EFFICIENT EXECUTION OF SCIENTIFIC WORKFLOWS ON BATCH-SCHEDULED CLUSTERS

A THESIS SUBMITTED TO THE GRADUATE DIVISION OF THE UNIVERSITY OF HAWAI'I AT MĀNOA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

MAY 2020

By
Evan Hataishi

Thesis Committee:

Henri Casanova, Chairperson
Edoardo Biagioni
Philip Johnson

Keywords: Batch Scheduling, Scientific Workflow Scheduling, Task Clustering

Copyright © 2020 by
Evan Hataishi

To my grandparents, parents, and sisters.

ACKNOWLEDGMENTS

I want to first thank my committee members, Edoardo Biagioni and Philip Johnson. Both of whom I was lucky enough to work with personally and learn from at some point over the years.

I especially want to give a heartfelt thank you to my chairperson and advisor, Henri Casanova, for all his guidance and support throughout my graduate studies. I truly could not have asked for a better advisor and mentor.

I would also like to thank David Bachmann, my mentor from when I was a young intern at IBM in the hottest place on this planet, Austin, Texas. I particularly enjoyed all of our conversations about how to become an awesome engineer and whether or not I should go to graduate school.

Last but not least, I would like to thank all the other faculty and graduate students I got to meet and befriend during my brief stint in academia. Thank you for all the fun and enlightening conversations.

ABSTRACT

Scientific workflows are ubiquitous and key applications in numerous scientific domains. These applications often have high computational demands and must be executed on High Performance Computing (HPC) platforms. Most production HPC platforms are managed by batch schedulers that turn out to be poorly suited to workflows that comprise many dependent tasks. An interesting question is thus: how can workflows be executed efficiently on batch-scheduled HPC platforms. Previous work has addressed this question at the resource management level and at the application level, where both kinds of solutions have their own drawbacks and merits. This thesis proposes an application-level algorithm that partitions a workflow into a chain of jobs that are submitted in sequence to the batch scheduler. The novelty is that these jobs are constructed and submitted so as to explicitly minimize workflow makespan, i.e., overall wall clock time. This is feasible because production batch scheduler implementations provide queue wait time estimates (albeit not necessarily accurate). The proposed algorithm is evaluated through simulation, based on production batch workloads and workflow configurations, and compared to both baseline algorithms and a recent algorithm proposed by others. Evaluation results show that, in general, our algorithm performs well against competing algorithms due to the way in which it partitions a workflow into clustered jobs. Furthermore, we find that performance improvements of well over 30% are achievable over those previously proposed algorithms in many cases. However, our algorithm relies heavily on wait time estimates to make clustering decisions. Thus, we also find that our algorithm can fare poorly on platforms in which the batch scheduler provides very inaccurate wait time estimates (e.g., due to platform users providing wildly inaccurate job execution time estimates).

TABLE OF CONTENTS

Acknowledgments . . . . **iv**

Abstract . . . . **v**

List of Tables . . . . **viii**

List of Figures . . . . **ix**

1 Introduction **1**

1.1 Context and Motivation 1

1.2 Contributions 1

1.3 Roadmap 2

2 Background and Problem Statement **4**

2.1 Scientific Workflow Applications 4

2.2 HPC Clusters and Batch Scheduling 6

2.2.1 Backfilling 7

2.3 Problem Statement 8

3 Related Work **10**

3.1 RJMS-level Approach 10

3.2 Application-level Approach 10

4 Methodology **12**

4.1 Simulator 12

4.2 Workflow Configurations 12

4.3 Batch Workloads 13

5 The Algorithm by Zhang et al. **15**

5.1 Overview . . . 15
5.2 Evaluation Results . . . 17
5.3 Improving ZHANG . . . 20
6 The GLUME Algorithm . . . 23
6.1 Intuition and Overview . . . 23
6.2 Pseudocode . . . 24
7 Evaluation Results . . . 28
7.1 Results for $*$-$*$-medium Workflow Configurations . . . 28
7.1.1 Results with Accurate Job Durations . . . 28
7.1.2 Results with Realistic Job Durations . . . 28
7.2 Overview of All Results . . . 30
7.3 Conclusion . . . 31
8 Conclusion . . . 33
8.1 Summary of Findings and Contributions . . . 33
8.2 Future Work . . . 33
A Additional Figures for Chapter 5 . . . 35
B Additional Figures for Chapter 6 . . . 36
Bibliography . . . 38

LIST OF TABLES

7.1 Comparison of GLUME and Zhang for both the KTH and SDSC workloads, assuming either accurate requested job durations (left side) or real-world requested job durations (right side) for $*$-$*$-short, $*$-$*$-medium, and $*$-$*$-long workflow configurations. Each cell is formatted as x/y, where x, resp. y, is the number of workflow configurations for which GLUME, resp. Zhang, beats Zhang, resp. GLUME. Results in bold are those for which $x \geq y$ (i.e., GLUME beats Zhang). 31

LIST OF FIGURES

2.1 Sample Montage workflow (reproduced from [1]) 5

5.1 Workflow execution as 4 batch jobs using Zhang. 17

5.2 Average percentage improvement relative to Zhang for ∗-250-∗ workflows on the KTH workload, with maximum number of simultaneously running workflow jobs at 128. 18

5.3 Average percentage improvement relative to Zhang for ∗-250-∗ workflows on the KTH workload, with maximum number of simultaneously running workflow jobs at 16. 19

6.1 Example execution of the J_i and J'_i jobs by GLUME. 23

7.1 Average percentage improvement relative to Zhang for ∗-∗-medium workflow configurations on the KTH workload, with maximum number of simultaneously running workflow jobs at 16 and accurate job requested durations. 29

7.2 Average percentage improvement relative to Zhang for ∗-∗-medium workflow configurations on the SDSC workload, with maximum number of simultaneously running workflow jobs at 16 and accurate job requested durations. 30

7.3 Average percentage improvement relative to Zhang for ∗-∗-medium workflow configurations on the KTH workload, with maximum number of simultaneously running workflow jobs at 16 and realistic job requested durations. 31

7.4 Average percentage improvement relative to Zhang for ∗-∗-medium workflow configurations on the SDSC workload, with maximum number of simultaneously running workflow jobs at 16 and realistic job requested durations. 32

A.1 Average percentage improvement relative to Zhang for ∗-∗-medium workflow configurations on the SDSC workload, with maximum number of simultaneously running workflow jobs at 16 (left-hand side) and 128 (right-hand side) and accurate job requested durations. 35

B.1 Average percentage improvement relative to Zhang for ∗-∗-short workflow configurations on the KTH (left-hand side) and SDSC (right-hand side) workloads, with maximum number of simultaneously running workflow jobs at 16 and accurate job requested durations. 36

B.2 Average percentage improvement relative to Zhang for ∗-∗-long workflow configurations on the KTH (left-hand side) and SDSC (right-hand side) workloads, with maximum number of simultaneously running workflow jobs at 16 and accurate job requested durations. 36

B.3 Average percentage improvement relative to Zhang for ∗-∗-short workflow configurations on the KTH (left-hand side) and SDSC (right-hand side) workloads, with maximum number of simultaneously running workflow jobs at 16 and realistic job requested durations. 37

B.4 Average percentage improvement relative to Zhang for ∗-∗-long workflow configurations on the KTH (left-hand side) and SDSC (right-hand side) workloads, with maximum number of simultaneously running workflow jobs at 16 and realistic job requested durations. 37

CHAPTER 1
INTRODUCTION

1.1 Context and Motivation

Scientific workflow applications have become mainstream in support of research and development activities in numerous scientific domains [55]. Many such applications have high computational demands, mandating the use of High Performance Computing (HPC) platforms such as commodity clusters. Most production HPC platforms run Resource and Job Management Software (RJMS) that implements some form of batch scheduling [62, 42, 48, 41, 46]: job execution requests are first queued and then eventually granted (exclusive) access to hardware resources. A job thus experiences a period of *wait time* followed by a period of *execution time*. RJMS can be configured to enforce various job scheduling and resource management policies. Regardless of the specifics, this resource management approach is geared to traditional parallel computing workloads in which users submit moderate numbers of potentially large, long-running, and, at most, loosely dependent parallel jobs. It is therefore poorly suited for large workflow applications that can comprise thousands of dependent tasks. A key question arises: *how can workflows be executed efficiently on batch-scheduled HPC platforms?* In this thesis we focus on minimizing workflow *makespan*, i.e., the time elapsed between the submission of the workflow's first task to the batch scheduler and the completion of the last task. The makespan thus comprises both (queue) wait time and execution time.

The above question can be attacked at either the RJMS level or the application level. At the RJMS level, one can implement new RJMS that is specifically designed to support workflows [3], enhance current batch-scheduling RJMS to become workflow-aware [50], or insert an extra workflow-aware RJMS layer on top of native RJMS [30]. At the application level, one must partition a workflow application into dependent jobs and submit these jobs judiciously to a non-workflow-aware RJMS [57, 63, 54, 64]. Both kinds of solutions have drawbacks. RJMS-level solutions face adoption challenges: The job scheduling literature is rife with promising ideas, but few have impacted production RJMS (although there are exceptions [46]). Conversely, application-level solutions are impeded by constraints imposed on job executions by batch schedulers and by non-deterministic wait times. As a result, the problem of deciding how to partition a workflow into jobs and when to submit these jobs is non-trivial. Poor solutions to this problem can translate to vastly sub-optimal workflow makespans.

1.2 Contributions

In this thesis we propose an application-level solution, which requires answering two main questions. The first is: How should a workflow be partitioned into (dependent) jobs? At one extreme is a

one-job-per-task approach. This approach is problematic for deep workflows and/or workflows with long tasks due to cascading wait times, but also because resource management policies typically throttle the number of jobs that a single user can run at a time. The other extreme is a workflow-as-a-single-job approach. This approach is also problematic, for it can suffer from high wait times and high compute resource waste. The second question is: When should subsequent jobs be submitted so as to overlap wait time and execution time? Too eager a scheme will cause job re-executions due to expiration of time allocations, but too lazy a scheme will lengthen workflow makespan.

The work in [64] proposes an application-level solution that answers the two aforementioned questions with heuristics. The main idea is to partition the workflow into sets of consecutive levels, and to submit each set to the batch scheduler as a job, striving to overlap the job's execution time with the next job's wait time. The partitioning of the workflow levels into jobs is such that each job's wait time is not greater than some factor of its execution time. Job wait times are predicted based on estimations provided by RJMS. In this thesis, we propose another approach that also partitions the workflow into sets of contiguous levels.

The contributions of this thesis include:

- We identify areas of improvement for the algorithm proposed in [64] and experimentally evaluate the impact of these improvements.
- The authors in [64] did not compare their approach to a workflow-as-a-single-job approach where the job specifications are informed by wait time predictions. We find that this simple competitor can vastly outperform the algorithm in [64] as well as our enhanced version thereof.
- The above finding raises the question of whether partitioning a workflow into sets of contiguous levels is worthwhile in practice. We answer this question by proposing a new algorithm, GLUME. The key difference with the algorithm in [64] is that GLUME explicitly attempts to minimize the workflow makespan.
- Using simulation for production workloads and workflow configurations we find that GLUME compares favorably to its competitors. However, for workloads in which queue wait time predictions are very inaccurate, GLUME can be outperformed by a naive one-job-per-task approach.

1.3 Roadmap

This thesis is organized as follows:

- Chapter 2 provides necessary background for this thesis and defines our target workflow scheduling problem;
- Chapter 3 reviews relevant related work;
- Chapter 4 details our experimental methodology;

- Chapter 5 describes and gives experimental results for the algorithm in [64], as well as for proposed improvements thereof;
- Chapter 6 presents our proposed algorithm and relevant pseudocode;
- Chapter 7 details and fives evaluation results for our proposed algorithm;
- Chapter 8 concludes this thesis with a summary of our results and directions for future work.

CHAPTER 2
BACKGROUND AND PROBLEM STATEMENT

In this chapter we first provide necessary background for this thesis. Specifically, we review the class of applications we target, *scientific workflows*, and the class of compute platforms on which we wish to execute these applications, *batch-scheduled clusters*. Then, we provide the *problem statement* of this thesis, detailing our assumptions and our objectives.

2.1 Scientific Workflow Applications

Many scientists today need to perform enormous amounts of computations on enormous amounts of data. Therefore, they must execute their data-processing applications on high-performance compute platforms. To do so, they need convenient and powerful abstractions for expressing the data and computational needs of their applications. One such popular abstraction is *Scientific Workflows*, or "Workflows." Workflows allow scientists to express multi-step computational processes that can include, for instance, extracting data from databases, transforming the data, running analyses on it, post-processing the analysis results, and archiving these results into other databases. As a result, workflows have become mainstream for conducting large-scale scientific research [55]. Astronomers are using workflows to generate science-grade mosaics of the sky [12, 10, 43] and to search for extra-solar planets using data collected by NASA's Kepler mission [11, 58]. The Laser Interferometer Gravitational-Wave Observatory (LIGO) uses workflows to search for binary inspiral gravitational waves [15]. Earthquake scientists use workflows to develop shake maps of Southern California [17, 16, 18]. Researchers in bioinformatics have embraced workflows for protein folding [23], DNA and RNA sequencing [14, 33, 37], and disease-related research [29, 31].

