UNIVERSITY OF CALIFORNIA
SANTA CRUZ

AN ANALYSIS AND APPLICATION OF GRAPHENE ON ALIBABA

Undergraduate Senior Thesis

in

COMPUTER SCIENCE AND ENGINEERING

by

Steven Xue

June 2025

Professor Andi Quinn

Professor Abel Souza

Professor Lindsey Kuper

Table of Contents

List of Figures

Abstract

An Analysis and Application of Graphene on Alibaba

by

Steven Xue

The modern landscape of cluster computing often deals with increasingly complex workloads characterized by DAG-structured jobs. One recent trace that highlights these complex characteristics is the 2018 Alibaba cluster trace. These modern workloads warrant a re-evaluation of existing algorithms to determine their efficacy in real-world scenarios. Graphene is a near-optimal algorithm for scheduling Directed Acyclical Graph (DAG) workflows. In its original evaluation with a synthetic workload, it achieved up to a 50% reduction in job completion time compared to a critical path scheduler. However, Graphene bases its claims on synthetically generated trace data. Since real world trace data is now available, such as the Alibaba cluster traces, this allows researchers to answer more complicated questions in cluster scheduling. Therefore, we analyzed concerns like queueing delay in the Alibaba cluster, then re-evaluated Graphene on the Alibaba trace to demonstrate its efficacy on real-world trace data.

Chapter 1

Introduction

Modern cluster workloads are constantly evolving to be increasingly complex. Handling such complexity presents new problems and challenges in cluster scheduling and cluster performance. However, improving cluster performance requires understanding the intricacies of real world traces. Some of these contemporary issues in clusters include job starvation, long wait times, and cluster efficiency. Thus, this paper will delve into these issues on the Alibaba trace to better understand what problems occur in the real world, and analyze the efficacy of state-of-the-art job scheduling algorithms under real-world workload settings.

While there has been a significant amount of prior research done on non-DAG batch jobs [3], modern traces have shown that real-world jobs typically do not look like singular instances of resources. In particular, jobs run on large clusters typically resemble a group of instances ordered in a DAG or directed acyclical graph. These DAG jobs result in new problems related to scheduling and utilization that previous analyses disregarded. With recently released trace datasets containing job DAG information, this warrants an investigation into how modern

clusters operate in running these complex jobs. Among these recently released traces is the Alibaba 2018 trace [2]. Thus, we are concerned with questions such as how queueing delay behaves in Alibaba, how the cluster deals with its resource demands, and what are the correlations between queueing delay and other cluster factors. From our analysis, we found that queueing delay is a huge problem, and can cost jobs minutes before running. We also found a significant correlation between a job's queueing delay and its number of instances. Additionally, we note certain characteristics in the cluster that point toward possible areas of improvement.

To deal with complex cluster characteristics and increasingly large job requests, job scheduling algorithms have been a heavy topic of interest. One algorithm is Graphene [1], an algorithm that claims to be nearly optimal in minimizing job completion time. However, Graphene's claims are bold, which warrants further investigation. The original Graphene paper used synthetically generated traces in its experiments, but how effective Graphene would be in the real world is our main concern. To solve this, we implemented and ran Graphene on a sample of the Alibaba trace and compared its performance to FIFO. We found that Graphene does indeed improve the overall completion time of the trace by about 30%, but improvements drastically decrease when the trace only contains large complicated jobs.

As this paper analyzes two different but related problems, we present two main chapters. The first addresses questions related to Alibaba and the second addresses questions related to Graphene.

Chapter 2

Real-World Workload Scenarios

2.1 Introduction

Companies often run diverse applications on large shared compute clusters. These applications request to run certain jobs that the cluster manager must schedule. Applications need increasingly complex jobs as they evolve, which puts an increasing amount of strain on the cluster. To quantify these performance problems, we can use certain metrics such as queueing delay and oversubscription factors.

In order to mitigate performance problems in clusters, we can use such metrics to better understand cluster performance. One way to do so is to analyze real-world execution traces, which are starkly different than synthetic traces often found in job scheduling simulations. Real-world traces are able to capture certain characteristics that synthetic traces cannot. For example, real-world workloads are dynamic and unpredictable, and can have drastically varying job requirements. Fortunately, in 2018, Alibaba released a new trace dataset consisting

of batch jobs and online service jobs [2].

Our analysis found a significant correlation between queueing delay and the size of a job DAG. We also highlighted certain characteristics of the cluster for possible improvements. For example, we identified the importance of critical path tasks queueing delay, the loose oversubscription factor in the cluster, and the imbalanced allocation of instances across machines.

2.2 Background

In job cluster scheduling, one of the most important user-facing problems is latency. Users of a cluster dislike needing long wait times before executing their jobs. In large clusters, queuing delay causes this latency. For example, cluster health at the time of job submission can vary depending on CPU or memory utilization, resulting in the scheduler waiting for a certain amount of available resources before the next user's task runs.

One method schedulers use to get around this resource requirement is oversubscription. Oversubscription assumes that jobs often request much more resources than they actually use. When the sum of the resource requirements amounts to the size of the cluster, the scheduler continues to run more jobs or oversubscribe the cluster. As long as the scheduler does not run out of resources, it can perform equally well with an increase in job throughput. For this analysis, a metric called efficiency is used, which is defined as the sum of all requested resources divided by the sum of used resources. If efficiency is above 1.0, then there is oversubcription. Inversely, if below 1.0, then there is under-provisioning. When at 1.0, then the cluster is fully utilized.

