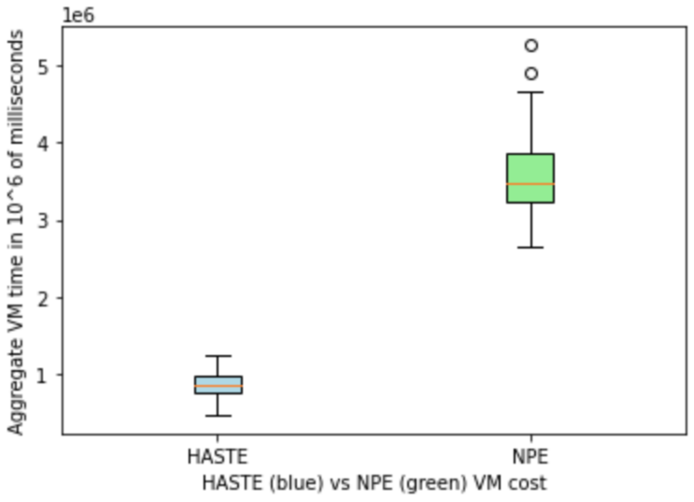

Figure 6.2: Latency comparison of 100 samples of HASTE and NPE on 100 node DAGs

For Figures 6.1 and 6.2, we sampled the Azure Functions dataset 100 times, constructed workloads with 10 DAGs, each with 100 nodes, 5 layers, 150 edges, and 10% of all workload nodes having an equivalent node with at least one other DAG. Figure 6.2 uses the same data as Figure 6.1, but excludes the STE executions for the sake of readability. These figures make it clear that HASTE vastly outperforms both STE in terms of latency, averaging a 25x speedup on this configuration. This is to be expected, as STE runs every individual node serially, while HASTE utilizes LWB to maximally parallelize execution. Compared to NPE, HASTE achieves an average 4x speedup for this configuration. While NPE has increased parallelism relative to STE, HASTE parallelizes execution at the level of the node, which contributes to its higher performance.

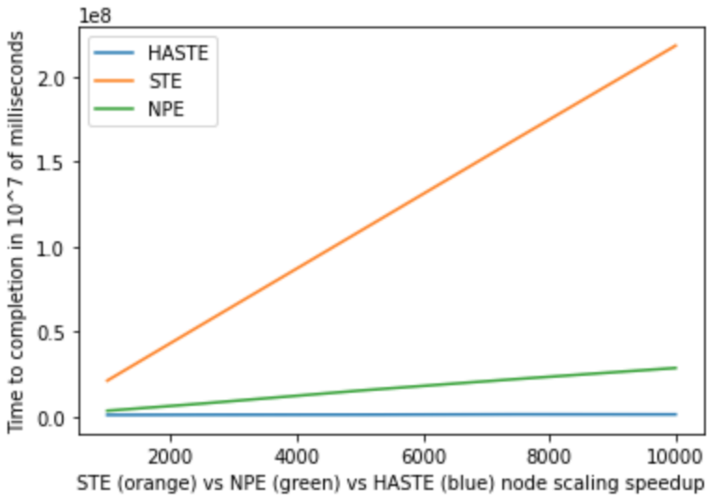

Figure 6.3: Latency comparison of 100 samples of HASTE, STE, and NPE on DAG sizes ranging from 100 to 1000 nodes

For Figure 6.3, we tested the scaling capabilities of each algorithm using total number of workload nodes ranging from 1000 to 10000. Similarly to Figures 6.1 and 6.2, at each datapoint, we used 100 samples from the dataset, constructed 10 equally sized DAGs per workload, and kept the percent of workload nodes which had an equivalent at 10%. However, we scaled the number of edges and layers proportionally to the number of nodes, so as to simulate an increase in overall DAG size. Figure 6.3 show how HASTE, and more specifically LWB, scales very well as the number of nodes in the workload scales. At 10000 nodes, HASTE experiences an average speedup over 188x versus STE, as well as a speedup of over 24x relative to NPE.