Most workflows are structured as Directed (usually) Acyclic Graphs (DAGs). DAG vertices represent compute tasks, and DAG edges represent dependencies between the compute tasks. Therefore, a task may not begin to execute until all its predecessor tasks have completed. The most common example of a dependency is when one task produces an output file that another task uses as input. In general, a workflow representation allows for parallelization and scaling of scientific applications in a view to executing them on large-scale compute platforms. A famous example is the Montage workflow created by the NASA/IPAC Infrared Science Archive to generate custom mosaics of the sky by stitching together multiple input images. During the production of the final mosaic, the geometry of the output is calculated from the geometry of the input images. The inputs are re-projected to be of the same spatial scale and rotation. The background emissions in the images are then corrected to be of the same level in all images. The re-projected, corrected images are co-added to form the final mosaic. Figure 2.1 shows the structure of a typical montage workflow. The workflow comprises 9 different types of task (each represented by a different color in the figure). Task dependencies are shown via directed edges, and the dependency structure exhibits

some parallelism (e.g., the four green tasks can be executed concurrently).

Workflows in different application domains may have completely different structures and specifications (e.g., number of tasks, compute- or data-intensive, varying amounts of parallelism, patterns of task dependencies). Workflow researchers have attempted to characterize typical workflow configurations [13, 35] and tools have been produced that can generate synthetic, but representative, workflow specifications for various scientific domains [28].

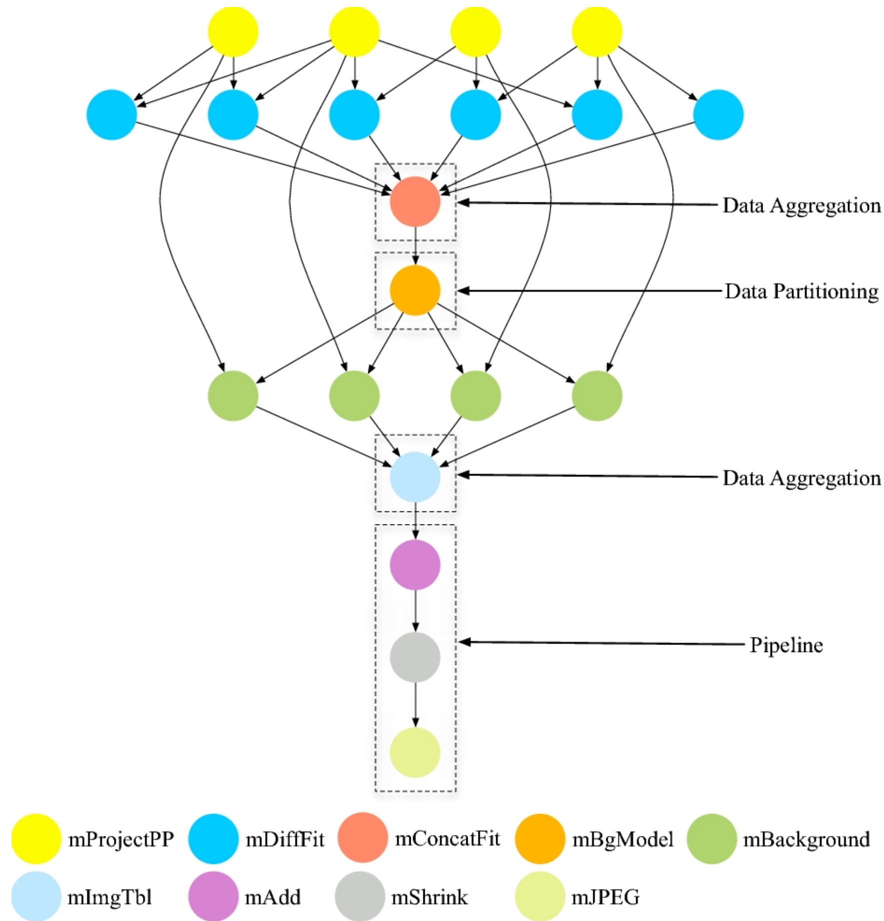

Figure 2.1: Sample Montage workflow (reproduced from [1])

Given the now mainstream use of workflows in science, it is crucial that workflows can be executed efficiently on available compute infrastructure, including large-scale parallel and distributed computing platforms. However, scientists should not have to be concerned with the details and the logistics of these executions. (After all, computation is only a means to an end for scientific research.) As a result, software tools known as Workflow Management Systems (WMSs) have been developed [4, 5, 24, 25, 27, 59, 60] that allow scientists to: (*i*) describe their applications as high-level abstractions decoupled from the specifics of workflow execution; and (*ii*) execute these applications seamlessly on possibly complex, multi-site distributed platforms that can accommo-

date large-scale executions. The work in this thesis provides an algorithm that ultimately is to be integrated as part of such WMSs.

2.2 HPC Clusters and Batch Scheduling

Given the computational demands of scientific applications and, in particular, of scientific workflows, scientists need access to HPC (High Performance Computing) platforms. The most popular such platforms are large *commodity clusters*. According to the November 2019 Top500 list of the 500 fasted supercomputers in the world [56], 91.60% of them are commodity clusters. A commodity cluster is a set of independent commodity computers, or *compute nodes*, connected with a high-bandwidth network. Clusters often comprise from hundreds to tens of thousands of nodes. They are expensive to build and operate (due to power consumption, a large part of which is due to cooling needs). As a result, they are usually run by an organization or institution. A cluster can then be managed by a single entity while allowing compute resources to be shared among many users. Because users compete for use of the cluster's compute nodes, software mechanisms are needed to control the access and usage of those compute resources. These mechanisms are typically implemented as part of a software infrastructure called a RJMS (Resource and Job Management System). Users must interact with their platform's RJMS to request access to compute resources by submitting *jobs* to it. The vast majority of RJMS deployed on HPC clusters implement a form of *batch scheduling* to handle these requests, with the objectives of keeping users happy and the platform as utilized as possible. We refer to these RMJSs as *batch schedulers*.

Originally, the term "batch" processing referred to punch cards being processed in a batch. While batch processing has certainly evolved since the era of the IBM Mainframes, batch schedulers for HPC clusters still rely on similar methods for scheduling user jobs. Batch schedulers use a *batch queue* in to hold pending job requests for compute resources, ordered by time of arrival. A job request consists of: (i) a number of required compute nodes; and (ii) a duration for usage. For example, a user may submit a job that requests 5 compute nodes for 10 consecutive hours. Job requests are placed in the batch queue until they are picked for execution. A scheduling algorithm decides when a job is picked from the queue. Once picked, the job is granted exclusive access to its requested compute resources for the requested duration. Therefore, a job's execution consists of the time spent in the batch queue, i.e., the *wait time*, plus the time spent using the compute resources, i.e., the *run time*. The combination of wait time and run time is referred to as a the job's *makespan*.

Across different platforms batch schedulers will have unique configurations so as to implement different scheduling behaviors. Although myriads of configurations exist, most batch schedulers implement a version of a First Come First Served (FCFS) algorithm to pick jobs from the batch queue. While the FCFS algorithm is fair and predictable, it causes the platform to suffer from low system utilization due to fragmentation and high wait times for jobs. With a FCFS scheduler,

jobs that request more compute resources than available at the time of request will bottleneck the batch queue and result in idle compute resources. As an extreme example, suppose a platform has 49 currently available nodes, but the job at the front of the batch queue, i.e., the oldest pending job, requires 50 nodes. If another node does not become available for 10 hours, those 49 nodes will remain idle for the 10 hours, and all other jobs in the queue can not be picked for execution, even if some jobs are small and could complete within the 10 hours. Furthermore, when a job does not complete but has used up its requested execution time, it is immediately killed by the RJMS, and its compute resources are freed. To avoid their job being killed prematurely, users typically request more execution time than they expect their job to use [39]. As a result, most jobs complete well ahead of their requested duration, and compute resources become free ahead of time. This creates unexpected "holes" in the schedule, which further increases the occurrence of idle compute resources.

2.2.1 Backfilling

To remedy the problems described above (namely high job wait times and idle compute resources), batch schedulers implement a feature called *backfilling*. There are two distinct versions of backfilling. The first, *aggressive backfilling*, was developed as part of the EASY (Extensible Argonne Scheduling system) batch scheduler [44]. Aggressive backfilling allows small jobs to move ahead in the queue as long as they do not delay the first job in the queue. Allowing smaller jobs to jump ahead solves the problem of low utilization but can lead to unbounded delays for all other jobs in the queue, for only the job at the front of the queue is guaranteed to run without delay. This can be bad for users with large jobs at the back of the queue. The number of jobs that cannot be delayed due to a backfill operation is sometimes termed the "reservation depth" [3]. The EASY scheduler only allows backfilling if the first job will not be delayed, so its reservation depth is 1. However, some RJMS allow the reservation depth to be configured arbitrarily by a system administrator [3]. If the reservation depth is set to infinity, meaning all jobs in the batch queue, we get the other version of backfilling: *conservative backfilling*. With conservative backfilling, small jobs are backfilled only if they do not delay the execution of *any* job in the queue. Thus, conservative backfilling has the same benefits of EASY backfilling without suffering from unbounded delays [44].

Wait time estimates are valuable to platform users, for they allow users to predict how different job configurations could exhibit different wait times. Suppose a user has a job with 10 tasks, and each task must run for 1 hour each. They could submit a job that requests 10 nodes for 1 hour or a job that requests 1 node for 10 hours. Depending on the state of the batch queue and of the currently running jobs, the wait time for these two possible job configurations could be drastically different. If the user knows these wait times, then they could pick the best job configuration, i.e., the one that would lead to the lowest job makespan. Researchers have studied the "batch queue wait time prediction problem". Although some results have been obtained [45, 34, 38] and some

prediction services have been deployed on particular systems (e.g., Karnak on XSEDE [36]), queue wait time prediction remains challenging.

Instead of full-fledged wait times predictions, production batch schedulers provide wait time *estimates*. These estimates are based on the current state of the queue and the currently running jobs. Importantly, these estimates are computed assuming that job durations are exactly as requested when they were submitted. Due to this assumption, wait time estimates are typically not 100% accurate. This is because of premature job completions due to users making pessimistic time requests, as discussed earlier in this section. The planned schedule is thus constantly perturbed, with jobs completing unexpectedly (from the perspective of the batch scheduler) early. These early completions create backfilling opportunities, i.e., some jobs can be started earlier than planned, thus making wait time predictions for these jobs pessimistic. Another source of estimate inaccuracy is that, depending on the manner in which backfilling is applied, future job arrivals may increase wait times of pending jobs. Some batch schedulers use *aggressive backfilling* [40], by which backfilling one job can delay the start time of other jobs. As a result, wait time predictions for these other jobs are optimistic, and, in fact, arbitrarily so since wait times are unbounded. By fixing the problem of unbounded delays, conservative backfilling gives batch schedulers the ability to provide more accurate wait time estimates. Because a job may only be backfilled if doing so does not delay any other job in the queue, conservative backfilling guarantees that backfilling a job will not delay the start time of any pending job. This leads to fairer schedules, possibly at the expense of resource utilization, but also makes wait time estimates more accurate. Several popular production batch scheduler implementations allow for conservative backfilling, such as OAR [46] and Slurm [62].

2.3 Problem Statement

We consider a cluster of homogeneous compute nodes with some globally accessible shared file system. Access to the cluster is managed by a RJMS that implements batch scheduling, i.e., a *batch scheduler*, and provides wait time estimates.

On the cluster we wish to execute a workflow application that consists of tasks with dependencies. Each task executes on a single compute node, and for each task we have an accurate estimate of its execution time, which includes computation and I/O time (i.e., reading and writing input and output files to the shared file systems). This is a common assumption in the literature, justified in production settings in which the same workflow applications are executed repeatedly, and previous executions can serve as benchmarks for future executions. The work in [57] questions this assumption and proposes an on-line approach to discover task resource demands as the workflow is running, which could also be done in this thesis (and in most previously proposed workflow scheduling approaches).

The objective is to minimize workflow *makespan*, i.e., the time between the first task submission (as part of a job) to the batch scheduler and the last task completion. To minimize makespan one

must decide how to group workflow tasks into jobs, how many compute nodes should be requested, how much execution time each such job should request, and when to submit these jobs to the batch scheduler. We do not consider the use of advance reservation mechanisms [53]. Using advance reservations would still require to group workflow tasks in individual jobs (so as to be able to start the execution reasonably early), but would not benefit from backfilling opportunities.

The main contribution of this thesis is an algorithm for achieving the above objective, which we we evaluate and compare to competing algorithms experimentally in simulation.

CHAPTER 3
RELATED WORK

The question of scheduling workflow applications has received a large amount of attention. In this chapter we discuss previous research that has been done in the area of scientific workflow scheduling on compute platforms managed by RJMSs, and, in particular, by batch schedulers. We first review works that have proposed RJMS-level solutions. We then discuss application-level algorithms, which group workflow tasks together and submit those groups to unmodified RJMSs. We also review previously proposed workflow scheduling algorithms that group tasks together, albeit for different platform settings.

3.1 RJMS-level Approach

Some authors have proposed RJMS-level solutions, i.e., RJMS designs and implementations that are well-suited to workloads that comprise, or consist exclusively of, workflow applications. The work in [3] proposes a RJMS that is designed from the ground up for workflows, advocating for a hierarchical approach that can support large-scale workflows. In [50], the authors propose to augment the Slurm RJMS [62] to make it workflow-aware. Although Slurm supports job dependencies it does not attempt to minimize intermediate wait times between dependent jobs. The approach in [50] ensures that tasks in workflow jobs are positioned in adjacent positions in the batch queue, so as to reduce intermediate wait times. Yet another approach is proposed in [30], in which a RJMS layer is inserted between the application and the platform's native batch scheduling RJMS. This layer makes it possible for batch-scheduled resources to be allocated to a workflow application in an elastic manner and uses application checkpointing to mitigate intermediate wait times.