The Alibaba trace consists of a large number of batch jobs and service jobs. Each batch job consists of a group of tasks, where each task consists of a group of instances. A service job is a long-running job or application that is latency-sensitive. Similar to traces like Google Borg [3], Alibaba batch jobs are co-located with high-priority and latency-sensitive jobs to improve resource utilization and efficiency [3]. We did not choose to analyze Google Borg due to the lack of DAG data. Additionally, for this analysis, we only considered batch jobs.

A batch job in Alibaba is either a DAG job or a non-DAG job. We can represent a DAG job as a directed acyclical graph, but not for a non-DAG job. A DAG job is further broken down into tasks. Each node in the graph represents a certain task that belongs to that job. A parent of a node represents a dependency between two tasks; if task A is the parent of task B, task B cannot run until task A is fully complete. Each task can be further broken down into instances. A task consists of a bunch of instances where each instance has the same resource requests. Each instance is dependent on its parent task. Therefore, an instance cannot run until all instances of its parent task are fully complete. In contrast, a Non-DAG job is a singular task that has any number of instances. From this point onward, we use the job-task-instance model when describing batch jobs in the Alibaba trace.

2.3 Analysis

The published trace tracks approximately 4 million batch jobs over 8 days. In total, 1.3 billion instances ran on the cluster. The cluster consists of 4023 homogenous servers with 96 CPU cores and 1 unit of normalized memory each. Not all machines have 100% uptime; at

any point in time, most machines are utilized while some are inactive.

The majority of the scheduled jobs in Alibaba are small jobs lasting less than an hour. Figure 2.1 shows a size breakdown of the scheduled jobs. 38% last under a minute, while about 60% last under an hour. Jobs greater than an hour represent a minuscule amount of the trace at 1%.

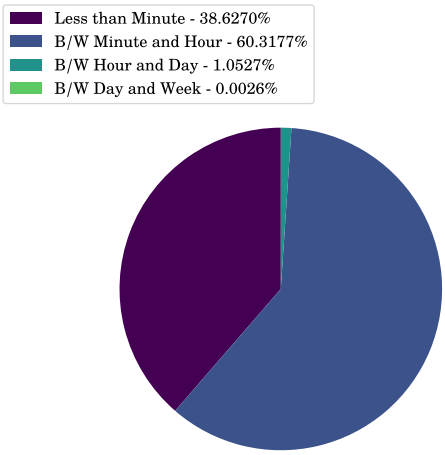


Figure 2.1: Breakdown of the size of scheduled jobs.

As an incredibly complicated trace, this analysis is primarily concerned with two questions: Where and how does queueing delay emerge? What are the unique characteristics of the scheduler that may impact performance? An analysis of queueing delay, resource constraints, and critical path analysis answers the former. Meanwhile, a look into the cluster's oversubscription and scheduling policy addresses the latter.

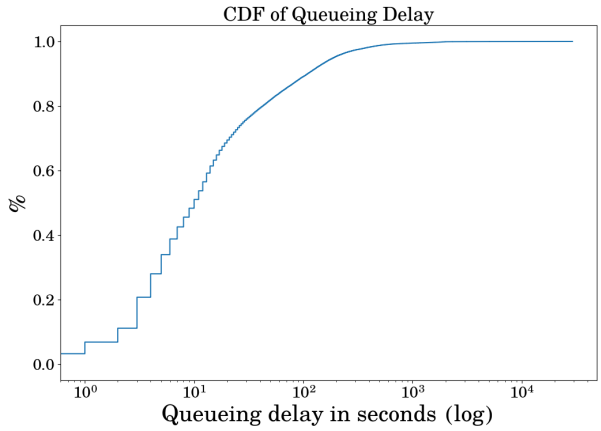


Figure 2.2: CDF of logged queueing delay computed over instances. Most of the queueing delay values are large.

2.3.1 Queueing Delay

Queueing delay in the Alibaba cluster is a significant problem. For all sections in this analysis, we measured queueing delay in seconds. Figure 2.2 shows how the majority of the queueing delay values over all instances are large numbers. This implies the Pareto principle, where a small subset is responsible for most of the observation; while most jobs are small, the minority of large jobs are occupying the most of the cluster, resulting in large jobs causing most of the queueing delay.

In general, large jobs dominate queueing delay in the Alibaba cluster. A significant correlation between the total number of instances in a job and queueing delay is shown in Figure 2.3 and Figure 2.4. In particular, it is at least incredibly unlikely for a job with thousands of instances to experience a minimal amount of queueing delay.

Interestingly, even though the trace consists of single instance jobs and DAG jobs,

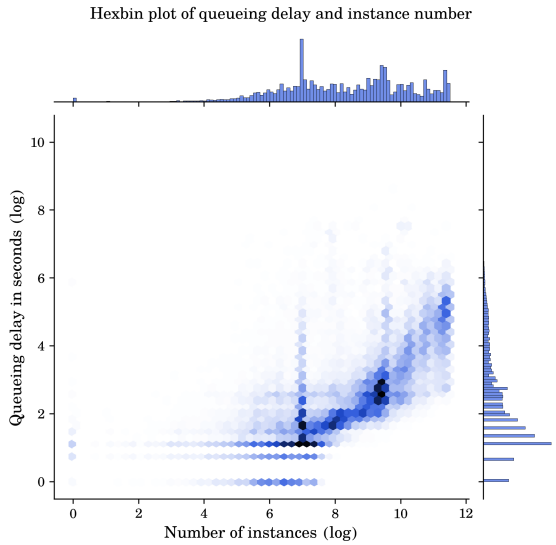


Figure 2.3: Queueing delay over number of instances in the corresponding job, graphed over instances.

