GLUME: A Strategy for Reducing Workflow Execution Times on Batch-Scheduled Platforms

Evan Hataishi[1], Pierre-François Dutot[2],
Rafael Ferreira da Silva[3], and Henri Casanova[1]

[1] Information and Computer Sciences, University of Hawaii, Honolulu, HI, USA
[evanhata,henric]@hawaii.edu
[2] Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, Grenoble, France
pierre-francois.dutot@univ-grenoble-alpes.fr
[3] Information Sciences Institute, University of Southern California, Marina Del Rey, CA, USA rafsilva@isi.edu

Abstract. Many scientific workflows have computational demands that require the use of compute platforms managed by batch schedulers, which are unfortunately poorly suited to these applications. This work proposes GLUME, a strategy for partitioning a workflow into batch jobs. The novelty is that these jobs are explicitly constructed to minimize overall workflow execution time. Experimental evaluation via simulation of production batch workloads and workflows shows that our heuristic is more effective than previously proposed strategies when executing workflows with moderate to high computational demand.

Keywords: Batch Scheduling · Workflows · Task Clustering.

1 Introduction

Workflow applications are mainstream in the sciences and often require High Performance Computing platforms. Most such platforms run Resource and Job Management Software (RJMS) that implements batch scheduling. Batch scheduling was designed for workloads in which users submit moderate numbers of large, long-running, and loosely dependent parallel jobs. It is thus poorly suited to workflows. Workflow scheduling on batch-scheduled platforms can be attacked at the RJMS level [1,6,15] or at the application level [20,21,24,25]. Both kinds of solutions have drawbacks: RJMS-level solutions face adoption challenges while application-level solutions are impeded by constraints imposed on batch jobs.

We proposes an application-level strategy for minimizing workflow execution time, or *makespan*. This requires answering two questions. First, how should a workflow be partitioned into batch jobs? At one extreme is a one-job-per-task approach. This approach is problematic for workflows with long tasks (due to cascading wait times) and/or with many tasks (due to per-user caps on the number of running jobs). The other extreme, a workflow-as-a-single-job approach, is also problematic as large jobs suffer from long wait times. Second, when should workflow batch jobs be submitted? Submitting a job only once the previous job has completed is inefficient, but aggressively overlapping wait times and run times can lead to job expirations. To the best of our knowledge, the only previous work that provides non-trivial answers to these questions is that in [25]. The strategy therein uses wait time estimates to submit sets of consecutive workflow levels as batch jobs. We make the following contributions:

- The authors in [25] did not compare their proposed strategy to a baseline workflow-as-a-single-job approach that uses wait time estimates. We find that this simple approach can outperform that in [25] significantly.
- We propose a new strategy, GLUME, which, unlike that in [25], explicitly aims at minimizing makespan.
- We compare GLUME to the strategy in [25] and to baseline strategies in simulation, and find that GLUME is more effective than its competitors for workflows with moderate to high computational demands.

2 Related Work

Some authors have proposed RJMS that are well-suited to workloads that include workflows. The RJMS in [1] uses a hierarchical design to ensure that scheduling decisions can be made quickly even when the workload includes workflows with thousands or even millions of possible short-running tasks. In [15], the authors propose to augment the Slurm RJMS [23] to make it workflow-aware, ensuring that a workflow's jobs are positioned in adjacent positions in the batch queue. The approach in [6] inserts a RJMS between the application and the platform's native RJMS so as to allocate resources to workflow applications elastically.

In this work, we propose an application-level approach that can be used with non-workflow-aware, standard RJMS. Also, it can benefit RJMS-level solutions that execute workflows one level at a time [6, 15], making it possible to decide how to aggregate consecutive levels judiciously. The workflow scheduling literature is enormous but in this work we target batch-scheduled platforms, for which only a few approaches have been proposed. Some authors have proposed submitting each workflow task as a single job [20, 21, 24] Given high wait times incurred by the one-task-per-job approach in practice with current RJMS implementations [15], a few authors have proposed to group workflow tasks together into batch jobs. A commonly used baseline approach is to execute each workflow level as a single batch job [6, 15]. To the best of our knowledge, the only work that goes further is [25], which we detail and evaluate in Section 5.

Our approach, like those in [24,25], relies on wait time estimates. Wait times are notoriously difficult to predict. In this work, instead of predictions, we rely on wait time *estimates* as provided by production batch schedulers (see Section 3).