3.2 Application-level Approach

Application-level approaches may not be able to rival some of the RJMS-level solutions discussed in the previous section. However, because they require no modification to the RJMS, they can immediately be used by users on their production HPC platforms with standard RJMS deployments. The workflow scheduling question has been investigated at the application level for several classes of distributed computing platforms, often inspired by algorithms for scheduling task graphs [26]. In this thesis we focus on batch-scheduled clusters, for which several application-level scheduling approaches have been proposed that target workflows. Some authors have proposed to submit each ready workflow task as an individual job. The work in [57] focuses on picking sizes for one-task jobs to avoid resource waste and increase throughput. The work in [63] proposes to use wait time predictions to schedule one-task jobs efficiently in platforms that consist of multiple batch-scheduled clusters. For these same multi-cluster platforms, the authors of [54] evaluate several algorithms

that also schedule workflows using one-task jobs. Given known problems with the one-task-per-job approach in practice with current RJMS implementations [50], other authors have proposed to group workflow tasks together into batch jobs so as to mitigate wait times. The approach in [64], which we detail and evaluate in Chapter 5, groups consecutive workflow levels into individual jobs. Note that application-level approaches, including the one proposed in this thesis, can benefit those RJMS-level solutions that execute workflows one level at a time [50, 30], making it possible to decide how to aggregate consecutive levels so as to reduce workflow makespan.

The idea of grouping workflow tasks together, a.k.a., "task clustering", is not novel. Many task graph scheduling algorithms perform task clustering for reducing network communication overheads [32]. In the context of workflows, authors have proposed clustering workflow tasks together not only to reduce communication overheads, but also to reduce task execution overheads [51], to improve load imbalance [22], and to improve fault-tolerance [21]. In this thesis, like in [64], we cluster tasks together into jobs so as to mitigate wait times on batch-scheduled clusters.

CHAPTER 4
METHODOLOGY

Like most previous works in this area (i.e., application scheduling and batch scheduling), we rely on simulation for experimental evaluations. This is because simulation results are reproducible, thus, making it possible to perform fair comparison of competing algorithms. More specifically, it is possible to quantify how different algorithms would fare when executing a given workflow under the exact same cluster workload conditions. In this chapter we detail our experimental methodology. We first describe the simulator we use for obtaining experimental results. We then detail the workflow configurations and the cluster workloads we use to drive the simulations.

4.1 Simulator

We have implemented a simulator in C++ using the WRENCH framework [19, 61]. WRENCH is built on top of SimGrid [20], which provides basic abstractions for distributed platform simulation. WRENCH also uses Batsched [49, 9], which is a batch scheduler simulator that implements a variety of popular batch-scheduling algorithms used in production systems. We configure Batsched to use conservative backfilling [44] (see Section 2.2.1).

Our simulator implements all the algorithms described hereafter and allows us to simulate arbitrary workloads and workflow configurations. More specifically, our simulator takes as input:

- The application-level workflow scheduling algorithm to employ;
- A workload trace file that defined the "background" load on the cluster throughout time. The workflow execution competes with this background load for access to compute nodes;
- The number of compute nodes in the cluster;
- A workflow description file that gives workflow task execution times and dependencies;
- The date at which the workflow execution begins (i.e., when the first task can be submitted for execution).

The simulator outputs the workflow makespan as well as other metrics of interest (e.g., queue wait times, compute time wasted due to poor parallel efficiency). The source code for our simulator is available at [2].

4.2 Workflow Configurations

Using the generator in [28], we generate workflow configurations that correspond to various application domains. Specifically, we consider workflows for four applications: GENOME (bioinformatics), SIPHT (bioinformatics), CYBERSHAKE (earthquake engineering), and MONTAGE (astronomy). The generator allows us to vary the number of tasks per workflow as well as the task execution times. For each of the above applications we generate workflows with $50, 250, 500 \pm 5$

tasks. The ± 5 is because of constraints imposed by the workflow's structure, making it impossible to generate a (realistic) configuration for every arbitrarily specified number of tasks. We generate these workflow configurations with sequential execution times (i.e., sum of task execution times) of $100, 500, 1000 \pm 3$ hours. The ± 3 is because the generator draws task execution times from random distributions. In total, we generate $4 \times 3 \times 3 = 36$ workflow configurations. We use the notation x-y-z to denote a workflow configuration, where x denotes the workflow as G, S, C, or M (for GENOME, SIPHT, CYBERSHAKE, and MONTAGE, resp.); y denotes the number of tasks as 50, 250, or 500; and z denotes the task duration as short, medium, long (for 100, 500, and 1000 hours, resp.). For instance C-250-short denotes a CYBERSHAKE workflow configuration with $\sim$250 tasks and a sequential execution time of $\sim$100 hours. Whenever applicable we use regular expression notations, e.g., $*$-500-[short—medium] denotes all 500-task workflow configurations with $\sim$100- and $\sim$500-hour sequential execution times.

4.3 Batch Workloads

We simulate background load on the batch-scheduled cluster based on two job submission/execution logs from the Parallel Workloads Archive [47]. The first log, KTH, is from a system with 100 compute nodes, $\sim$28,000 jobs, and utilization of $\sim$70% over an 11-month period. The second log, SDSC, is from a system with 128 compute nodes, $\sim$60,000 jobs, and utilization of $\sim$83% over a 24-month period. All simulations include a one-day "warm-up" period, after which we simulate the workflow execution when submitted on the half-hour until the end of the week that follows the warm-up day, for a total of $(48 \times 6) + 1 = 289$ different submission times.

We simulate the execution of jobs in the background workloads with either accurate or actual job requested execution times. For the former, we set of each job's requested duration to its actual (as seen post-mortem) duration. This is unrealistic, as requested execution times are known to be inaccurate [39], but corresponds to an ideal case for wait time estimates provided by the batch scheduler when there are no premature job completions. For the latter, we use actual requested execution times as reported in the logs. When discussing results we specify which method is used (either "accurate" or "inaccurate" execution time estimates). We include results with the accurate method because they make it possible to compare workflow scheduling algorithms in ideal conditions w.r.t. queue wait time predictions.

HPC facilities typically cap the number of batch jobs that a user can run simultaneously in the system, as supported by production batch scheduler (e.g., by setting the `MaxSubmitJobsPerAccount` limit value in Slurm [52]). Although an exhaustive survey of HPC facilities is beyond the scope of this thesis, we take the Argonne Leadership Computing Facility as an example: the recently decommissioned Mira allowed users to have at most 5 running jobs per batch queue [7]; The Cooley and the Theta systems allow both for up to 10 running jobs [6, 8]. Our inspection of batch scheduling policies at the Oak Ridge Leadership Computing Facility, the National Energy Research Scientific

Computing Center, and the Texas Advanced Computing Center yield similar observations, with most batch queues imposing relatively low caps on per-user numbers of jobs. Since we consider executions with potentially many number of workflow jobs (e.g., for the one-job-per-task approach), we present results with different caps on the per-user number of simultaneous running jobs.

CHAPTER 5
THE ALGORITHM BY ZHANG ET AL.

In this chapter we first provide an overview of the algorithm proposed by Zhang et al. [64], which we call ZHANG. Recall from Chapter 3 that this algorithm is the most relevant previously proposed algorithm to solve our target problem (as defined in Section 2.3). We then present an experimental evaluation of this algorithm, which leads us to investigate possible improvements. We then conclude on the merit of the general approach proposed by Zhang et al.

5.1 Overview

ZHANG is invoked repeatedly to decide which group of workflow tasks should be submitted as a single job to the batch scheduler next. More specifically, when invoked ZHANG submits a sequence of consecutive workflow levels as a single job to the batch scheduler, starting with the next level to be executed. To decide how many levels should be included in this sequence, the algorithm relies on a simple heuristic described hereafter.

Assuming that the first yet-to-be-executed level of the workflow is level l_{start} and that the workflow's last level is level l_{end}, the heuristic proceeds as follows. It iteratively considers scheduling all levels from level l_{start} to level l_i, for $i = end, start, start + 1, \ldots, end - 1$. Note that submitting all remaining tasks as a single job ($i = end$) is considered first because it corresponds to a "safe" baseline option. This iteration stops prematurely if a previously considered i is deemed to be a better option than the currently considered i. More precisely, at iteration i the algorithm considers executing all tasks in levels l_{start} to l_i as a single job that would request a number of nodes equal to the maximum width of these levels (in number of tasks) and sufficient run time to execute all tasks in these levels. A wait time estimate is obtained for this job from the batch scheduler. The ratio of the wait time to the run time is then computed. If this ratio for the currently considered i is lower than that for the previously considered i, then the iteration stops and the previously considered i is selected. Levels l_{start} to l_i are then submitted as a single job to the batch scheduler. However, if $i = end$ and if the estimated wait time is more than twice the duration of the run time, then each task is submitted as an individual job. This is another heuristic, based on the intuition that if wait time is much larger than run time, one should default to the one-job-per-task strategy in the hope that these small jobs will benefit from backfilling.

To improve overall makespan, the algorithm overlaps the run time of one job with the queue wait time of the following job. Whenever a submitted job begins execution, ZHANG is re-invoked for the yet-to-be-executed workflow levels, leading to a new job submission. However, wait time estimates are not 100% accurate. Also, the algorithm greedily submits jobs without consideration of when successive jobs are submitted. As a result, the overlap of one job's run time with the next job's wait time is likely never perfect. In particular, the next job could start "too soon." This

leads to unsatisfied task dependencies, causing tasks in this next job to wait idly before executing and possibly not complete within their job's requested execution time. In this case, if another job (following the current job) has already been submitted (it may even be running), it is canceled and ZHANG is invoked again for the yet-to-be-executed workflow levels. The execution of some of these levels may have been previously attempted (with a subset of their tasks successfully completed).

To reduce the number of occurrences of the above problem, ZHANG introduces the concept of a *leeway*. A leeway is an extra amount of time that is requested for a job, to ensure that tasks in the job can complete even if the job starts too early. Consider a job that starts execution at time 0 and will complete at time t. At time 0 ZHANG is invoked to submit a new job for execution. Say that this job has run time r and (estimated) wait time $w < t$. A naive leeway would be $t - w$, i.e., requesting $t - w + r$ time for this new job, making sure that all its task will complete successfully in spite of the job starting too early. This leeway is naive because requesting the extra time will change the wait time. In other words, the leeway both depends on and changes the wait time. ZHANG addresses this issue using a simple iterative approach to compute a sensible leeway. Note that, given non-determinism in the batch queue behavior, premature job expiration could still occur in spite of the added leeway. Note that some batch schedulers (e.g., Slurm) make it possible to declare job dependencies so that jobs cannot start too early, which could be used instead of the leeway mechanism (but may not allow as efficient overlap of wait times and execution times).

Figure 5.1 shows an example of a workflow execution in which ZHANG is repeatedly invoked. In this example, ZHANG splits the workflow execution into four individual jobs. ZHANG is first invoked at time t_1 and submits job #1, which executes all the tasks in a first set of consecutive workflow levels. At t_2, this job begins execution, and ZHANG is invoked again and submits job #2 (which comprises the tasks in the next set of contiguous levels). In this example, the run time of job #1 perfectly overlaps with the wait time of job #2; thus, job #2 can begin executing tasks as soon as it starts. At time t_3, ZHANG is invoked again and submits job #3. As shown, the wait time of job #3 is insufficient to completely overlap with the run time of job #2, so ZHANG adds a leeway to job #3's execution time to ensure that job #3 can complete all its tasks successfully. Job #3 begins its execution at time t_4 but must wait for job #2 to complete. This lost time is made up with the leeway, and job #3 can actually begin to execute at t_5. Although job #3 is idle until t_5, ZHANG is still invoked at t_4. At t_4, ZHANG submits job #4 (which in this example is the last job). As shown in the figure, the wait time of job #4 is longer than the execution time of job #3, meaning that for some period of time, until t_6, no workflow task is executing.

We found two issues with the algorithm as it is described in [64]. First, the algorithm aborts if the workflow's width is greater than the number of available compute nodes on the platform, i.e., if all tasks in a level cannot be executed in parallel. Since workflows can be very wide in practice, we have modified the algorithm so that a level can be executed in a job that has fewer allocated compute nodes than the level's width. We do this by using list-scheduling to greedily schedule all

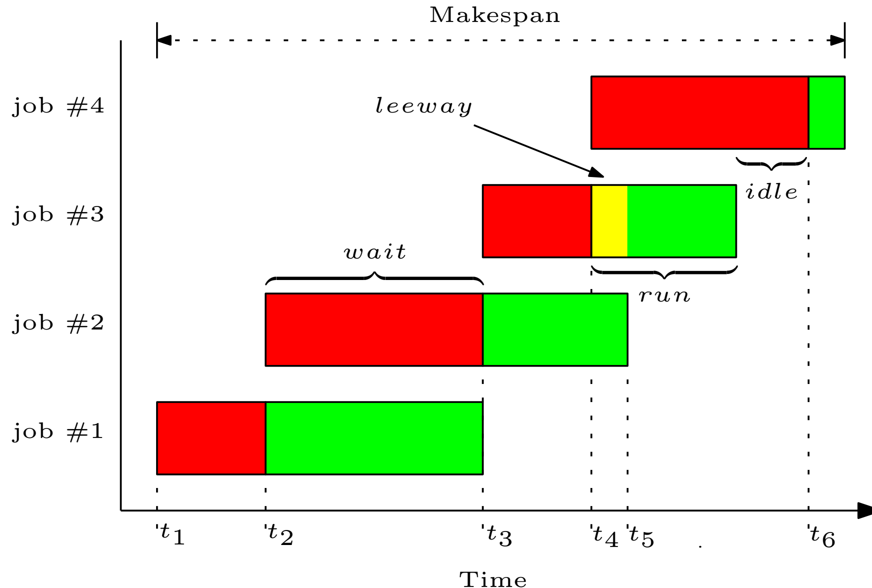

Figure 5.1: Workflow execution as 4 batch jobs using ZHANG.

tasks in a job onto an arbitrary number of compute nodes. Second, the pseudo-code in Figure 5 in [64] contains a potentially infinite loop for the leeway computation due to a mistake in the loop condition. We have modified this loop to ensure that it terminates (and computes the leeway as intended). From here on, when we refer to ZHANG we mean the algorithm in [64] with these two modifications.