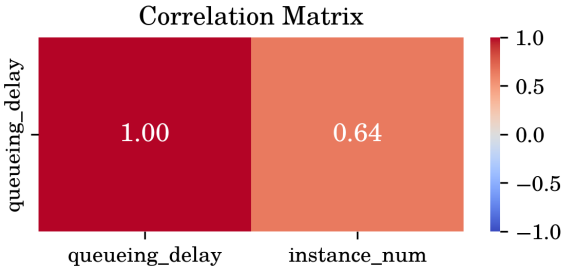


Figure 2.4: Correlation between queueing delay and number of instances in the corresponding job in a square matrix.

the smaller number of large DAG jobs quickly overwhelms queueing delay metrics due to its sheer size; either the large DAG jobs impact the huge percentage of small running jobs or the statistics from the large jobs instances dwarf the small jobs.

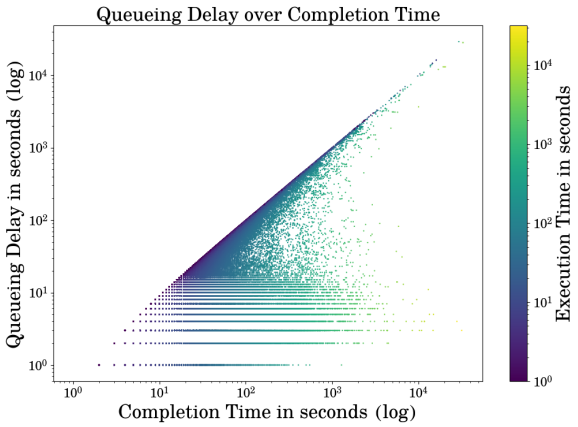


Figure 2.5: Queueing delay over completion time across a sample of instances. Color represents the length of the instance with darker color representing shorter jobs, not logged.

Figure 2.5 shows the relationship between queueing delay, completion time, and execution time over a small sample of the trace. Completion time is the sum of the queueing delay and execution time while execution time only includes the time an instance spent running. With completion time and queueing delay on the x and y axes respectively, we represent execution time by color, with darker colors symbolizing a shorter runtime. Both queueing delay and completion times are on a log scale while execution time is linear. In an ideal cluster, all instances will lie on the horizontal $y = 0$ line. This is because instances on this line have no queueing delay, resulting in their execution time matching their completion time. Points that fall on the diagonal represent small instances delayed by the scheduler even though they ran for a very

short time.

Based on the graph, jobs with exceedingly large execution times are more likely to have some amount of queueing delay; the most delayed instances tend to be longer-running jobs. While most short-running jobs experience less queueing delay, a significant portion experience a substantial amount. In contrast, a good portion of long-running instances do not have much queueing delay. Therefore, it is likely that the scheduler does not differentiate between short and long-running jobs like SJF.

2.3.2 Queueing Delay and Resources

While previous analyses of the trace cite memory as the main limiting factor [2], no correlation has been found between queueing delay and various statistics of resources in the cluster. A correlation matrix in Figure 2.6 shows the Pearson coefficient between queueing delay and other metrics. A Pearson coefficient of 1.0 means the two metrics are correlated, while a coefficient of -1.0 means the two metrics are inversely correlated. A value of 0 means the two metrics have no correlation at all. Importantly, the highest Pearson coefficient with queueing delay is with instance_num, but unfortunately, most metrics are not statistically correlated with queueing delay. As an example, Figure 2.7 displays a graph between queueing delay and remaining cluster memory.

Therefore, while the cluster is memory constrained as shown in Figure 2.8, there is no observable relationship between queueing delay and memory. This may be due to the failure to consider online service jobs, which this analysis does not further explore.