6.4 Cost Testing

As is made clear in the previous Section, HASTE generates schedules that are much faster than both STE and NPE by leveraging the *minimal makespan* schedules that LWB generates. As we will explore in this Section, HASTE also generally outperforms STE and NPE when it comes to financial cost, measured as the sum of VM time of all allocated VMs. This is because neither STE nor NPE utilize the multiple vCPUs that exist on a single VM, with STE executing everything on a single processor of a single VM, and NPE executing each DAG on a single processor of each DAG's dedicated VM. HASTE, on the other hand, is able to reduce the total amount of compute through equivalent node merging, and is able to utilize the multiple vCPUs in a single VM through its execution grouping. Equivalent node merging is described in Section 3.1, and execution grouping is described in Section 5.4.

Because both STE and NPE execute all DAGs on a single processor, and neither STE nor NPE utilize equivalent node merging, they will both take the same amount of summed VM time for any input or system configuration. As such, in this Section, we will refer to them together as STE/NPE.

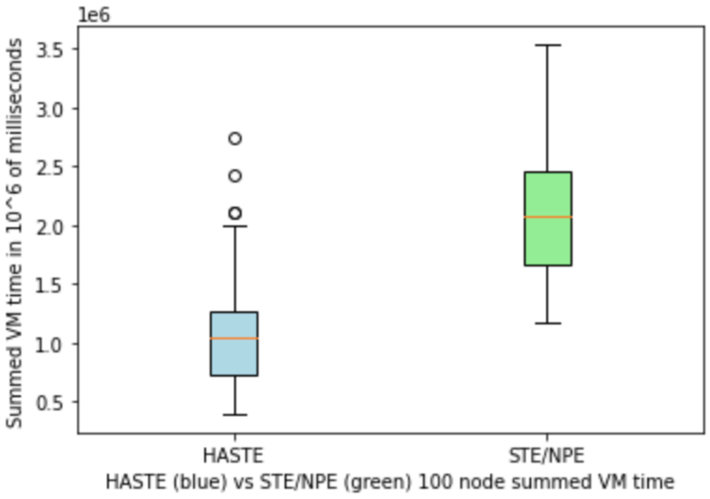

Figure 6.4: Summed VM time comparison of 100 samples of HASTE and STE/NPE on 100 node DAGs

For Figure 6.4, we again sampled the Azure Functions dataset 100 times, constructed workloads with 10 DAGs, each with 100 nodes, 5 layers, 150 edges, and 10% of all workload nodes having an equivalent node with at least one other DAG. The simulated VMs had 4vCPU and 2500MB memory. For reference, the average node has a max allocation of 175MB, with the maximum among all nodes being 1227MB. While not nearly as stark a contrast as the HASTE speedup , we can see in Figure 6.4 that HASTE does incur lower summed VM time compared to STE/NPE through the utilization of multiple vCPUs per VM. With this configuration, HASTE incurred on average roughly half the financial cost of STE/NPE.

For Figure 6.5, we tested the ability of HASTE to leverage intra-VM parallelism by running simulations with VMs that contained between 1 and 9 vCPUs. We used the same input and system parameters as the tests shown in Figure 6.4, except with 4000MB memory