5.2 Evaluation Results

In this section we perform an evaluation of ZHANG. We compare ZHANG to three competitors: ONEJOBPERTASK, ONEJOB, and LEVELBYLEVEL. ONEJOBPERTASK is the baseline approach that consists in submitting each task as a single job. ONEJOB is the other baseline approach that consists in submitting the entire workflow as a single job. This algorithm determines how many compute nodes should be requested. More specifically, it exhaustively considers every possible number of compute nodes (from 1 to the maximum parallelism of the workflow or the maximum number of compute nodes in the platform, whichever is smaller). For each, it estimates the workflow execution time (based on a schedule computed with a list-scheduling algorithm) and obtains an estimate of the wait time, so as to estimate the workflow makespan as the sum of these two times. It then submits the workflow as a single job, requesting the number of compute nodes that leads to the shortest (estimated) makespan. This algorithm was not considered as a competitor in [64]. Instead, the authors only consider a naive version that always uses the maximum number of compute nodes. The naive version is thus a weaker competitor as requesting fewer compute nodes often achieves

a better trade-off between wait time and execution time. Finally, LevelByLevel is a standard approach that consists simply in submitting each level of a workflow as a single job [50, 30]. This algorithm was not considered as a competitor in [64]. Like in OneJob, the best number of nodes for each job is determined based on wait time predictions. The job for a level is submitted only after the job for the previous level has completed.

We simulate the execution of the $*$-250-$*$ workflow configurations for the KTH batch workloads for 289 different submission times, and assuming accurate requested job execution times (see Section 4.3). We set the bound on the number of workflow jobs that can be in the system at the same time to 128, which, for these workflow configurations, means that OneJobPerTask is never limited by this bound. For each workflow submission we compute the percentage improvement in workflow makespan achieved by OneJobPerTask, OneJob, and LevelByLevel relative to Zhang. Positives values thus correspond to cases in which Zhang is outperformed by a baseline competitor.

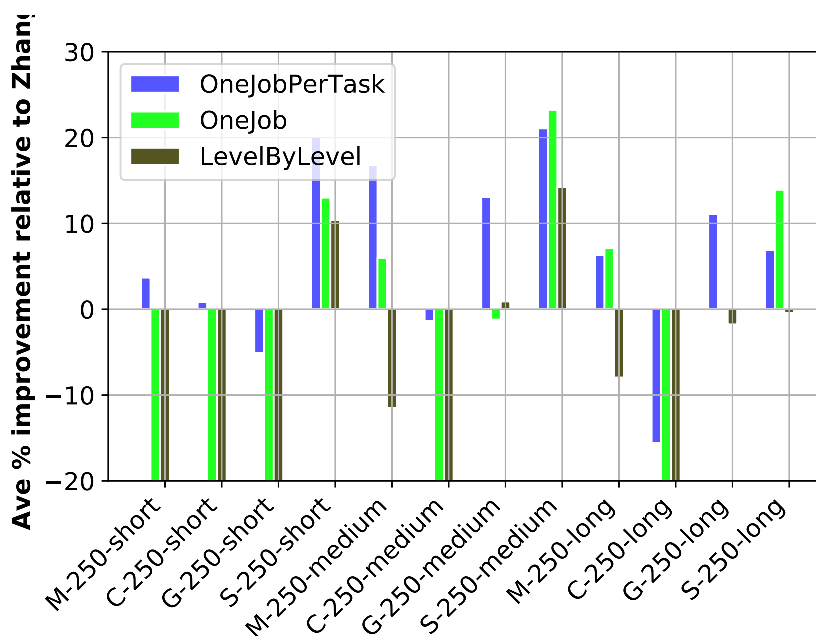

Figure 5.2: Average percentage improvement relative to Zhang for $*$-250-$*$ workflows on the KTH workload, with maximum number of simultaneously running workflow jobs at 128.

Figure 5.2 shows average relative improvements for the 12 workflow configurations. When comparing Zhang to OneJobPerTask, we find that Zhang leads to better results for only 3 workflow configurations (by 5.1%, 1.39%, and 15.29%). For the other 9 workflow configurations, OneJobPerTask leads to relative improvements of 11.16% on average and up to 21.06%. Recall from Section 5.1 that Zhang can default to OneJobPerTask. In these results, Zhang defaults to OneJobPerTask in 23.79% of the cases. When it does not default to OneJobPerTask, it

outperforms it in less than 30% of the cases. Overall, we find that although ZHANG can outperform ONEJOBPERTASK, in the vast majority of the cases ONEJOBPERTASK is preferable in practice (and is trivial to implement to boot). The other baseline competitor, ONEJOB, is similar to or outperforms ZHANG on average for 6 of the 12 workflow configurations. As expected, ONEJOB tends to fare better for workflow configurations with higher total execution times (even though there are some exceptions, e.g., the C-250-long configuration). This is because for workflows with long tasks, ONEJOBPERTASK (and ZHANG when it defaults to ONEJOBPERTASK) suffers from cascading task wait times. That is, while a set of (long) one-task jobs are executing (e.g., all tasks in the same level of the workflow), many other (background workload) jobs arrive to the system, causing longer queue wait times for the next set of one-task jobs. By contrast, ONEJOB "locks in" a slot in the batch queue at submission time and, due to the use of conservative backfilling, is not delayed by future job arrivals. Finally, LEVELBYLEVEL does not perform well. It outperforms ZHANG significantly for only 2 of the 12 workflow configurations, and in both cases it is outperformed by ONEJOB. This is because, unlike ONEJOB, it does not "lock in" a slot in the batch queue for the entire workflow execution.

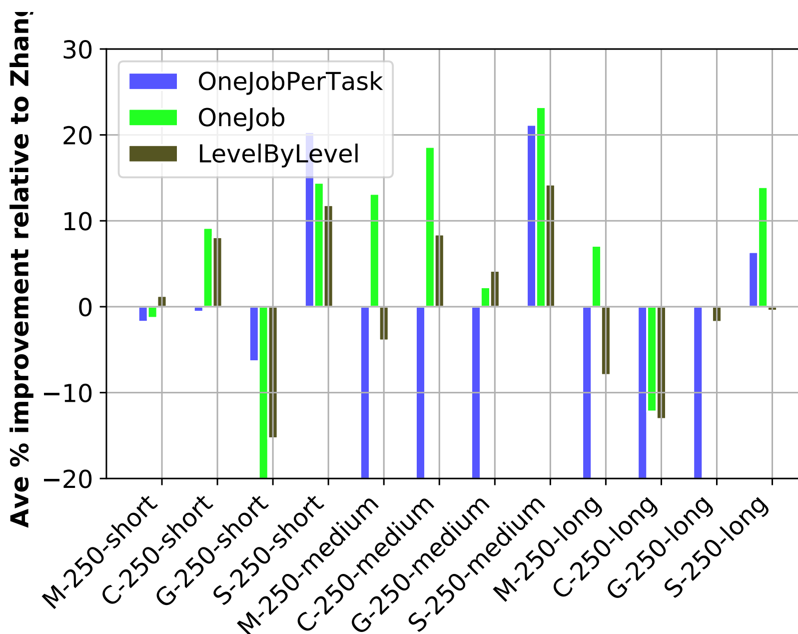

Figure 5.3: Average percentage improvement relative to ZHANG for ∗-250-∗ workflows on the KTH workload, with maximum number of simultaneously running workflow jobs at 16.

The results in Figure 5.2 correspond to a best case for ONEJOBPERTASK (and thus for ZHANG when it defaults to ONEJOBPERTASK) as the bound on the number of workflow jobs that can be in the system at a time (128) does not limit workflow job submissions. As discussed in Section 4.3, many production systems impose low bounds on the number of jobs by a single user. Figure 5.3

shows results when this bound is set to 16. In these results OneJob outperforms Zhang on average for 9 of the 12 workflow configurations (losing by 22.45%, 1.21%, and 12.29% in the other 3 workflow configurations). Expectedly, OneJobPerTask leads to the worst results overall, losing to Zhang for 9 of the 12 workflow configurations. Recall that Zhang can default to OneJob. In these results, Zhang does so in 48.32% of the cases. When it does not default to OneJob, Zhang outperforms OneJob in 52.53% of the cases. Here again, LevelByLevel does not perform well. When it outperforms Zhang significantly it is itself outperformed by OneJob.

We have performed the above experiments for our second workflow, SDSC, with the bound on the number of workflow jobs that can be in the system at the same time set to 16 and 128. Results are provided in Appendix A (Figure A.1), and show the same trends as the results for the KTH workload.

The takeaway from the above is that: (i) when the number of allowed workflow jobs in the system is not a limiting factor, OneJobPerTask typically outperforms Zhang; and (ii) when the number of allowed jobs is a limiting factor, OneJob typically outperforms Zhang; (iii) although used in practice, LevelByLevel does not compare well to its competitors. Importantly, Zhang does not consistently outperform the baseline competitors. A simple approach that would consist in choosing OneJobPerTask or OneJob based on the bound on the number of allowed workflow jobs may be as effective, and much simpler to implement, than Zhang. This leads us to the following question: is Zhang's poor performance due to flaws of its fundamental principle, or is it due to missed opportunities for implementing its principle more efficiently? To answer this question, we have implemented three improvements to Zhang, as described in the following section.

5.3 Improving Zhang

There are three straightforward ways to enhance the algorithm proposed in [64], while preserving its fundamental principles.

Enhancement #1: Making the search global – Zhang iteratively considers adding one extra workflow level to a batch job, greedily stopping when adding this level would worsen (i.e., increase) the wait time to execution time ratio of the job. The intuition is that shorter jobs are better, for they experience lower wait times. However, a larger job could have a lower ratio. More generally, the ratio can have several local minima over the possible numbers of consecutive levels included in a job. The enhancement, which does not increase the worst case asymptotic complexity of the algorithm, is simply to examine all options (rather than stopping as soon as the next option is worse than the previous option) and pick the best one.

Enhancement #2: Picking judicious numbers of requested compute nodes – Zhang always submits jobs that request as many compute nodes as possible to execute workflow tasks in parallel. This is almost never the best choice, and judicious job sizing is a well-known opportunity for striking a good compromise between wait time and execution time. The enhancement to the

algorithm is to consider all possible job sizes, between using a single compute node and using as many compute nodes as the workflow parallelism (bounded by the number of available compute nodes). For each such option, the wait time and the execution time of the jobs are estimated, and the option that leads to the earliest job completion time can then be picked. This enhancement linearly increases the number of wait time estimates that must be obtained.

Enhancement #3: Computing leeway via binary search – Recall that ZHANG, when submitting a job, can ask for more time than needed to ensure that the job does not start too early. This extra time is called the leeway and is computed as follows. Starting with the maximum amount of leeway needed to ensure that a job does not start too early (the naive leeway mentioned in Section 5.1), ZHANG adds half this leeway to the run time, requests a wait time estimate, and estimates the expected job start time. If this expected job start time is within 10 minutes (an arbitrarily chosen "small" duration) after the completion of its predecessor job, then the process stops and the augmented run time is used for the job submission. Otherwise, it repeats, adding half of a newly computed naive leeway to the job run time. This leeway computation is akin to a one-way binary search as the leeway can only increase. It is unclear why this idiosyncratic search algorithm is used instead of a standard binary search that could find a more precise leeway value. The enhancement consists in using a standard binary search to search the range between 0 and the naive leeway for the smallest possible leeway value that would cause the job to start within 10 minutes after the completion of its predecessor job.

The above enhancements yield 8 versions of ZHANG, depending on whether a particular enhancement is activated or not. We simulate workflow executions using these 8 algorithms for a subset of $*$-250-$*$ workflow configurations, using our two production workloads (KTH and SDSC), assuming accurate requested job execution times (see Section 4). We set the bound on the number of workflow jobs in the system to 8, 16, 32, 64, and 128.

We found no case in which one of our proposed enhancements (or any combination thereof) leads to a performance degradation. But the only enhancement that leads to any improvement is Enhancement #2, e.g., the use of queue wait time estimates to determine judicious job sizes. The use of this enhancement leads to improvements in 88 of the 1068 $*$-250-short workflow executions on SDSC. When an improvement is achieved, it is on average 3.57%. For the $*$-250-long workflows on KTH, the enhancement leads to improvements in 222 of the 822 executions. When an improvement is achieved, it is on average 6.52%. For the $*$-250-long workflows on SDSC, the enhancement leads to improvements in 425 of the 819 executions. When an improvement is achieved, it is on average 5.36%.

For the remainder of this paper, when using ZHANG, we always activate Enhancement #2. But given that only marginal performance improvements are achieved, the conclusions from the previous section still stands: ZHANG does not strike a worthwhile compromise between ONEJOB and ONEJOBPERTASK. This opens up an interesting question: is it possible to partition workflow

levels in a way that does strike a worthwhile compromise?