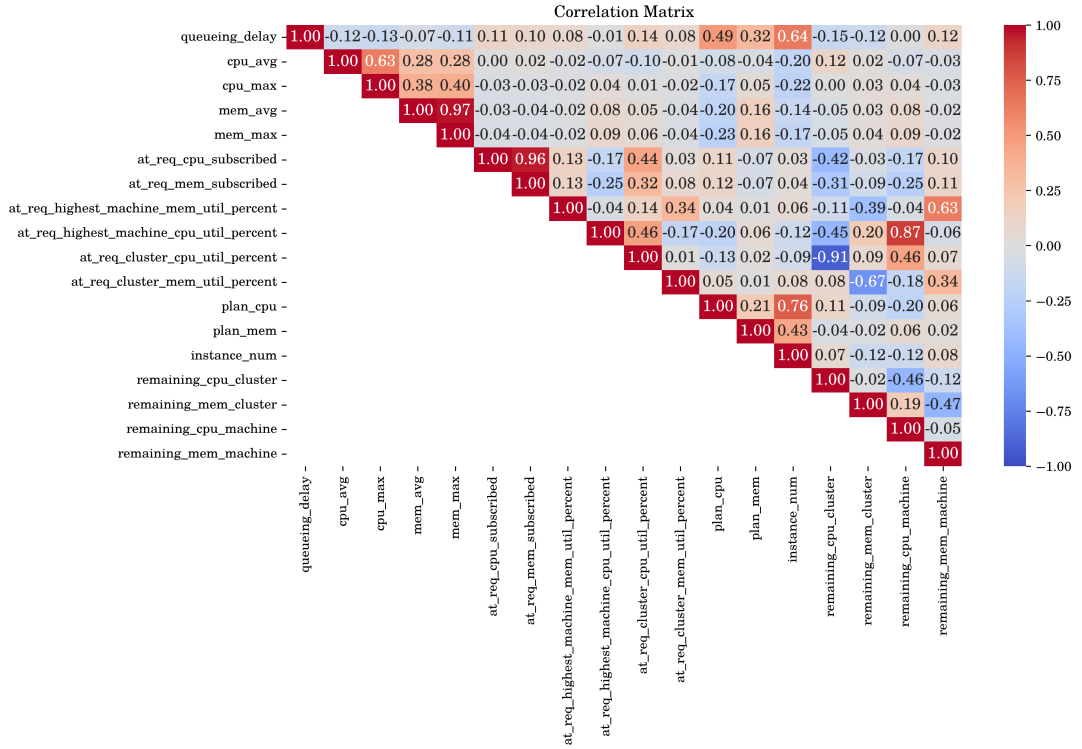


Figure 2.6: Correlation matrix between queueing delay and various measurements. Queueing delay, plan_cpu, plan_mem, and instance_num are logged.

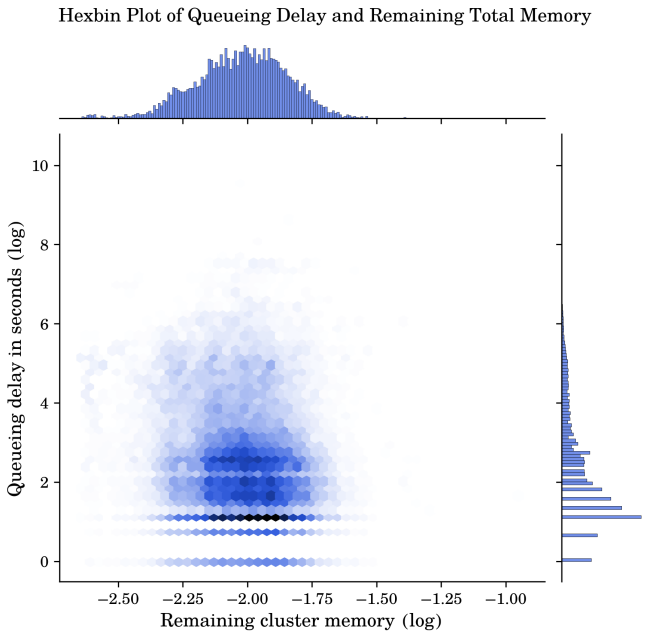


Figure 2.7: Example jointplot between queueing delay and remaining total memory in the cluster at the earliest schedulable time.

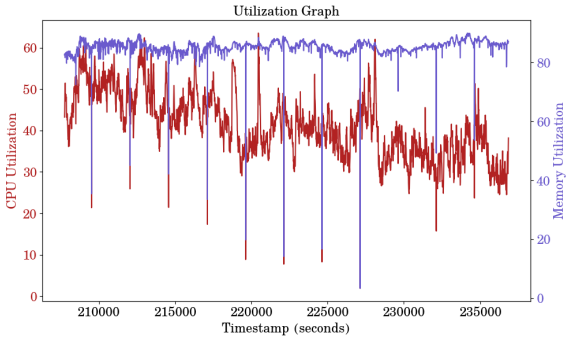


Figure 2.8: Total cluster utilization over time. For a utilization graph over the entire trace, see Figure 14 from the original paper [2].

2.3.3 Critical Path

A unique trait of DAG jobs is how completion time is primarily impacted by tasks on the critical path. In this context, the critical path of a job DAG is the tasks in a job whose queueing delay dominates each stage. For example, if a task is waiting on two parent tasks, the longer-running parent task will be part of the critical path. The important instances to then consider are those that correspond to critical path tasks.

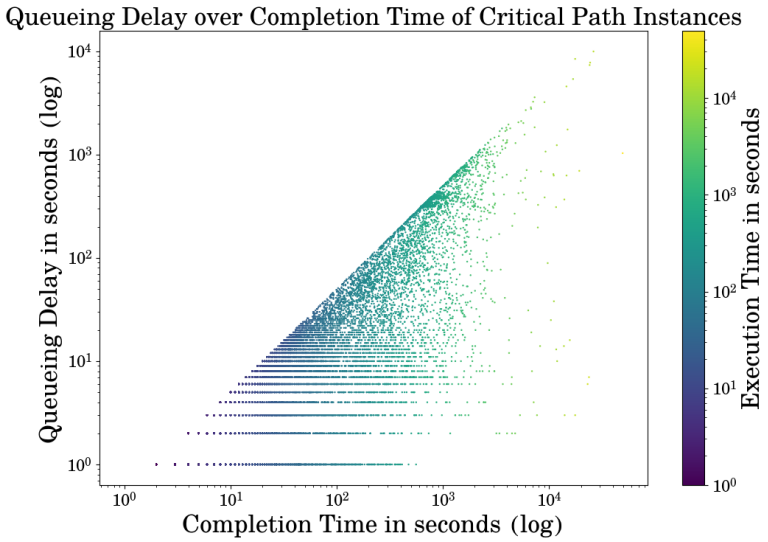
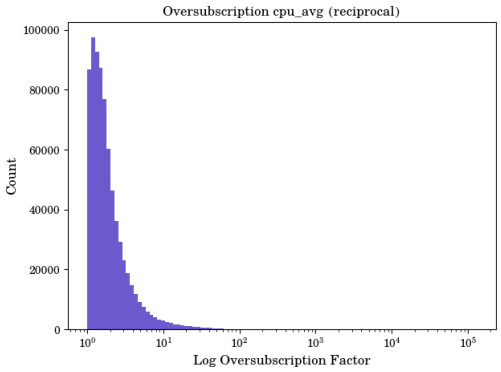