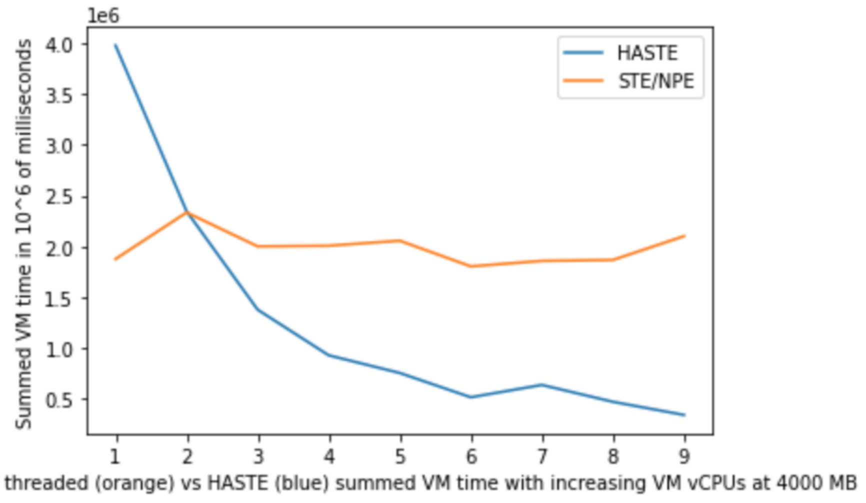

Figure 6.5: Summed VM time comparison of 100 samples of HASTE and STE/NPE on 100 node DAGs, with vCPUs per VM ranging from 1 to 9

per VM. For the simulation configuration where VMs had 1 vCPU, we see that HASTE performs drastically worse than STE/NPE. Because with 1 vCPU there is no intra-VM parallelism to exploit, HASTE must run all of its *execution sequences* on individual VMs. The task duplication of LWB means that HASTE must complete more total computation than STE/NPE and therefore use more summed VM time than STE/NPE. However, with 2 vCPU onward, the gains in intra-VM parallelism eclipse this extra computation, and HASTE performs better than STE/NPE from 2 to 10 vCPUs per VM. HASTE also performs generally better as the number of vCPUs increase while STE/NPE stays the same.

Chapter 7

Future Work

While this report works out a formalization of handling the memory requirements of executing DAGs on cloud computing resources and provides some heuristics to achieve low cost and low latency executions in practice, there is still much that can be done to iterate on the HASTE project. This includes testing the use of other task dependence graph scheduling algorithms. Also, different heuristics and strategies could yield even better performance than we recorded through our testing. Finally, integrating this work with other systems research, particularly research concerning automated machine learning, could be of great benefit to the computer science community.

7.1 Other DAG scheduling algorithms

While we chose LWB because of its simplicity and optimality guarantees, other task DAG scheduling algorithms exist, including others that assume an unbounded number of processors [17]. Each algorithm has its own set advantages and disadvantages.

Critical Path Fast Duplication Algorithm

The Critical Path Fast Duplication (CPFD) Algorithm, creating by Ahmad and Kwok in 1998 [1], is an iteration on previous task duplication and unlimited processor algorithms such as LWB. While CPFD does not have the same optimality guarantees as LWB, it is not restricted to low communication DAGs as LWB is, and, for most DAGs, outperforms LWB and most other scheduling algorithms. CPFD first groups nodes in the DAG into three distinct groups: Critical Path Nodes, which are nodes which reside directly on the critical path of the DAG, In Branch Nodes, which are nodes on paths that connect to the critical paht, and Out Branch Nodes, which are nodes which don't fit either of the previous groups. CPFD then constructs a Critical Path Node dominant sequence which prioritizes the execution of Critical Path Nodes first, then In Branch Nodes, and lastly Out Branch Nodes. CPFD has $O(N^4)$ time complexity [17], and produces schedules, which, according to testing by Ahmad and Kwok, are lower latency than most other task DAG scheduling algorithms. As in LWB, CPFD does not account for memory needs for either function execution or communication, so the HASTE method of constructing *memory sequences* and placing them on VMs could be applied to CPFD to generalize HASTE for many different DAG types.

Dynamic Critical Path (DCP) Algorithm

The Dynamic Critical Path Algorithm, created in 1996 also by Ahmad and Kwok [16], is a task DAG scheduling algorithm which unlike LWB and CPFD, does not allow for task duplication. However, it does still assume an unbounded number of processors. The DCP algorithm operates by heavily prioritizing nodes which have the least difference between the latest possible time they can be scheduled and the earliest possible they can be scheduled relative to the critical, known as “mobility”. The time complexity of DCP is $O(N^3)$. Because DCP is not task duplicating, it is likely to produce higher latency schedules compared to task duplicating algorithms such as CPFD and LWB, but is also likely to create fewer *execution sequences*. This means that, if used as a replacement for LWB in HASTE, it can produce cheaper VM allocations. Furthermore, this can maximize the benefits of equivalent node removal via the *merged DAG* that HASTE creates.