ZHANG tries to find a good compromise between wait time and execution time for the next set of workflow levels to be executed. The hope is that these local decisions will minimize makespan, without taking a global view of the workflow's execution, and even though wait time is at least partially hidden by execution time. We claim that this is the key problem with ZHANG: although intended to minimize makespan, it does not explicitly consider it as the objective function. In the next chapter, we propose an algorithm that tries to optimize the makespan explicitly.

CHAPTER 6
THE GLUME ALGORITHM

In this chapter, we describe a new algorithm, Group Levels Using Makespan Estimates (GLUME), to solve the workflow scheduling problem defined in Section 2.3. This algorithm, by contrast with ZHANG, is designed to group workflow levels into jobs based on estimates of the overall workflow makespan. In what follows, we give the main intuition behind the algorithm, followed by full details and pseudocode.

6.1 Intuition and Overview

Like ZHANG, GLUME is invoked repeatedly throughout workflow execution to decide on the next set of consecutive, yet-to-be-executed workflow levels to submit as a single job. While ZHANG greedily optimizes the wait time to execution time ratio of the next job to be submitted, GLUME explicitly minimizes the makespan. Given a workflow with n levels, GLUME considers all possible ways to partition the levels into two jobs: a first job to execute levels 0 to i and a second, subsequent, job to execute levels $i+1$ to $n-1$, for $i = 0, 1, \ldots, n-1$ (the second job could be empty). From here on, we refer to the first job (levels 0 to i) as J_i and to the second job (levels $i+1$ to $n-1$) as J'_i. Figure 6.1 shows an example workflow execution, where J'_i is submitted as soon as J_i begins executing, and where J'_i's wait time overlaps with J_i's execution. Like ZHANG, the overlap is achieved using a leeway mechanism. In the case of a premature job expiration, we use the same simple approach of canceling any subsequent job that has already been submitted, and invoking the algorithm again on the yet-to-be-executed workflow tasks.

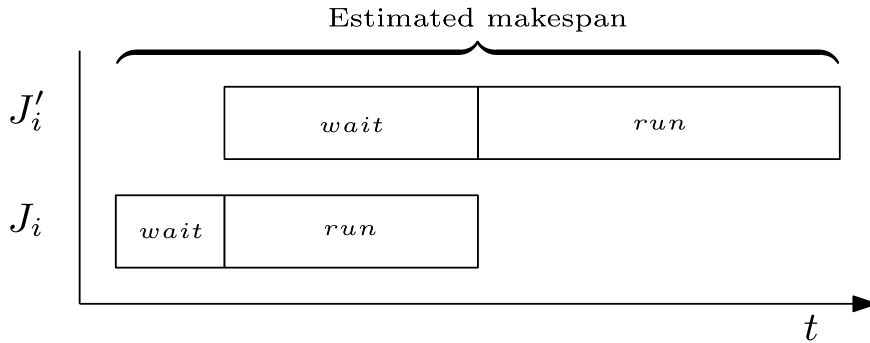

Figure 6.1: Example execution of the J_i and J'_i jobs by GLUME.

For each $i = 0, \ldots, n-1$, GLUME computes the best job configuration (a number of compute nodes and a requested execution time) for J_i and J'_i, i.e., the job configurations that minimize the estimated makespan (depicted in Figure 6.1). The i that leads to the lowest estimated makespan is chosen. J_i is then submitted to the batch scheduler, but J'_i is not. When J_i begins execution,

GLUME is invoked again on the remaining yet-to-be-executed workflow levels, possibly leading to partitioning these level into another two groups. In other words, every invocation of GLUME assumes a two-job execution but only submits the first job; thus, the overall execution of a workflow with n levels may be split into up to n distinct jobs. Unlike ZHANG, GLUME never opts for the one-job-per-task approach. It is thus never impacted by bounds on per-user numbers of jobs in the system (since at most two workflow jobs are in the system).

GLUME must estimate the wait times and execution times of J_i and J'_i, for all possible numbers of compute nodes, so as to estimate the overall makespan. For a given number of compute notes, execution time is estimated based on task schedules computed using list-scheduling, and a wait time estimate is obtained from the batch scheduler. The one difficulty is obtaining wait time estimates for J'_i. GLUME obtains this estimate "now", but J'_i will be submitted in the future. By then, the state of the batch queue will have changed and the wait time estimate could be (even more) inaccurate. Regardless, given no knowledge of the future, GLUME uses this estimate as a best effort wait time estimate.

6.2 Pseudocode

The pseudocode for GLUME is shown in Algorithm 1. GLUME is invoked at the beginning of the workflow execution, and repeatedly throughout workflow execution whenever a previously submitted job begins execution. GLUME takes two input: a workflow to execute (G) and a duration in seconds (*delay*). G only contains yet-to-be-executed tasks, and can thus be a subset of a larger workflow currently being executed. *delay* corresponds to an amount of time before the next submitted job, as decided by this invocation of GLUME, should begin executing. It is used to overlap the execution of a job with the wait time of the next job. Specifically, the first time GLUME is invoked, *delay* is zero, as the first job should be submitted immediately. For all subsequent invocations, which happen each time a previously submitted job begins executing, *delay* is this job's requested execution time. Each invocation of GLUME returns a level (l), a number of compute nodes (n), and a duration in seconds (t). Based on this allocation, a job is submitted to the batch scheduler that requests n compute nodes for t seconds to execute all tasks in levels 0 to l (inclusive) of G.

Line 1 of the algorithm sets variable *last* to the index of the last level of the workflow, for convenience. Line 2 declares an array A, where $A[i]$ will hold the best computed allocation for every considered J_i This allocation stores the job's wait time ($A[i].wait$), execution time ($A[i].run$), leeway to be added to the execution time ($A[i].leeway$), and number of nodes ($A[i]$.nodes). Line 3 declares an array M, where $M[i]$ holds the estimated workflow makespan for every considered J_i.

As mentioned in Section 5.1, submitting the entire workflow as a single job is often the safest choice. This is because once this job is submitted its wait time can be bounded (e.g., if the batch scheduler uses conservative backfilling), and thus also the workflow makespan. Therefore,

Algorithm 1 GLUME($G, delay$)

```
1: last ← number of levels in G − 1
2: let A[0..last] be a new array                         ▷ Allocations
3: let M[0..last] be a new array                         ▷ Makespans
4: A[last] ← PickBestAllocation(G, delay, 0, last)
5: M[last] ← A[last].wait + A[last].leeway + A[last].run
6: for l ← 0 to last − 1 do
7:     M[l] ← ∞
8:     A[l] ← PickBestAllocation(G, delay, 0, l)
9:     if A[l].leeway > 0.1 × A[l].run then
10:        continue
11:    end if
12:    delay′ ← A[l].leeway + A[l].run
13:    a′ ← PickBestAllocation(G, delay′, l + 1, last)
14:    if a′.leeway > 0.1 × a′.run then
15:        continue
16:    end if
17:    t ← A[l].wait + a′.wait + a′.leeway + a′.run
18:    if t < M[last] × (1 − beat) then
19:        M[l] ← t
20:    end if
21: end for
22: l ← argmin(M)
23: return (l, A[l].nodes, A[l].leeway + A[l].run)
```

Algorithm 2 PickBestAllocation ($G, delay, l_{start}, l_{end}$)

```
1: n ← PickBestNumNodes(G, delay, l_start, l_end)
2: run ← EstimateRuntime(G, n, l_start, l_end)
3: leeway, wait ← PickBestLeeway(G, delay, n, l_start, l_end, run)
4: return (n, run, leeway, wait)
```

Algorithm 3 PickBestNumNodes ($G, delay, l_{start}, l_{end}$)

```
1: maxnodes ← maximum usable number of compute nodes
2: let M[1..maxnodes] be a new array                     ▷ Putative Makespans
3: for n ← 1 to nodes do
4:     run ← EstimateRuntime(G, n, l_start, l_end)
5:     wait ← queue wait time prediction for a (n, run) job
6:     M[n] ← max(delay, wait) + run
7: end for
8: return argmin(M)
```

we consider executing the entire workflow as a single job as a baseline. This is done at lines 4 and 5. Line 4 computes the best allocation (procedure PickBestAllocation is described later in this section) for executing the entire workflow, and Line 5 computes the corresponding makespan (which is just the sum of the wait time, the leeway, and the execution time).

The for loop at lines 6-21 iterates over all i, $0 \leq i < last$, to search for the best way to partition the workflow levels, i.e., for the best J_i and J_i' pair. Line 7 sets the workflow makespan for J_i, $M[i]$, to infinity. At line 8, we calculate the best allocation for J_i, $A[i]$. This allocation has a certain leeway, and lines 9-11 are used to remove the current partition from consideration (i.e., leave $M[i]$ to infinity) if the leeway is more than 10% of the run time. This is a heuristic for avoiding resource waste (since all compute nodes are idle during the leeway period). Assuming that J_i is submitted to the batch scheduler, then as soon it begins executing J_i' will be submitted to the batch scheduler. The algorithm now calculates the allocation for J_i'. The delay for J_i', $delay'$ is computed at line 12 to J_i's total execution time (the sum of its leeway and its run time). The best allocation for J_i' is computed at line 13. As explained in the previous section, this allocation is computed using a wait time estimate obtained "now", even though the job will only be submitted in the future. Lines 14-16 apply again the heuristic to ensure that the leeway for J_i' is not more than 10% of the run time. Finally, the overall workflow makespan can be estimated at Line 17. This estimation account for the overlap of J_i's run time with J_i''s wait time. Lines 18-21 simply update the workflow makespan for this considered partition, but only if it is some fraction shorter than that of the baseline one-job option. The parameter $beat$ ($0 < beat \leq 1$) defines this fraction. The reason for this parameter is that since using multiple jobs is more "risky," one should prefer it to using a single job only if the expected performance gain is substantial.

Finally, at Line 22 the algorithm computes the index of the considered partition, l, that leads to the shortest makespan. The argmin notation denotes the index of the minimum element in an array, but this index can be computed easily as part of the loop using an extra variable. The algorithm then returns, its decision being that workflow levels 0 to l should be submitted for execution to the batch scheduler as a job that requests $A[l].nodes$ compute nodes and $A[l].leeway + A[l].run$ seconds of execution times.

The pseudocode for PickBestAllocation in shown in Algorithm 2. PickBestAllocation takes four input: a workflow (G), a duration in seconds ($delay$), a start level (l_{start}), and an end level (l_{end}). It returns a number of nodes (n), a run time (run), a leeway ($leeway$), and a wait time ($wait$). These are computed so that executing levels l_{start} to l_{end} as a single job that requests n compute nodes for $run + leeway$ seconds would lead to the earliest completion time, incurring a wait time of $wait$ seconds. n is computed at Line 1 via a call to PickBestNumNodes (described hereafter). run is computed at Line 2 by invoking helper function EstimateRuntime (pseudocode not shown), which estimate the overall execution time for executing all tasks in levels l_{start} to l_{end} on n compute nodes. This is done simply based on a list-scheduling heuristic (which

greedily schedules the ready tasks that can complete the earliest). At Line 3, another helper function, PickBestLeeway (pseudocode not shown), is called that returns *leeway* and *wait*. PickBestLeeway computes the best leeway (and the resulting wait time) in the interval $[0, delay]$ using a binary search (as described for Enhancement #3 in Section 5.3).

The pseudocode for PickBestNumNodes is shown in Algorithm 3. PickBestNumNodes takes four input: a workflow (G), a duration in seconds ($delay$), a start level (l_{start}), and an end level (l_{end}). PickBestNumNodes returns the best number of nodes that should be requested for a job that executes levels l_{start} to l_{end} of G. At line 1, we compute the maximum number of nodes that can be used, $maxnodes$. It is simply the minimum of the number of available compute nodes in the platform and of the maximum width of levels l_{start} to l_{end} in G. At line 2, we declare an array M, where $M[n]$ will be the estimated makespan when using n ($n = 0, \ldots, maxnodes$) compute nodes. These makespans are computed in the loop at Lines 3-7. The run time (run) is computed at Line 4, using the previously described EstimateRuntime helper function. The wait time ($wait$) is estimated at Line 5 based on a wait time estimate obtained from the batch scheduler for a job that requests n compute nodes for run seconds. $M[n]$ is then computed at Line 6. The first term accounts for the overlap of the execution of the currently running job (which will last $delay$ seconds) with the wait time of the job that is to be submitted (which will last $wait$ seconds). Finally, at Line 8 we return the index of the shortest makespan, which is also the best number of compute nodes to use.

Note that the pseudocode in this section is written for best readability rather than for best complexity. For instance, although EstimateRuntime is invoked redundantly by PickBestAllocation and PickBestNumNodes, our actual implementation avoids such redundant calls.

In the next chapter, we evaluate GLUME's performance experimentally and compare it to its competitors, that include a number of baseline algorithms as well as Zhang.

CHAPTER 7
EVALUATION RESULTS

In this chapter we compare GLUME to Zhang and to the baseline algorithms (OneJobPerTask, OneJob, and LevelByLevel) for both the KTH and SDSC workloads. In all experiments we fix the cap on the per-user number of simultaneous running jobs to 16, based on real-world configuration discussed in Section 4.3.