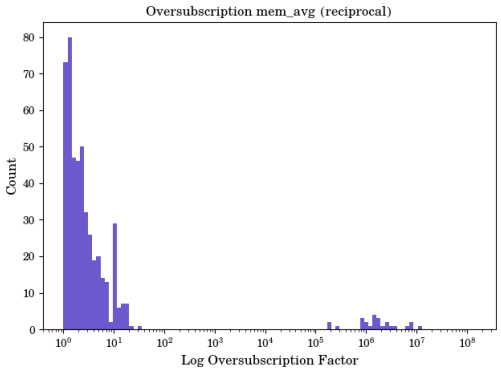
Figure 2.9: Critical path instances queueing delay over completion.

By replicating the previous graph of queueing delay over completion time from subsection 2.3.1 onto Figure 2.9, short-running jobs with large amounts of queueing delay are no longer visible. This enforces the idea that a small subset of jobs are responsible for the majority of the queueing delay. Therefore, one viable method to tackle queueing delay may be a critical path-based approach.

2.3.4 Oversubscription



(a) CPU oversubscription (logged).



(b) Memory oversubscription (logged).

Figure 2.10: Oversubscription values over CPU and memory averages.

Due to clusters having limited resources, some job schedulers have relied on oversubscription to improve utilization. In practice, this performs well thanks to inaccurate resource requests. For example in the Alibaba case, the task *task_ODYzMjgxNjM3NDU5NTUwMjc5MA==* in job *j_2057344* requests 6000 instances with 2 CPU cores and 2.74 units of normalized memory each. As each machine in the cluster only has 1 unit of normalized memory, this task re-

quested 4 times the amount of total memory in the cluster. However, out of all 6000 instances, the maximum amount of memory used is 0.19 normalized units or 7% of the requested amount.

As shown in Figure 2.10, while Alibaba does perform oversubscription, the requirements appear to be loose and not bound to a specific value. At certain times, the cluster is incredibly oversubscribed on memory, with efficiency values as high as 1,000,000. This hints towards more jobs that have poor estimates of requested resources, similar to the example job above. At least for certain jobs, the scheduler ignores large memory requirements due to unreasonable requests.

2.3.5 Scheduling Policy

At any point in time, the scheduler is able to process several dozen instance requests per second. An interesting characteristic of the scheduler is that it prefers to schedule in chunks; the scheduler waits before scheduling instances of a task all at once. This waiting may lead to some loss in performance, as running any portion of a task greedily may allow for a small speedup.

Large jobs typically are spread across multiple machines, implying that locality is not a big concern in the cluster. It is likely that communication between machines is negligible compared to a job's runtime. Additionally, the scheduler uses a round-robin-like method when assigning specific instances to a machine. However, the scheduler assigns a non-uniform number of instances to each machine for any given task. Figure 2.11 shows a breakdown of how the scheduler assigned instances for the job j_2057344. The graph itself shows how many instances each unique machine received. For example, 1700 machines received only one instance while

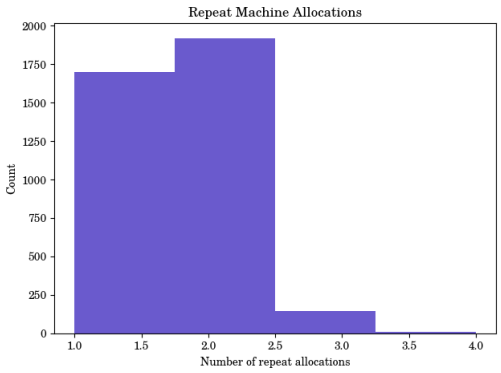


Figure 2.11: Histogram of number of times a machine was assigned the same instance.

almost 2000 machines received the same instance twice. Fewer than 200 instances used the same machine 3 times.

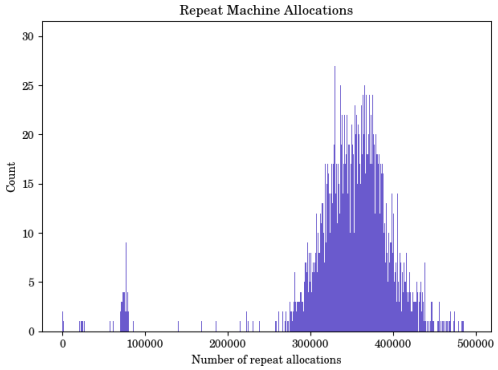


Figure 2.12: Histogram of instances assigned to machines.

The scheduler does not enforce a uniform distribution of allocated instances as shown in Figure 2.12, which displays a histogram of the amount of times machines are given an instance. Despite being a homogenous cluster, job allocation is not uniform. As a result, spreading out job allocations across machines evenly may help improve efficiency in the cluster.

Chapter 3

Graphene

3.1 Introduction

Effectively scheduling jobs on a large-scale compute cluster is a continuously challenging problem. To address this, job scheduling algorithms have been a critical area of research. One such algorithm is Graphene [1], an algorithm that claims to be nearly optimal when creating offline job schedules. However, Graphene's original claims utilized synthetic traces. Therefore, whether its claims hold when simulated on real-world traces was our concern. Thus, the next section of this report focuses on the implementation of Graphene and a comparison of its performance against FIFO. Additionally, we utilized the previous Alibaba trace for experiments to analyze Graphene's performance in real-world scenarios.