7.2 Potential Optimizations of HASTE

HASTE’s performance, both in terms of latency and total compute cost, could be improved by utilizing more sophisticated heuristics, as well as by leveraging certain properties of the problem space.

LogP Communication for Distributed Computing

HASTE assumes a relatively simple model of communication, as well as the memory needed to handle said communication, for the purpose of constructing *memory sequences*, which is laid out in detail in Section 5.3. However, this simple model relies on a number of optimistic assumptions which are not necessarily true of real networks in the cloud. A potential avenue to increase the robustness of HASTE *memory sequences*, while also potentially helping increase performance, would be to incorporate the LogP model [4]. The LogP Model of communication for distributed computing very closely resembles production distributed machines. Accounting for the finite capacity of the network, as well as the per processor communication bandwidth, among other LogP Model parameters, would improve calculation of the communication time between nodes in the DAGs. Also, because HASTE now allocates the memory required for communication pessimistically, utilizing LogP assumptions could reduce the amount of time communication-related memory is allocated by either the sender or receiver. This could potentially allowing HASTE to minimize the amount of VM time required to run its created schedules even further.

Optimizations for VM packing

There are a number of smaller optimizations that could be considered when fitting the *execution sequences* onto VMs. Firstly, if multiple *execution sequences* that run duplicated tasks are placed on the same VM, only one of them needs to run, freeing the memory used by the others. Furthermore, if *execution sequences* that communicate to each other are placed on the same VM, they would only needed to allocate the space necessary for the intermediate results once, unlike communication between VMs, which requires allocation for the intermediate results in both the sender and receiver. Finally, while we assume that intermediate results sent between DAG nodes are placed as soon as possible on the reciever's VM, these results could be shuffled around VMs as long as they had enough time to be communicated around, leaving room to "cache" these results intelligently and further reduce the amount of aggregate VM time used.

7.3 Integration with other AutoML offerings

Our work on HASTE, while general to any DAG execution, is very closely related to work in the space of automated machine learning, or AutoML. AutoML work seeks to create tools and processes for aiding data scientists in creating machine learning models/pipelines. AutoML tools target steps in the ML lifecycle from data visualization [6] to automated hyperparameter tuning [18] to logging of checkpoints in model training [10]. One system in particular which we believe could be used in conjunction with HASTE to great effect is HELIX, by Xin et al. [24]. HELIX is an ML system that optimizes execution across iterations of an ML pipeline by storing intermediate results that are frequently used in each iteration. HASTE could leverage HELIX's stored intermediate results to eliminate Sections of computation in the HASTE *merged DAG*, which would create even lower latency schedules and use less VM time.

Chapter 8

Conclusion

In this report, we demonstrate that DAG scheduling algorithms that produce fast schedules but require an unbounded number of processors have viability with modern cloud computing offerings. We presented HASTE, an execution optimizer for workloads of sets of DAGs being run on serverless VMs. HASTE utilizes the concept of a *merged DAG* to eliminate equivalent nodes between DAGs in the workload, thereby preventing unnecessary duplicated computation of the same output. HASTE then schedules the merged DAG using the Lower Bound (LWB) algorithm, an unbounded processor, task duplication algorithm, which, as we proved, produces schedules for merged DAGs have equivalent latency to scheduling each DAG individually with LWB and waiting for all of them to complete. The LWB algorithm produces schedules for individual processors we label *execution sequences*. HASTE takes these *execution sequences* and creates timelines of how much VM memory is necessary to run them, timelines we denote as *memory sequences*. When determining the necessary memory, HASTE leverages S3 style cloud storage to store function outputs when the schedule permits in order to minimize the amount of VM memory needed. Finally, with a the *memory sequences* constructed, HASTE assigns *execution sequences* to VMs with the goal of minimizing VM time across all allocated VMs, and thereby financial cost to the user, across all VMs, without violating the memory requirements of each sequence.