We do not describe in detail all results for all workflow configurations. Instead, we first describe detailed results for the $*$-$*$-medium configurations. We then present summary results for drawing a clear comparison between GLUME to Zhang across all workflow configurations. In all our discussion hereafter we say that an algorithm "beats" another algorithm for a particular workflow configuration if the average improvement in makespan is 5% or higher. Otherwise we say that the algorithms are equivalent.

7.1 Results for $*$-$*$-medium Workflow Configurations

7.1.1 Results with Accurate Job Durations

We first discuss results obtained when assuming accurate requested job durations (see Section 4.3). Figure 7.1 shows results for $*$-$*$-medium workflow configurations for the KTH workload, and Figure 7.2 shows results for the same workflow configurations for the SDSC workload. Positive results correspond to cases in which a particular algorithm outperforms Zhang. The main observation is that GLUME is equivalent or beats Zhang across the board. In these results, Zhang never beats GLUME, and GLUME beats Zhang for 5, resp. 7, of the 12 configurations for the KTH, resp. SDSC, workload. For several workflow configurations, improvements are above 30%. Among the baseline algorithms, OneJobPerTask and LevelByLevel overall fare poorly, while OneJob fares well, especially for the SDSC workload, but not as well as GLUME. Both GLUME and OneJob tend to fare better, relatively to Zhang, for larger workflow configurations (i.e., workflows with more tasks).

We have also performed the above experiments for $*$-$*$-short and $*$-$*$-long workflow configurations on the KTH and SDSC workloads with accurate job requested durations. Additional results are provided in Appendix B: Figure B.1 shows results for $*$-$*$-short workflow configurations, and Figure B.2 shows results for $*$-$*$-long configurations. Results in these figures are summarized and discussed further in Section 7.2.

7.1.2 Results with Realistic Job Durations

The results discussed in Section 7.1.1 are with the optimistic assumption that job durations, when submitted to the batch scheduler, are accurate. As discussed in Section 4.3, this is not the case

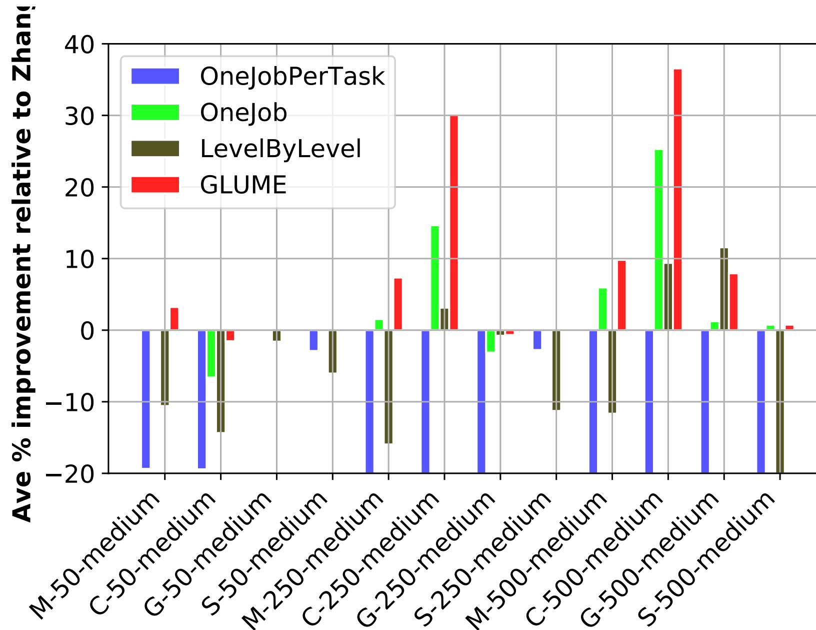

Figure 7.1: Average percentage improvement relative to ZHANG for ∗-∗-medium workflow configurations on the KTH workload, with maximum number of simultaneously running workflow jobs at 16 and accurate job requested durations.

in practice. As a result, wait time estimates provided by the batch scheduler are necessarily less accurate. Since GLUME relies heavily on these estimates, it could then be at a disadvantage. Figure 7.3 and Figure 7.4 are similar to Figure 7.1 and Figure 7.2, but the results are obtained with job requested durations taken directly from the batch logs of our two workloads. For the KTH workload, results are actually improved: GLUME beats ZHANG for 6 of the workflow considerations and is never beaten by it. However, results are drastically different for the SDSC workload: ZHANG beats GLUME for 10 of the 12 workflow configurations (by up to 55.3 % and by 24.6% on average). It turns out, that the jobs in the SDSC workload have very inaccurate durations. On average, jobs in the KTH workload request 1.01 hours more than they actually need. By contrast, in the SDSC workload, jobs request on average 3.34 hours more than needed! As a result, wait time estimates provided by the batch scheduler, which are heavily used by GLUME, are less accurate due to very large number of premature job completions. It is striking in Figure 7.4 that the ONEJOBPERTASK baseline algorithm performs almost equivalently to ZHANG, which often defaults to it, for 250- and 500-task workflows. In the face of a workloads with difficult to estimate wait times, the ONEJOBPERTASK approach works well for these workflow configurations.

We have also performed the above experiments for ∗-∗-short and ∗-∗-long workflow configurations on the KTH and SDSC workloads with realistic job requested durations. Results are provided in Appendix B: Figure B.3 shows results for ∗-∗-short workflow configurations, and Fig-

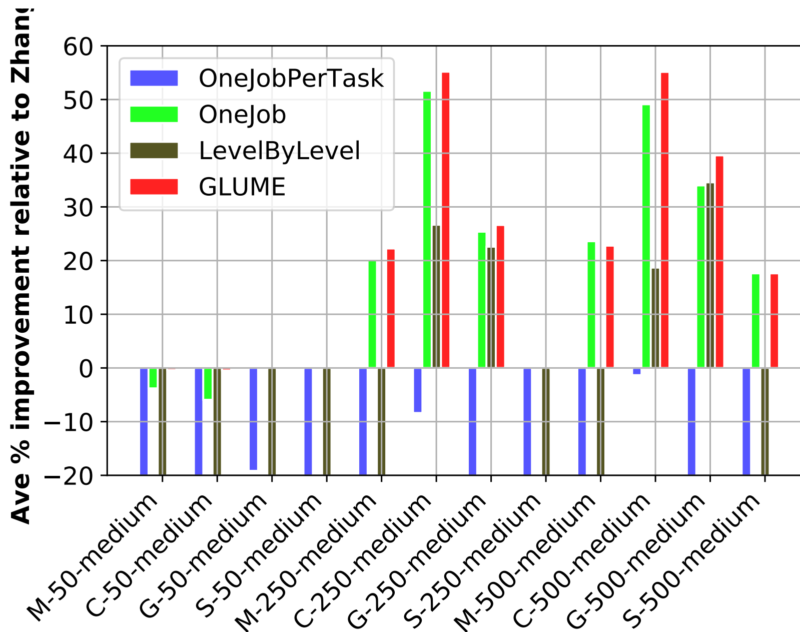

Figure 7.2: Average percentage improvement relative to Zhang for ∗-∗-medium workflow configurations on the SDSC workload, with maximum number of simultaneously running workflow jobs at 16 and accurate job requested durations.

ure B.4 shows results for ∗-∗-long configurations. Results in these figures are summarized and discussed further in Section 7.2.

7.2 Overview of All Results

Table 7.1 presents summary results over all workflow configurations, where each cell in the table shows the number of "wins" (i.e., more than 5% better) for GLUME and Zhang. For ∗-∗-long workflow configurations, GLUME outperforms Zhang for both workloads regardless of whether job requested durations are accurate or realistic. When the workflow's computational load is large, GLUME is able to make good decisions (or at least better than Zhang) in spite of inaccurate wait times. For ∗-∗-short workflow configurations, GLUME beats Zhang for the KTH workload, but less significantly than for ∗-∗-medium configurations (i.e., Zhang beats GLUME for 1 or 2 workflow configurations). For the SDSC workload, instead, GLUME is beaten by Zhang. It is beaten only by up to 8% when assuming accurate job requested durations, but by on average a factor 2 when using real-world such durations. The reason is similar to that for the ∗-∗-medium workflow configurations (i.e., inaccurate wait time estimates), but its effect is more pronounced for shorter workflows. Here again, the OneJobPerTask baseline algorithm, to which Zhang often defaults, works well (since tasks are short and/or wait time estimates are poor).

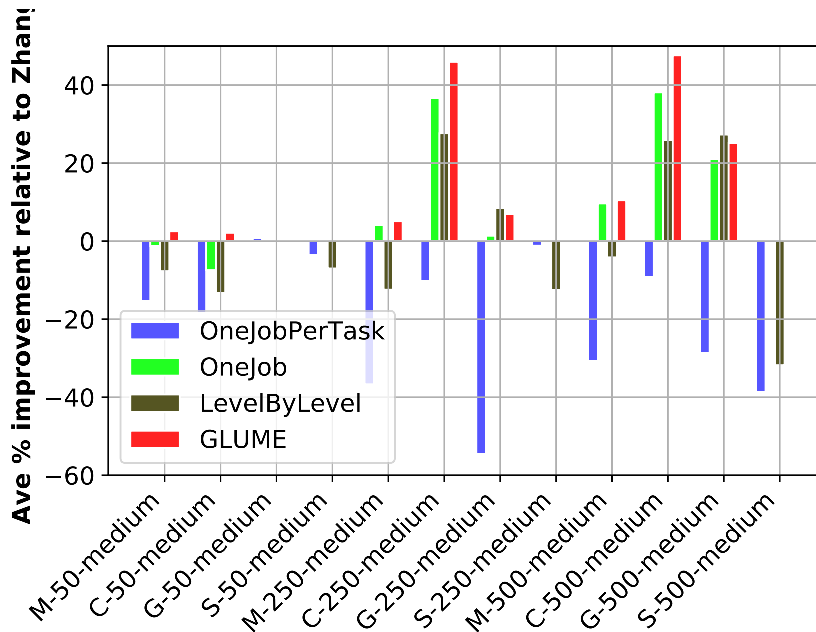

Figure 7.3: Average percentage improvement relative to ZHANG for ∗-∗-medium workflow configurations on the KTH workload, with maximum number of simultaneously running workflow jobs at 16 and realistic job requested durations.

7.3 Conclusion

The overall conclusion from the results presented in this chapter is that, in general, GLUME outperforms all other baseline algorithms regardless of workflow structure, task numbers, and task durations. However, GLUME relies heavily on accurate wait time predictions. Therefore, large disruptions to the batch scheduler, due to other users overestimating their job durations, detrimentally impact the effectiveness of GLUME and result in ONEJOBPERTASK and ZHANG (when defaulting to ONEJOBPERTASK) performing well. While on the KTH workload GLUME outper-

	accurate req. durations			real-world req. durations		
	workflow configurations			workflow configurations		
	short	medium	long	short	medium	long
KTH	**3/1**	**5/0**	**2/0**	**5/2**	**6/0**	**2/0**
SDSC	0/6	**7/0**	**4/0**	0/12	0/10	**4/0**

Table 7.1: Comparison of GLUME and ZHANG for both the KTH and SDSC workloads, assuming either accurate requested job durations (left side) or real-world requested job durations (right side) for ∗-∗-short, ∗-∗-medium, and ∗-∗-long workflow configurations. Each cell is formatted as x/y, where x, resp. y, is the number of workflow configurations for which GLUME, resp. ZHANG, beats ZHANG, resp. GLUME. Results in bold are those for which $x \geq y$ (i.e., GLUME beats ZHANG).

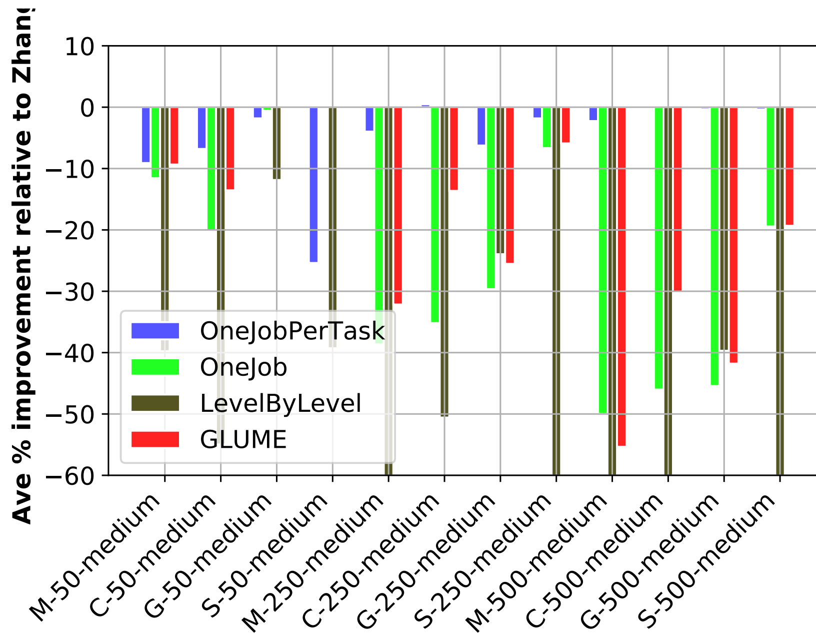

Figure 7.4: Average percentage improvement relative to ZHANG for ∗-∗-medium workflow configurations on the SDSC workload, with maximum number of simultaneously running workflow jobs at 16 and realistic job requested durations.

forms its competitors consistently, the same is not ture on the SDSC workload for which requested job durations are significantly less accurate.

CHAPTER 8
CONCLUSION

The research described in this thesis has produced propose a novel algorithm for executing the scientific workflows on batch-scheduled clusters, with the objective of minimizing the makespan. In this chapter we first summarize the findings and contributions of this thesis, and then outline directions for future work.