3 Problem Statement

We consider a cluster of homogeneous compute nodes, or *nodes*, managed by a RJMS that implements batch scheduling, i.e., a *batch scheduler*. The batch scheduler provides job wait time estimates based on currently running and pending jobs, assuming that all job run times are exactly as requested upon submission. Virtually all production batch schedulers provide "start time estimates" for pending jobs (e.g., `--start` option of Slurm's `squeue` command), and can thus in principle provide wait time estimates speculatively (e.g., the `showstart` command in MOAB/Torque. One source of inaccuracy of these estimates is that jobs terminate earlier than expected since users specify conservative time bounds [8]. Early job completions create *backfilling* opportunities, causing some jobs to start earlier than expected, and wait time estimates for these jobs were then pessimistic. Another source of inaccuracy is that backfilling may increase the wait times of other pending jobs, making wait time estimates for these jobs optimistic. This is the case with *aggressive backfilling* [10], but not with *conservative backfilling* [12], which leads to fairer schedules, possibly at the expense of lowered resource utilization, and makes wait time estimates more accurate. Several popular production batch schedulers implement conservative backfilling [13, 23]. In this work we only consider conservative backfilling.

On the cluster we wish to execute a static workflow, i.e., a directed acyclic graph of compute tasks, where each task executes on a single compute nodes and where edges denote task dependencies. Such static workflows of non-parallel tasks are commonplace in today's production scientific applications [9, 11, 17]. We assume that for each task we have an accurate estimate of its run time, including computation and I/O. 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 objective is to minimize *makespan*, i.e., the wall clock time between the first job submission and the last job completion.

4 Experimental Methodology

In this work we compare different strategies for executing workflows on a batch-scheduled cluster. These comparisons must be for a given workflow instance submitted at a given time to a batch-scheduled cluster subject to some background workload (i.e., other users submitting competing batch). It is impossible to perform these comparisons fairly on real-world systems, as back-to-back workflow executions would face different competing workload conditions. For this reason, like most previous works, we use simulation. We have implemented a simulator in C++ using the WRENCH [4] simulation framework and Batsched [3], a batch scheduler simulator. WRENCH provides the necessary high-level simulation abstractions for implementing the workflow scheduling algorithms we consider in this work. Batsched, which is a component of the Batsim simulator [5], implements the necessary batch scheduling algorithms.

Our simulator takes as input (i) a number of nodes; (ii) a workload trace file with competing job submissions; (iii) a workflow description file that specifies task run times and dependencies; (iv) the date at which the workflow execution begins (i.e., when the first task can be submitted for execution); and (v) a workflow scheduling strategy to use. The simulator outputs an execution log and the workflow makespan. The source code for our simulator is publicly available [18].

4.1 Workflow Configurations

Using a popular generator [16], we instantiate workflow configurations from these four applications: EPIGENOMICS (bioinformatics; many independent multi-level-fork join patterns with a single final a 3-task chain), SIPHT (bioinformatics; many independent 31-task structures whose first level consists of 23 independent tasks), CYBERSHAKE (earthquake engineering; sets of independent and massively parallel 2-level fork-join patterns), and MONTAGE (astronomy; a moderately parallel phase followed by a massively parallel phases, followed by a 6-level chain of sequential or moderately parallel phases). We refer the reader to [22] for more details about these workflows. For each application we generate workflows with $50, 250, 500 \pm 5$ tasks. The ± 5 is because the generator cannot produce a (realistic) configuration for any arbitrary workflow size. We generate workflows with sequential run times (i.e., sum of task run times) of $100, 500, 1000 \pm 3$ hours. The ± 3 is because the generator draws task execution times from random distributions. The 36 generated workflow configurations have overall and per-task computational demands that vary by orders of magnitude. We name each workflow x-y-z, where x denotes the application (E, S, C, or M); y denotes the number of tasks; and z denotes the sequential execution time (short, medium, or long). For instance C-250-short denotes a $\sim$ 250-task CYBERSHAKE workflow with sequential run time of $\sim$100 hours.