In our results, we found that compared to FIFO, Graphene did improve general completion time given a sample real-world trace. However, this improvement was less than expected. Comparing this improvement to a synthetic trace consisting only of large and compli-

cated jobs shows that Graphene is significantly more effective in more complicated synthetic workloads.

3.2 Background

Graphene itself separates into two parts: The online scheduler and the offline scheduler. While both are important, the offline scheduler holds the core part of Graphene. The input into Graphene is a job DAG. A job DAG is a directed acyclical graph that dictates which tasks cannot run before another. A parent task prevents child tasks from being able to run, thus forming a dependency.

3.3 Offline Graphene

The offline portion of Graphene focuses on turning a job DAG into an optimal scheduling order of jobs. In theory, this will improve the utilization of the cluster by reducing suboptimal task requests. For example, if a given job has TaskA and TaskB where TaskB depends on TaskA, Graphene will output an ordering that puts TaskA first. This prevents the scheduler from attempting to run TaskB before TaskA, because the dependency disallows it. A more realistic example is if a task depends on two other tasks with differing runtimes. Graphene prioritizes longer predecessor tasks due to them being considered more troublesome.

To determine the order of tasks, Graphene runs through the following steps:

1. Given a job DAG, identify the troublesome tasks.

2. Fill in a resource space with only troublesome tasks while respecting dependency order.

3. Put every other task into the resource space while respecting dependency order.

4. Repeat steps 1 to 3 with different sets of troublesome tasks until the most optimal ordering that reduces the makespan

Graphene uses a heuristic to define troublesome tasks. For every task in a job DAG, it calculates two metrics: Longscore and Fragscore. Graphene calculates Longscore as the duration of a task divided by the maximum duration over all tasks. Intuitively, a higher Longscore means the task consists more of the job's runtime. It also calculates Fragscore as the total work of the stage divided by the runtime of the stage, where a stage is a set of tasks that are siblings or that have shared dependencies. For example, a stage of tasks in a MapReduce is either the set of tasks in a map or a reduce. The total work of a task equals the resource utilization multiplied by the duration normalized by the cluster share. As there can be multiple resources, Graphene uses the resource that returns the maximum amount of work. Therefore, for a set of tasks, the total work is the sum of the dot product of task resources by task duration divided by the number of total resources used, where the maximal resource is used. Essentially, Fragscore represents the packability of a set of tasks. The higher the Fragscore, the more parallel the stage. When a stage has a runtime of 1 and a total work of 2, this means while an infinite amount of resources could run the entire stage in 1 second, one unit of work is only able to run half of the stage. Conversely, if a stage has a runtime of 2 and a total work of 1, this means while it only needs 2 seconds to run the stage, twice the amount of resources are necessary. In practice, both Fragscore and Longscore are between 0 and 1. Therefore, Graphene iterates through values between 0 and 1 to use as a threshold to consider when a task is troublesome.

Before placing in a resource space, Graphene performs a transitive closure on the troublesome tasks. This ensures that the resulting set includes any task that falls in between two troublesome tasks. During placing, tasks must respect dependency order as dictated by the job DAG while packing as compactly as necessary. The resource space can be thought of as a timeline of all resources. When placing a task, to reduce total runtime, Graphene places the task as close to its parent as possible. This placement for troublesome tasks is only done once, resulting in only one of the many valid orders being chosen.

Graphene places the rest of the tasks carefully to avoid dead-ends. A dead-end is when Graphene cannot place a task in the resource space due to dependency constraints. For example, if we first place the parent and child of some task, another task that shares the same dependencies may not be able to be placed afterward. This is because we must place tasks compactly to reduce runtime and because we must finish running all dependencies beforehand. To deal with this problem, Graphene first splits the rest of the tasks into subsets: P, C, and S, which represent parents, children, and siblings of the troublesome tasks respectively. After this split, there are only 4 resulting possible subset orders of placements. With T as the troublesome set and P, C, and S as the parents, children, and siblings respectively: T→S→C→P, T→S→P→C, T→P→S→C, and T→C→S→P.

Graphene tries all 4 orders and calculate the completion time in each resource space. The order that has the lowest completion time is the final output.

3.4 Online Graphene

The online portion of Graphene takes the offline output to schedule in a real cluster. This section is simpler and only consists of matching a task to a machine.

To determine the optimal machine, each task has a calculated packing score and overbooking score. Graphene calculates a performance score by multiplying the rank of the task in the offline job scheduled (in decreasing order and normalized by the length of ordering) by the maximum of the packing and overbooking score, subtracted by the remaining amount of work in the job. This score prioritizes the task earliest in the schedule, the task with the best fit in a machine, and the job with less work left. Graphene also contains a deficit counter to enforce fairness. If a job's deficit counter goes beyond a set threshold, Graphene forces the next scheduled task to be from that job.

3.5 Implementation

As there is no published source code for Graphene, the following section details the modifications and omissions needed during this attempted implementation. Source code is located at `https://github.com/stxue1/graphene`.

For offline Graphene, we can convert a good portion of the provided pseudocode without trouble. Computationally, the offline portion is the most limiting. While the original Graphene paper recommends splitting the DAGs prior to processing, the authors did not actually implement this. Therefore, as it is not a core part of Graphene, it is omitted from this report's implementation.