8.1 Summary of Findings and Contributions

In this thesis we have proposed an application-level approach for executing workflows efficiently on batch-scheduled resources. The main relevant previous work is the algorithm in [64], that uses several heuristics to partition workflow levels into batch jobs and overlap wait times with execution times. The algorithm proposed in this thesis, GLUME, operates in a similar manner, but its key contribution is that it explicitly optimizes workflow makespan. More precisely, at each decision point, GLUME considers that the yet-to-be-executed workflow levels will be split into two batch jobs. GLUME then considers all possible numbers of levels in the first job and estimates the overall makespan to pick the best such number. Importantly, this estimate relies on a estimate of the wait time for the second job "now", even though this job will be submitted at a later date (and which point the state of the batch queue will have changed). This estimate may thus be completely inaccurate, but allows GLUME to directly optimize for the makespan. A key question, that we answer experimentally, is whether GLUME can thus outperform its competitors. Another design decision for GLUME is that, unlike ZHANG, it never defaults to a one-job-per-task mode of execution. Therefore, GLUME is not susceptible to caps on per-user numbers of jobs enforced on production HPC platforms.

Simulation results obtained with production HPC workloads and various workflow configurations show that our proposed algorithm can vastly outperform the ZHANG algorithm in [64], as well as baseline ONEJOB, ONEJOBPERTASK, and LEVELBYLEVEL algorithms. In particular, the way in which GLUME groups workflow levels into jobs is superior to the only (to the best of our knowledge) previously proposed method for doing so, namely, the ZHANG algorithm. Although the results are overall positive, we have found GLUME to not perform well for workloads in which user-provided job duration estimates are very inaccurate. This is because GLUME relies heavily on wait time predictions. For such workloads, unless workflows are really large and/or with very long tasks, one is better off using the baseline ONEJOBPERTASK approach.

8.2 Future Work

There are three broad future directions for building on the work in this thesis.

The first future direction is for GLUME to no longer prohibit a one-job-per-task execution mode. This was a key design decision of the algorithm, motivate by the fact that production HPC platforms impose caps on the number of jobs a single user can have in the system at a time. However, our results have showed that for some workloads, queue wait time estimates can be so poor that GLUME can be outperformed by OneJobPerTask (at least for the workflow configurations considered in this thesis). This is not only because GLUME's makespan estimates are inaccurate, but also because OneJobPerTask can benefit greatly from backfilling opportunities created by premature job completions. Therefore, an enhancement to GLUME would be to allow it to, at some point during the workflow execution, default to OneJobPerTask. The Zhang algorithm uses a particular heuristic for defaulting to OneJobPerTask, and a similar heuristic would likely be effective for GLUME as well.

The second future direction is to augment GLUME so that it partitions workflows vertically as well as horizontally. In other words, a single workflow level could be split into multiple jobs, or, more generally, the workflow could be split into arbitrary, and dependent, sub-workflow structures (possibly allowing for task duplication so as to allow for simpler partitioning algorithms). This enhancement would be useful for workflows with parallelism that is relatively large when compared to the platform's parallelism. This would occur when a single workflow level would already be considered a very large job on the target platform. The design space for an algorithm that performs both vertical and horizontal partitioning of the workload is enormous, and the challenge will be to design relatively simple and yet effective heuristics.

The third future direction for this thesis is to implement the GLUME algorithm as part of a Workflow Management System (WMS) or as a stand-alone light-weight software framework. This framework would use standard interfaces to popular batch schedulers, e.g., Slurm, to submit appropriate jobs given a user-specified workflow specification. Users could then simply install this framework on their cluster to automate all their workflow executions.

APPENDIX A
ADDITIONAL FIGURES FOR CHAPTER 5

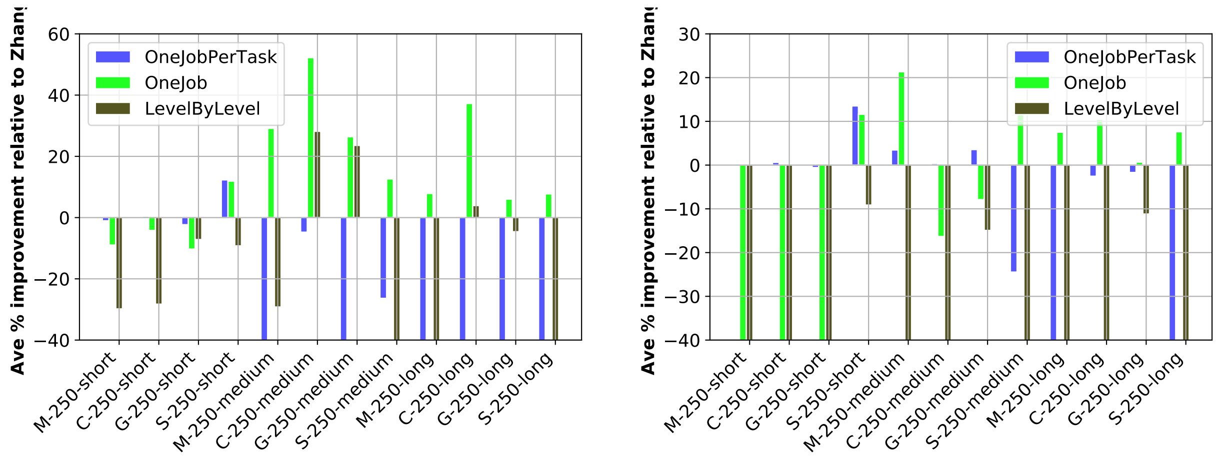

Figure A.1: Average percentage improvement relative to ZHANG for ∗-∗-medium workflow configurations on the SDSC workload, with maximum number of simultaneously running workflow jobs at 16 (left-hand side) and 128 (right-hand side) and accurate job requested durations.

APPENDIX B
ADDITIONAL FIGURES FOR CHAPTER 6

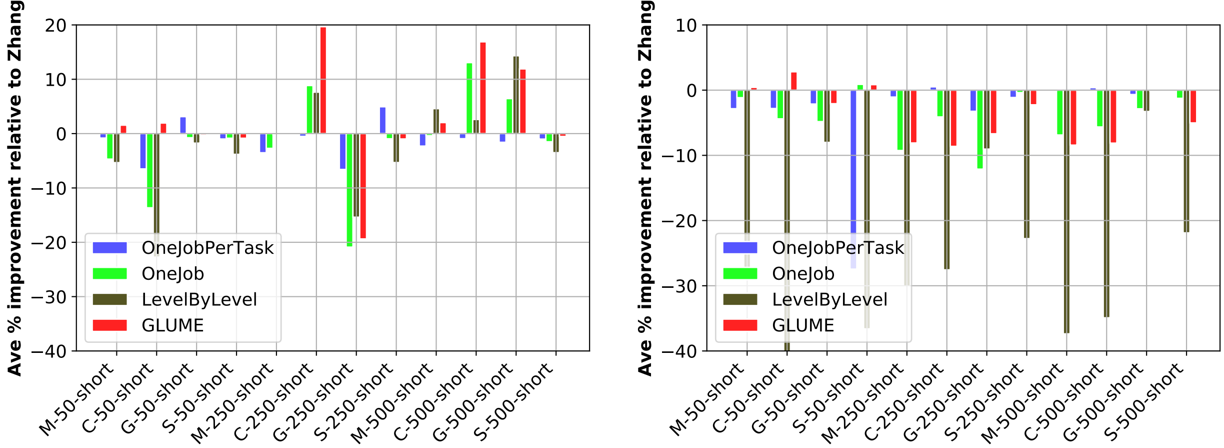

Figure B.1: Average percentage improvement relative to ZHANG for ∗-∗-short workflow configurations on the KTH (left-hand side) and SDSC (right-hand side) workloads, with maximum number of simultaneously running workflow jobs at 16 and accurate job requested durations.

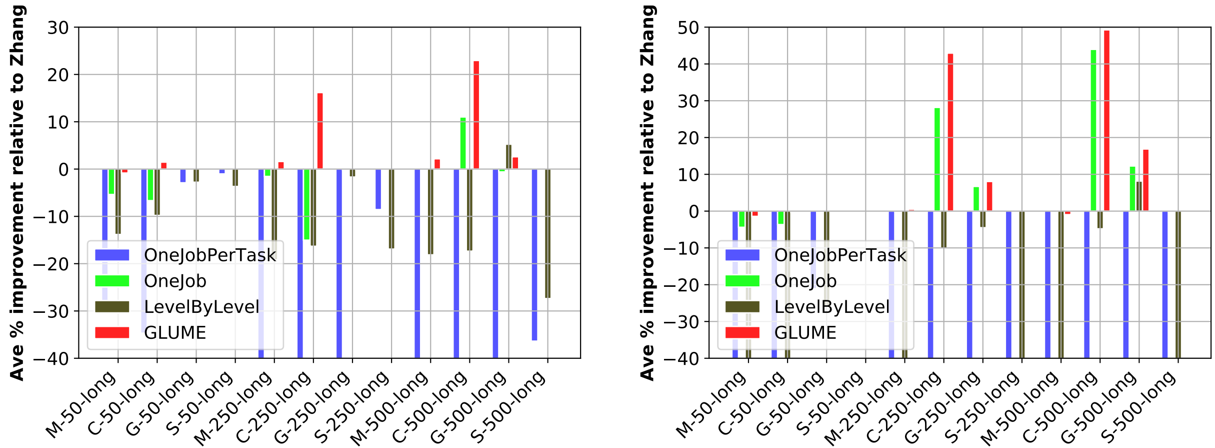

Figure B.2: Average percentage improvement relative to ZHANG for ∗-∗-long workflow configurations on the KTH (left-hand side) and SDSC (right-hand side) workloads, with maximum number of simultaneously running workflow jobs at 16 and accurate job requested durations.

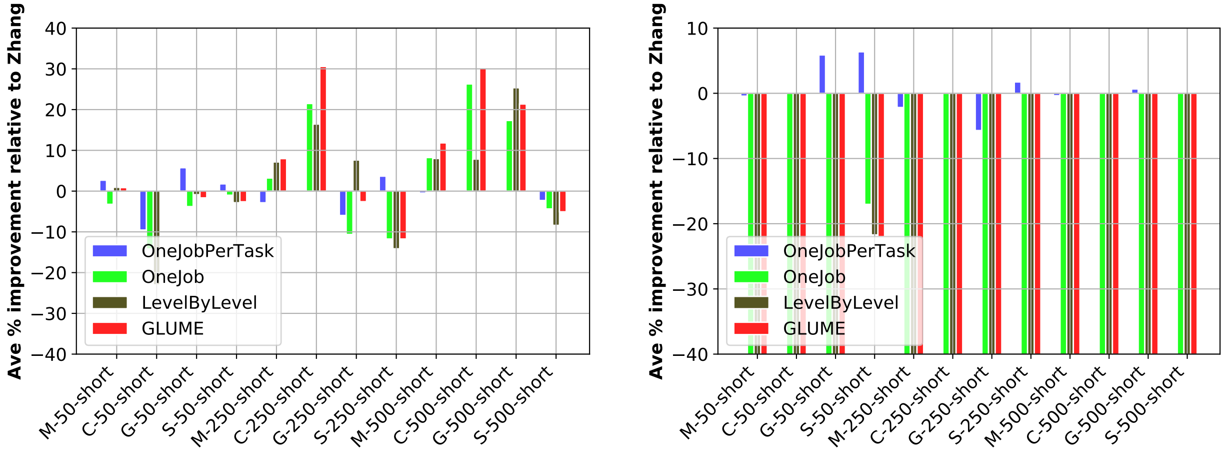

Figure B.3: Average percentage improvement relative to ZHANG for ∗-∗-short workflow configurations on the KTH (left-hand side) and SDSC (right-hand side) workloads, with maximum number of simultaneously running workflow jobs at 16 and realistic job requested durations.

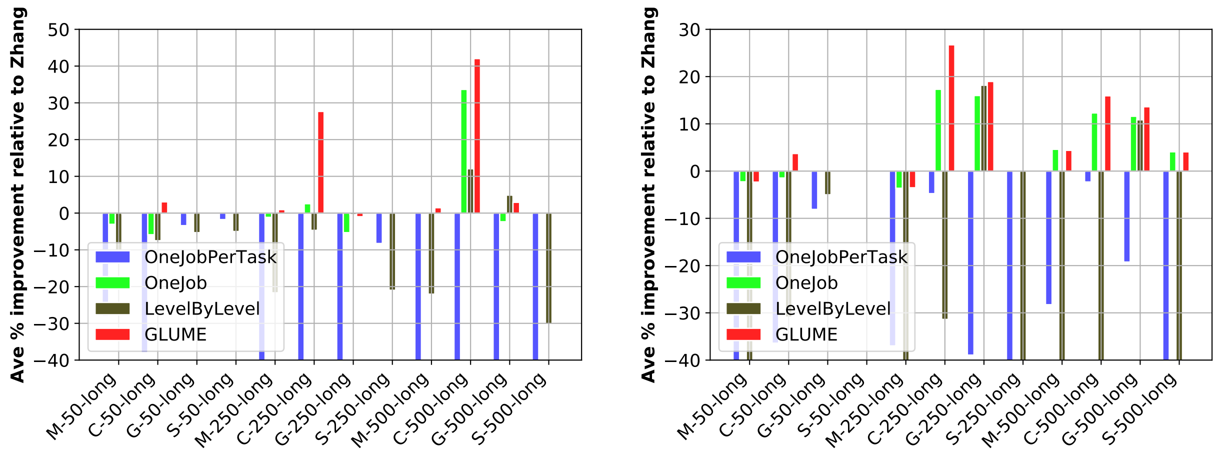

Figure B.4: Average percentage improvement relative to ZHANG for ∗-∗-long workflow configurations on the KTH (left-hand side) and SDSC (right-hand side) workloads, with maximum number of simultaneously running workflow jobs at 16 and realistic job requested durations.