With this outline for the HASTE system established, we formalize the ILP that packs the *execution sequences* into VMs to minimize VM time. We then describe our heuristics that approximate the solution to the ILP, and, using DAGs constructed from Azure Function invocation traces, we show that our algorithm produces low latency schedules compared to similar algorithms while also incurring reasonable VM time.

Bibliography

[1] Ishfaq Ahmad and Yu-Kwong Kwok. "On Exploiting Task Duplication in Parallel Program Scheduling". In: 9.9 (1998), pp. 872–892.

[2] Benjamin Carver et al. "Wukong: a scalable and locality-enhanced framework for serverless parallel computing." In: (2020).

[3] D. Sculley et al. "Hidden Technical Debt in Machine Learning Systems". In: 28 (2015), pp. 2503–2511.

[4] David Culler et al. "LogP: Towards a realistic model of parallel computation". In: (1993).

[5] Denis Baylor et al. "Continuous Training for Production ML in the TensorFlow Extended (TFX) Platform". In: (OpML 19) (2019), pp. 51–53.

[6] Doris Jung-Lin Lee et al. "Lux: Always-on Visualization Recommendations for Exploratory Data Science". In: (2021).

[7] Edo Liberty et al. "Elastic Machine Learning Algorithms in Amazon SageMaker". In: (2020), pp. 731–737.

[8] Eli Cortez et al. "Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms". In: (2017).

[9] Matei Zaharia et al. "Accelerating the Machine Learning Lifecycle with MLflow". In: 41.4 (2018), pp. 39–45.

[10] Rolando Garcia et al. "Hindsight Logging for Model Training". In: (2020).

[11] Thomas Kluyver et al. *Jupyter Notebooks-a publishing format for reproducible computational workflows.* Vol. 2016. IOS Press, 2016.

[12] Vaishaal Shankar et al. "Numpywren: Serverless linear algebra". In: (2018).

[13] Eugen Cepoi and Liping Peng. "Runway - Model Lifecycle Management at Netflix". In: (2020).

[14] Jean-Yves Colin and Philippe Chrétienne. "CPM scheduling with small communication delays and task duplication". In: 39.4 (1991), pp. 680–684.

[15] Xiaoliang Dai et al. "FBNetV3: Joint Architecture-Recipe Search using Predictor Pretraining". In: (2020).

[16] Yu-Kwong Kwok and Ishfaq Ahmad. "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors". In: 7.5 (1996), pp. 506–521.

[17] Yu-Kwong Kwok and Ishfaq Ahmad. "Static scheduling algorithms for allocating directed task graphs to multiprocessors". In: 31.4 (1999), pp. 406–471.

[18] Liam Li et al. "Exploiting Reuse in Pipeline-Aware Hyperparameter Tuning". In: (2018).

[19] Füsun Özgüner Michael A. Iverson and Gregory J. Follen. "Parallelizing existing applications in a distributed heterogeneous environment." In: (1995).

[20] Christos H. Papadimitriou and Kenneth Steiglitz. "Combinatorial optimization: algorithms and complexity". In: (1998).

[21] Amazon Web Services. *Step 2: Create your EC2 resources and launch your EC2 instance.* URL: `https://docs.aws.amazon.com/efs/latest/ug/gs-step-one-create-ec2-resources.html`.

[22] Vikram Sreekanti et al. "Optimizing Prediction Serving on Low-Latency Serverless Dataflow". In: (2020).

[23] Bichen Wu et al. "FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search". In: (2019), pp. 10734–10742.

[24] Doris Xin et al. "HELIX: Holistic Optimization for Accelerating Iterative Machine Learning". In: (Sept. 2019).

[25] Doris Xin et al. "Production Machine Learning Pipelines: Empirical Analysis and Optimization Opportunities". In: (2021).