One of the computationally most intensive tasks in offline Graphene is the computation of the transitive closure. Built-in methods such as networkx's all_simple_paths cannot feasibly handle DAGs with thousands of instances. To get around this, we define the transitive closure of a set of nodes as the intersection between two sets. The first set includes the starting set and all the successors reachable with a breadth first search from all nodes in the starting set. The second set is calculated the same as the first but finds all predecessors reachable with a breadth first search.

To handle placing tasks in the resource space, we also define a custom data structure that tracks the latest availability of any resource. This enables the placement of tasks to happen in O(logn) time and faster.

The placement of tasks in Graphene can happen in two ways. We either place the parent first or place the child first. Graphene calls these forward and backward placements respectively. Intuitively, a forward placement is the inverse of a backward placement. Programmatically, a backward placement is analogous to a forward placement, except the resource space is inverted before and after the placement. We define a space inversion as the reverse order of all placed tasks. We also use the same placement of subset orders as specified in the original paper.

An immediate issue in task placement is dependencies while placing troublesome tasks. Some troublesome tasks may have dependencies that are not troublesome. Prior to any placement, we have not yet placed these dependencies. Graphene does not specify how to handle this case. Intuitively, there are two solutions: Either only independent troublesome tasks are placed, or all are placed by disregarding dependencies. As the former drastically limits the

search space of the troublesome task set, we implemented the latter. This results in some further modifications to the placement of subsets. For example, placing siblings of the troublesome set directly after can also lead to dependency issues. Therefore, we define the new TrySubsetOrders function in Figure 3.1.

```
return min(
    PlaceTasksF(subset_C, PlaceTasksB(subset_P, PlaceTasks(subset_S,
        ↪ disregard_dependencies=True))),
    PlaceTasksB(subset_P, PlaceTasksF(subset_C, PlaceTasks(subset_S,
        ↪ disregard_dependencies=True))),
    PlaceTasksB(subset_P, PlaceTasksB(subset_S, PlaceTasksF(subset_C),
        ↪ disregard_dependencies=True)),
    PlaceTasksF(subset_C, PlaceTasksF(subset_S, PlaceTasksB(subset_P),
        ↪ disregard_dependencies=True))
)
```

Figure 3.1: Pseudocode for our modified TrySubsetOrders in our Graphene implementation.

The functions PlaceTasksF, PlaceTasksB, and PlaceTasks take a subset of tasks and a resource space, returning a new space. PlaceTasksF and PlaceTasksB perform the forward and backward placements respectively, while PlaceTasks performs both and chooses the most optimal placement. The subsets P, C, and S as defined in 3.3 are subset_P, subset_C, and subset_S respectively. A flag called disregard_dependencies controls whether or not the modified

placement algorithm will be used. When this is enabled, the placement algorithm essentially considers each task independent. An example run is defined as below:

1. Prior to TrySubsetOrders, place troublesome tasks T in an empty resource space while disregarding dependencies.

2. Try subset orders:

   (a) Try T→S→P→C. Place sibling tasks S while disregarding dependencies. Take the best between a forward and backward placement.

   (b) Place parent tasks P with a backwards placement.

   (c) Place parent tasks C with a forwards placement.

   (d) Continue with rest of subset orders T→S→C→P, T→P→S→C, and T→C→S→P.

3. Return the resource space with the lowest completion time out of all subset orders.

The above does not break the no dead-ends invariant by enforcing a proven working order.

For online Graphene, a significant amount of pseudocode is left unexplained. The heuristics for overbooking scores and deficit counters are poorly defined. Therefore, we ignored overbooking and only support slot fairness. However, we did not use slot fairness for the experiments below.

3.6 Experiment

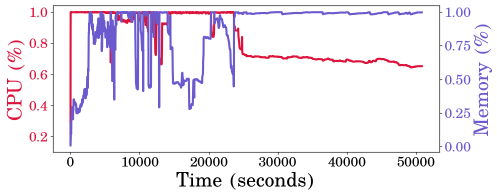
Rather than a heartbeat-based scheduler similar to the original Graphene implementation, this report's scheduler schedules a task at the earliest possible time. Additionally, the original Graphene experiment ran on a synthetic trace. As a synthetic trace can vary drastically from a real-world trace, we use the Alibaba batch job trace for this experiment.

The original experiment also compared Graphene to different types of schedulers. As Tetris and Critical Path schedulers are not available, we used FIFO instead. Here, we implement FIFO to enforce the order of jobs; if the queue is full but there is no space for the first job in the queue, FIFO waits until there is space. Thus, FIFO represents a perfectly fair scheduler while Graphene theoretically represents a fully greedy scheduler.

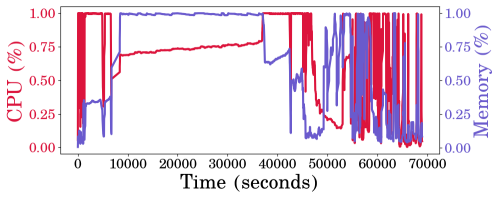
To extract the data from the Alibaba trace, we generate DAGs from real-time instance data. In the DAG, each node represents an instance of a task, and a dependency between two nodes indicates a task dependency represented instance-wise. As Graphene is dependent on annotated data for CPU, memory, and duration estimates, we use the real average CPU, average memory, and completion time from the Alibaba trace. Additionally, we only consider DAG jobs, and we do not include singular tasks in this experiment. We give the scheduler one machine out of the original Alibaba cluster, and jobs arrive every 5 seconds.