BIBLIOGRAPHY

[1] Montage figure. https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator.

[2] Task-Clustering Batch Simulator. Available at https://github.com/wrench-project/task_clustering_batch_simulator, 2020.

[3] Dong Ahn, Ned Bass, Albert Chu, Jim Garlick, Mark Grondona, Stephen Herbein, Joseph Koning, Patki Tapasya, Thomas Scogland, Becky Springmeyer, and Michela Taufer. Flux: Overcoming Scheduling Challenges for Exascale Workflows. In *Proc. IEEE/ACM Workshop on Workflows in Support of Large-Scale Science (WORKS)*, pages 10–19, 11 2018.

[4] Michael Albrecht, Patrick Donnelly, Peter Bui, and Douglas Thain. Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids. In *1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, page 1. ACM, 2012.

[5] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludascher, and Steve Mock. Kepler: an extensible system for design and execution of scientific workflows. In *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*, pages 423–424. IEEE, 2004.

[6] Cooley Batch Scheduling Policies. https://www.alcf.anl.gov/support-center/cooley/job-scheduling-policies-cooley, 2020.

[7] Mira Batch Scheduling Policies. https://www.alcf.anl.gov/support-center/miracetusvesta/job-scheduling-policy-miracetusvesta, 2020.

[8] Theta Batch Scheduling Policies. https://www.alcf.anl.gov/support-center/theta/job-scheduling-policy-theta, 2020.

[9] Batsim-compatible algorithms implemented in C++. Available at https://gitlab.inria.fr/batsim/batsched, 2017.

[10] G Bruce Berriman, Ewa Deelman, John C Good, Joseph C Jacob, Daniel S Katz, Carl Kesselman, Anastasia C Laity, Thomas A Prince, Gurmeet Singh, and Mei-Hu Su. Montage: a grid-enabled engine for delivering custom science-grade mosaics on demand. In *Astronomical Telescopes and Instrumentation*, pages 221–232. International Society for Optics and Photonics, 2004.

[11] G Bruce Berriman, Gideon Juve, Ewa Deelman, Moira Regelson, and Peter Plavchan. The application of cloud computing to astronomy: A study of cost and performance. In *e-Science Workshops, 2010 Sixth IEEE International Conference on*, pages 1–7. IEEE, 2010.

[12] GB Berriman, JC Good, AC Laity, A Bergou, J Jacob, DS Katz, E Deelman, C Kesselman, G Singh, M-H Su, et al. Montage: a grid enabled image mosaic service for the national virtual observatory. In *Astronomical Data Analysis Software and Systems (ADASS) XIII*, volume 314, page 593, 2004.

[13] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M. Su, and K. Vahi. Characterization of scientific workflows. In *2008 Third Workshop on Workflows in Support of Large-Scale Science*, pages 1–10, November 2008.

[14] Daniel Blankenberg, Gregory Von Kuster, Nathaniel Coraor, Guruprasad Ananda, Ross Lazarus, Mary Mangan, Anton Nekrutenko, and James Taylor. Galaxy: a web-based genome analysis tool for experimentalists. *Current protocols in molecular biology*, pages 19–10, 2010.

[15] Duncan A Brown, Patrick R Brady, Alexander Dietz, Junwei Cao, Ben Johnson, and John McNabb. A case study on the use of workflow technologies for scientific analysis: Gravitational wave data analysis. In *Workflows for e-Science*, pages 39–59. Springer, 2007.

[16] Scott Callaghan, Ewa Deelman, Dan Gunter, Gideon Juve, Philip Maechling, Christopher Brooks, Karan Vahi, Kevin Milner, Robert Graves, Edward Field, et al. Scaling up workflow-based applications. *Journal of Computer and System Sciences*, 76(6):428–446, 2010.

[17] Scott Callaghan, Philip Maechling, Ewa Deelman, Karan Vahi, Gaurang Mehta, Gideon Juve, Kevin Milner, Robert Graves, Edward Field, David Okaya, et al. Reducing time-to-solution using distributed high-throughput mega-workflows-experiences from scec cybershake. In *eScience, 2008. eScience'08. IEEE Fourth International Conference on*, pages 151–158. IEEE, 2008.

[18] Scott Callaghan, Philip Maechling, Patrick Small, Kevin Milner, Gideon Juve, Thomas H Jordan, Ewa Deelman, Gaurang Mehta, Karan Vahi, Dan Gunter, et al. Metrics for heterogeneous scientific workflows: A case study of an earthquake science application. *International Journal of High Performance Computing Applications*, pages 274–285, 2011.

[19] H. Casanova, S. Pandey, J. Oeth, R. Tanaka, F. Suter, and R. Ferreira da Silva. WRENCH: A Framework for Simulating Workflow Management Systems. In *13th Workshop on Workflows in Support of Large-Scale Science (WORKS'18)*, 2018.

[20] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917, 2014.

[21] Weiwei Chen, Rafael Ferreira da Silva, Ewa Deelman, and Thomas Fahringer. Dynamic and Fault-Tolerant Clustering for Scientific Workflows. *IEEE Transactions on Cloud Computing*, 4(1):49–62, 2016.

[22] Weiwei Chen, Rafael Ferreira da Silva, Ewa Deelman, and Rizos Sakellariou. Using Imbalance Metrics to Optimize Task Clustering in Scientific Workflow Executions. *Future Generation Computer Systems*, 46(0):69–84, 2015.

[23] Tracy Craddock, Phillip Lord, Colin Harwood, and Anil Wipat. E-science tools for the genomic scale characterisation of bacterial secreted proteins. In *All hands meeting*, pages 788–795, 2006.

[24] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G Bruce Berriman, John Good, et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.

[25] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Phil J. Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, and Kent Wenger. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, 46(0):17–35, 2015.

[26] Maciej Drozdowski. *Scheduling for Parallel Processing.* Springer-Verlag London, 2009.

[27] Thomas Fahringer, Radu Prodan, Rubing Duan, Jüurgen Hofer, Farrukh Nadeem, Francesco Nerieri, Stefan Podlipnig, Jun Qin, Mumtaz Siddiqui, Hong-Linh Truong, et al. Askalon: A development and grid computing environment for scientific workflows. In *Workflows for e-Science*, pages 450–471. Springer, 2007.

[28] Rafael Ferreira da Silva, Weiwei Chen, Gideon Juve, Karan Vahi, and Ewa Deelman. Community resources for enabling and evaluating research on scientific workflows. In *10th IEEE International Conference on e-Science*, eScience'14, pages 177–184, 2014.

[29] Paul Fisher, Harry Noyes, Stephen Kemp, Robert Stevens, and Andrew Brass. A systematic strategy for the discovery of candidate genes responsible for phenotypic variation. In *Cardiovascular Genomics*, pages 329–345. Springer, 2009.

[30] William Fox, Devarshi Ghoshal, Abel Souza, Gonzalo P. Rodrigo, and Lavanya Ramakrishnan. E-HPC: A Library for Elastic Resource Management in HPC Environments. In *Proc. 12th Workshop on Workflows in Support of Large-Scale Science*, WORKS '17, pages 1:1–1:11, 2017.

[31] Rob Gaizauskas, Neil Davis, George Demetriou, Yikun Guod, and Ian Roberts. Integrating biomedical text mining services into a distributed workflow environment. In *Proceedings of UK e-Science All Hands Meeting*, 2004.

[32] A. Gerasoulis and T. Yang. A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors. *Journal of Parallel and Distributed Computing*, 16(4):276–291, 1992.

[33] Belinda Giardine, Cathy Riemer, Ross C Hardison, Richard Burhans, Laura Elnitski, Prachi Shah, Yi Zhang, Daniel Blankenberg, Istvan Albert, James Taylor, et al. Galaxy: a platform for interactive large-scale genome analysis. *Genome research*, 15(10):1451–1455, 2005.

[34] Hui Li, Juan Chen, Ying Tao, D. Gro, and L. Wolters. Improving a Local Learning Technique for Queue Wait Time Predictions. In *Proc. 6th IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, volume 1, pages 335–342, 2006.

[35] Gideon Juve, Ann Chervenak, Ewa Deelman, Shishir Bharathi, Gaurang Mehta, and Karan Vahi. Characterizing and profiling scientific workflows. *Future Generation Computer Systems*, 29(3):682–692, 2013.

[36] The Karnak prediction service. `http://karnak.xsede.org/`, 2019.

[37] Kepler/clotho integration. `http://sourceforge.net/projects/keplerclotho`.

[38] Rajath Kumar and Sathish Vadhiyar. Prediction of Queue Waiting Times for Metascheduling on Parallel Batch Systems. In *Proc. 20th Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 108–128, 2014.

[39] Cynthia Lee, Yael Schwartzman, Jennifer Hardy, and Allan Snavely. Are User Runtime Estimates Inherently Inaccurate? In *Proc. 10th international conference on Job Scheduling Strategies for Parallel Processing*, 2004.

[40] David A. Lifka. The ANL/IBM SP Scheduling System. In *Proc. Workshop on Job Scheduling Strategies for Parallel Processing*, volume 949, pages 295–303, 1995.

[41] The LSF Scheduler. `https://www.ibm.com/support/knowledgecenter/en/SSWRJV_10.1.0/lsf_welcome/lsf_welcome.html`, 2019.

[42] The MOAB Scheduler. `https://www.adaptivecomputing.com/moab-hpc-basic-edition/`, 2019.

[43] Montage. `http://montage.ipac.caltech.edu`.

[44] A. W. Mu'alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543, 2001.

[45] Daniel Nurmi, John Brevik, and Richard Wolski. QBETS: Queue Bounds Estimation from Time Series. In *Proc. 13th Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 76–101, 2007.

[46] The OAR Scheduler. `http://oar.imag.fr/`, 2019.

[47] Parallel Workloads Archive. https://www.cs.huji.ac.il/labs/parallel/workload/, 2019.

[48] The PBS Professional Scheduler. https://www.pbspro.org/, 2019.

[49] Millian Poquet. *Simulation approach for resource management.* Theses, Université Grenoble Alpes, December 2017.

[50] Gonzalo P. Rodrigo, Erik Elmroth, Per-Olov Östberg, and Lavanya Ramakrishnan. Enabling Workflow-Aware Scheduling on HPC Systems. In *Proc. 26th International Symposium on High-Performance Parallel and Distributed Computing*, pages 3–14, 2017.

[51] Gurmeet Singh, Mei-Hui Su, Karan Vahi, Ewa Deelman, Bruce Berriman, John Good, Daniel S. Katz, and Gaurang Mehta. Workflow Task Clustering for Best Effort Systems with Pegasus. In *Proceedings of the 15th ACM Mardi Gras Conference*, pages 1–9:8, 2008.

[52] SLURM - Resource Limits. https://slurm.schedmd.com/resource_limits.html, 2020.

[53] W. Smith, I. Foster, and V. Taylor. Scheduling with advanced reservations. In *Proc,. 14th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 127–132, May 2000.

[54] Ozan Sonmez, Nezih Yigitbasi, Saeid Abrishami, Alexandru Iosup, and Dick Epema. Performance Analysis of Dynamic Workflow Scheduling in Multicluster Grids. In *Proc. of 19th ACM International Symposium on High Performance Distributed Computing*, pages 49–60, 2010.

[55] I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields. *Workflows for e-Science: scientific workflows for grids.* Springer Publishing Company, Incorporated, 2014.

[56] Top500. https://www.top500.org/.

[57] Benjamin Tovar, Rafael Ferreira da Silva, Gideon Juve, Ewa Deelman, William Allcock, Douglas Thain, and Miron Livny. A Job Sizing Strategy for High-Throughput Scientific Workflows. *IEEE Transactions on Parallel and Distributed Systems*, accepted, 2017. Funding Acknowledgments: DOE ER26110.

[58] Jens-Sönke Vöckler, Gideon Juve, Ewa Deelman, Mats Rynge, and Bruce Berriman. Experiences using cloud computing for a scientific workflow application. In *Proceedings of the 2nd international workshop on Scientific cloud computing*, pages 15–24. ACM, 2011.

[59] Michael Wilde, Mihael Hategan, Justin M Wozniak, Ben Clifford, Daniel S Katz, and Ian Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, 2011.

[60] Katherine Wolstencroft, Robert Haines, Donal Fellows, Alan Williams, David Withers, Stuart Owen, Stian Soiland-Reyes, Ian Dunlop, Aleksandra Nenadic, Paul Fisher, et al. The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic acids research*, page gkt328, 2013.

[61] The WRENCH Project. `http://wrench-project.org`, 2018.

[62] A. B. Yoo, M. A. Jette, and M. Grondona. SLURM: Simple linux utility for resource management. In *Proc. 9th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 44–60, June 2003.

[63] Zhi-Feng Yu and Wei-Song Shi. Queue Waiting Time Aware Dynamic Workflow Scheduling in Multicluster Environments. *J. Comput. Sci. Technol.*, 25:864–873, 7 2010.

[64] Y. Zhang, C. Koelbel, and K. Cooper. Batch queue resource scheduling for workflow applications. In *Proc. IEEE International Conference on Cluster Computing and Workshops*, pages 1–10, August 2009.