4.2 Batch scheduling and workloads

Table 1: Batch logs used to drive simulations

Workload	#nodes	#jobs	duration	utilization
KTH	100	∼28,000	11 months	∼70%
SDSC	128	∼60,000	24 months	∼83%
HPC2N	100	∼203,000	42 months	∼60%
CTC	338	∼77,000	11 months	∼85%

We configure our simulator to use conservative backfilling, using arrival times as job priorities. We simulate background workload by replaying job submission logs from production systems, as available in the Parallel Workloads Archive [14] (see Table 1). All simulations include a one day "warm-up" period, after which we simulate workflow execution when submitted on the half-hour for 6 days, 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 real job requested run times. For the former, we replace each job's requested run time by its actual run time. For the latter, we instead use the job's actual requested time from the log, which is known to be conservative [8]. Although unrealistic, we include the former method as a way to compare the merit of scheduling algorithms in ideal conditions w.r.t. wait time estimates. When discussing results we specify which method is used (either "accurate" or "real" estimates).

It is typical to cap the number of jobs a user can run simultaneously (e.g., setting the `MaxSubmitJobsPerAccount` for Slurm [19]). Taking the Argonne Leadership Computing Facility as an example: the recently decommissioned Mira system allowed users to have at most 5 running jobs per batch queue; The Cooley and the Theta systems allow both for up to 10 running jobs. Surveys of 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 low caps (i.e., below 20 and often below 10). As the caps can have a large impact of workflow executions we present results for different cap values.

5 The Algorithm by Zhang et al.

5.1 Overview

The algorithm proposed by Zhang et al. [25], which we call ZHANG, is invoked repeatedly to decide which workflow tasks should be submitted as batch job next. It considers the workflow as a sequence of *levels*, where each level comprises the tasks that have the same top-level (i.e., the same maximum distance from entry tasks). When all tasks in a level have been executed, all tasks in the next level can begin execution as it is guaranteed that all their parents have been executed. When invoked, ZHANG always submits a sequence of consecutive workflow levels as a single job to the batch scheduler, starting with the next level to be executed. A heuristic decides how many levels should be included in each job as follows.

Let l_{start} be the first yet-to-be-executed level of the workflow and l_{end} be the workflow's last level. The heuristic iteratively considers the option of scheduling all levels from level l_{start} to level l_i, for $i = end, start, start + 1, \ldots, end - 1$. Submitting all remaining tasks as a single job ($i = end$) is considered first as a "safe" baseline option. The iteration stops prematurely whenever a current option is deemed worse than the previously considered option. More precisely, at each iteration i ZHANG considers executing all tasks in levels l_{start} to l_i as a single job, requesting a number of nodes equal to the maximum width of these levels (in number of tasks) and a sufficiently long run time to execute all tasks in these levels. A wait time estimate is obtained from the batch scheduler and the ratio of the wait time to the run time is computed. If the ratio computed at the previous iteration, if any, was lower, then the iteration stops, the option evaluated at the previous iteration is selected, and the corresponding job is submitted to the batch scheduler. However, if $i = end$ is selected and the estimated wait time is more than twice the run time, then each task is submitted as an individual job. The rationale is that if wait time is much larger than run time, the one-job-per-task strategy is preferable as many small jobs may benefit from backfilling.

To overlap run times and wait times, whenever a submitted job begins execution ZHANG is re-invoked for the yet-to-be-executed workflow levels. Thus it is possible that the next job would start "too soon." This leads to unsatisfied task dependencies, causing the job to idle before being able to execute its tasks. These tasks may then fail to complete within the job's requested run time. Whenever a job expires with uncompleted tasks, any subsequent job that had been submitted is canceled, and ZHANG is re-invoked. To reduce the number of job expirations and re-executions, ZHANG uses a *leeway*, i.e., an extra requested amount of time to ensure that all 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 new 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 tasks 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. Since the leeway both depends on and changes the wait time, ZHANG computes the leeway using a simple iterative approach. However, given non-determinism in the batch queue behavior, a job could still expire before completing all its tasks.

Figure 1 shows an example execution in which ZHANG splits a workflow execution into four jobs. It 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 time t_2, this job begins execution, and ZHANG is invoked again and submits job #2 (which comprises the tasks in the next set of consecutive levels). In this example, the run time of job #1 overlaps perfectly 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 in the figure, 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 run 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 tasks in job #3 begin executing at time t_5. Although job #3 is idle until t_5, ZHANG is still invoked at time t_4 and submits job #4. But the wait time of job #4 is longer than the run time of job #3, so between times t_6 and t_7 no workflow task is being executed.

We found two issues with the algorithm as described in [25]. First, it aborts if the workflow's width is greater than the number of available compute nodes. Since workflows can be very wide in practice, we have modified the algorithm so that jobs can execute on arbitrary numbers of nodes. This is done via a standard list-scheduling approach (which greedily schedules the ready task that can complete the earliest). Second, the pseudocode in Figure 5 in [25] contains a potentially infinite loop for the leeway computation. We have modified this loop to ensure that it terminates (and computes the leeway as intended). From here on, by ZHANG we mean the algorithm in [25] with these two modifications.