3.7 Result

As shown in Figure 3.2, Graphene has a decent improvement over FIFO when it comes to completion time. This provides evidence that Graphene works as intended by im-



(a) Graphene utilization over time.



(b) FIFO utilization over time.

Figure 3.2: Utilization across different schedulers with a trace created from the first 30 jobs from the Alibaba trace.

plementing a near-optimal improvement; CPU utilization is completely used up, leaving little room for improvement.

We noticed that our scheduler reduced execution time by approximately 30% when going from FIFO to Graphene. This is mainly due to the underutilized gaps seen with FIFO. Graphene does not have these gaps thanks to being a greedy scheduler. Additionally, since the simulation pins CPU utilization, this indicates that the lack of CPU cores instead of memory is what starves DAG jobs from the Alibaba trace, contrary to what the original Alibaba paper claims. For Alibaba, this indicates that singular-task jobs and online service jobs use most of the memory.

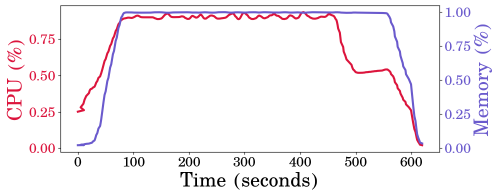
In the original Graphene experiment, they observed a modest increase in average utilization and not an absurd increase of one metric to 100%. This is likely due to the fully

greedy nature of this Graphene implementation. Additionally, an observation of the inputted trace is that the average job DAG size is small; most job DAGs consist of a couple dozen instances while a few go to the thousands. Large DAG jobs are harder to schedule and, as shown in the previous section, are often prone to queueing delay. Conversely, small jobs request fewer resources and thus are easier to schedule, making it easier to fully utilize a resource. As a result, this trace shows how real-world clusters do not strictly consist of hard-to-schedule jobs. Instead, such traces are typically full of smaller and easier-to-schedule jobs.

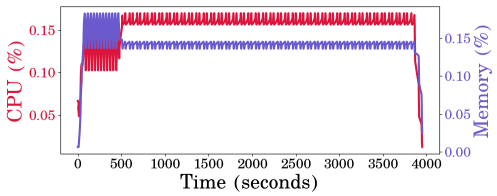
As the authors of Graphene defined it to handle large and hard-to-schedule jobs, the nature of real-world job traces reduces its real-world efficacy. If fairness is key, FIFO has less of an impact than expected. One way to see this effect is through the inverse; if the trace is full of large and complicated jobs, FIFO has more impact on the runtime, as shown in Figure 3.3. Here, on large jobs, the runtime is more than quadrupled and the utilization goes drastically down. Thus, while the performance between Graphene and FIFO is still significant, the nature of real-world clusters tempers the improvements and performance gains, and the usual sacrifice between fairness and efficiency may not have as large of a gap as synthetic benchmarks imply.

3.8 Conclusion

With job DAGs becoming a popular method to represent workloads, modern clusters face increasingly large performance-related challenges. One of the common metrics of concern is queueing delay. From this analysis, we showed a significant correlation between the size of a job DAG and how many seconds the job experienced queuing delay. Interestingly, despite



(a) Graphene utilization over time on a trace strictly containing large jobs.



(b) FIFO utilization over time on a trace strictly containing large jobs.

Figure 3.3: Utilization across FIFO and Graphene on a trace strictly containing large jobs. The trace consists of 20 duplicates of a ~7000 instance job.

existing analyses outlining how the cluster is consistently starved for memory, there is no correlation between the queueing delay and the remaining memory in the cluster. Similarly, there is no correlation between metrics such as oversubscription and queueing delay, despite other modern clusters using an oversubscription factor as a configurable option for cluster performance. However, this may be because this analysis does not explore the impact of co-located online service jobs, which future studies may address. Fortunately, queueing delay on the cluster is primarily impacted by those tasks lying on the critical path. This implies that a critical path-based scheduler may be a viable option to reduce queueing delay.

As an optimal scheduling algorithm for job DAGs, Graphene is shown to be effective in improving overall completion time compared to a fully fair scheduler such as FIFO. However, when applying real-world traces in simulations, we saw significantly less speedup than

expected. In contrast, when creating a synthetic trace consisting only of complicated DAG jobs, we saw a speedup of over 6 times. This implies that synthetic benchmarks for job schedulers are overly optimistic in performance gains and that real-world situations result in perfect schedulers not having as much of an impact as originally thought. If a user does not need all the performance out of a scheduler, a fully fair and basic scheduler such as FIFO may be enough.

Bibliography

[1] Robert Grandl et al. “Graphene: packing and dependency-aware scheduling for data-parallel clusters”. In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI’16. Savannah, GA, USA: USENIX Association, 2016, pp. 81–97. ISBN: 9781931971331.

[2] Jing Guo et al. “Who limits the resource efficiency of my datacenter: an analysis of Alibaba datacenter traces”. In: *Proceedings of the International Symposium on Quality of Service*. IWQoS ’19. Phoenix, Arizona: Association for Computing Machinery, 2019. ISBN: 9781450367783. DOI: 10.1145/3326285.3329074. URL: https://doi.org/10.1145/3326285.3329074.

[3] Abhishek Verma et al. “Large-scale cluster management at Google with Borg”. In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys ’15. Bordeaux, France: Association for Computing Machinery, 2015. ISBN: 9781450332385. DOI: 10.1145/2741948.2741964. URL: https://doi.org/10.1145/2741948.2741964.