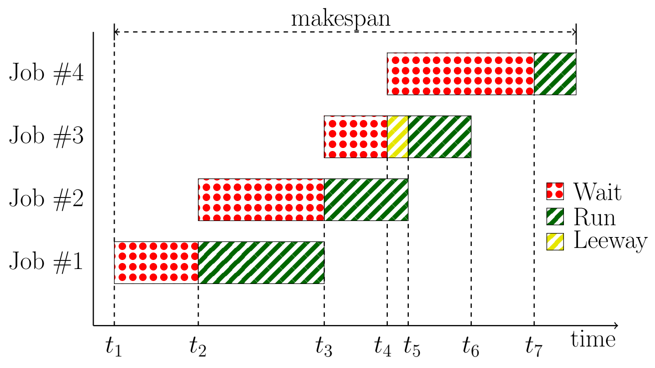


Fig. 1: Example workflow execution with 4 batch jobs using Zhang.

5.2 Evaluation Results

We compare Zhang to three baseline competitors: OneJobPerTask, OneJob, and LevelByLevel. OneJobPerTask submits each task as a single job. OneJob submits the entire workflow as a single job. To determine the number of nodes to request, OneJob exhaustively considers all possibilities (from 1 to the maximum parallelism of the workflow or the maximum number of nodes in the platform, whichever is smaller). For each, it estimates the workflow run time (using list-scheduling for scheduling tasks on nodes) and obtains an estimate of the wait time. It then submits the workflow as a single job, requesting the number of nodes that leads to the shortest (estimated) makespan. This strategy is not considered as a competitor in [25]. Instead, the authors consider a version that always requests the maximum number of nodes. It is thus a much weaker competitor than OneJob since requesting fewer nodes typically achieves a better trade-off between wait time and run time. Finally, LevelByLevel is a standard approach that submits each workflow level as a single job [6, 15]. This strategy was not considered as a competitor in [25]. Like OneJob, it determines the best number of nodes for each job based on wait time estimates. The job for a level is submitted only after the job for the previous level has completed. Finally, the results in [25] are only for wait times, while we instead consider overall makespan (which includes wait time) since this is the main metric of interest to users. Given all the above, it is not straightforward to make a direct comparison between the results hereafter and those in [25].

We simulate the execution of the $*$-250-$*$ workflow configurations for the KTH batch workload for our 289 different submission times, assuming accurate requested job run times (see Section 4.2). We set the cap on the number of running workflow jobs to 128, which, for these workflow configurations, means that OneJobPerTask is never limited. For each workflow submission we compute the percentage improvement in workflow makespan achieved by OneJobPerTask, OneJob, and LevelByLevel relative to Zhang. Positive values thus correspond to cases in which Zhang is outperformed by a baseline competitor.

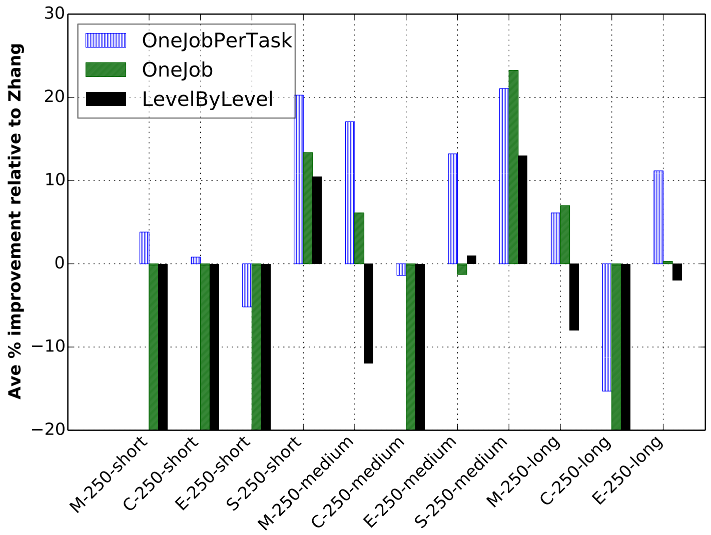


Fig. 2: Average percentage improvement relative to Zhang for ∗-250-∗ workflows on the KTH workload (number of simultaneously running workflow jobs capped at 128).

Figure 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%. Zhang, which can default to OneJobPerTask, does so in these results 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, OneJobPerTask is very often preferable to Zhang (and it is trivial to implement). The second baseline competitor, OneJob, is similar to or outperforms Zhang on average for 6 of the 12 workflow configurations. As expected, it fares better for workflows with higher total run times (but there are exceptions, e.g., the C-250-long configuration). This is because for workflows with long tasks, OneJobPerTask and Zhang, when it defaults to OneJobPerTask, suffer from cascading task wait times: while a set of (long) one-task workflow jobs are executing, many other (background workload) jobs arrive, causing longer queue wait times for the next set of one-task workflow jobs. By contrast, because conservative backfilling is used, OneJob "locks in" a slot in the batch queue at submission time. The third baseline competitor, LevelByLevel, does not perform well. It can outperform Zhang, but is always outperformed by OneJob. This is because, unlike OneJob, it does not "lock in" a slot in the batch queue.

The results in Figure 2 correspond to a best case for OneJobPerTask (and thus for Zhang when it defaults to OneJobPerTask) as the number of ongoing

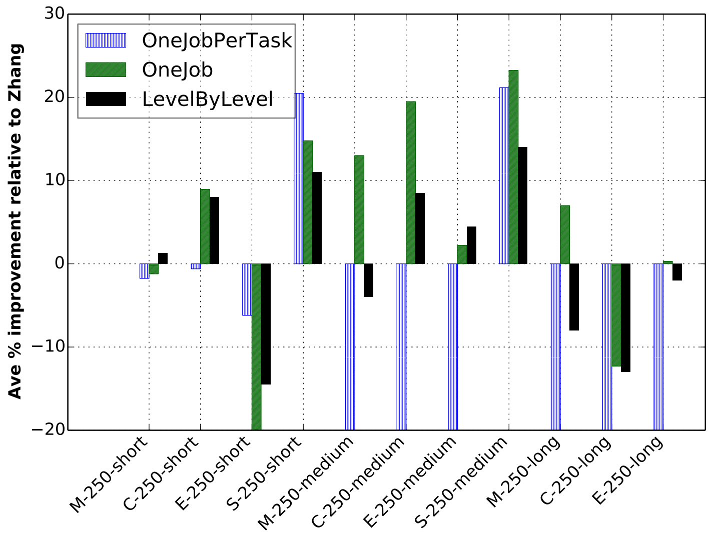


Fig. 3: Average percentage improvement relative to ZHANG for ∗-250-∗ workflows on the KTH workload (number of simultaneously running workflow jobs capped at 16).

workflow jobs is not capped. As discussed in Section 4.2, many platforms impose low caps on per-user numbers of running jobs. Figure 3 shows results when this cap 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, losing to ZHANG for 9 of the 12 workflow configurations. ZHANG can default to ONEJOB, and in these results it 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. Results for other workloads show similar trends (see [7] for full results).

5.3 Discussion

The takeaway from our evaluation of ZHANG is: (i) when the cap on the number of running workflow jobs is not a limiting factor, ONEJOBPERTASK typically outperforms ZHANG; (ii) when this cap is a limiting factor, ONEJOB typically outperforms ZHANG; and (iii) although used in practice, LEVELBYLEVEL does not compare well to its competitors. Importantly, ZHANG does not consistently outperform all the baseline competitors. This is unlike the results presented in [25], and is due to our considering more sound competitors and using the overall makespan as our performance metric. We experimented with three possible improvements to ZHANG, namely: (i) exhaustively examine all options for grouping levels together; (ii) pick the best number of nodes for each job based on wait and run time estimates; and (iii) compute better leeways via a true binary search. We have found that these improvements only lead to marginal improvements (see [7] for all details).

We hypothesize that ZHANG's disappointing results are because it does not explicitly seek to minimize the makespan. Instead, it strikes a compromise between wait time and run time for the next set of workflow levels to be executed, with the hope that these local decisions will reduce the makespan.

6 Proposed Algorithm

In this section, we describe GLUME (Group Levels Using Makespan Estimates), a strategy that explicitly aims at reducing workflow makespan. The pseudocode in this section is written for best readability rather than for best complexity.

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. But while ZHANG greedily optimizes the wait time to run time ratio of the next job to be submitted, instead GLUME attempts to minimize the makespan directly. 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 4 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 case of a premature job expiration, we use the same approach of canceling any subsequent job that has already been submitted and invoking GLUME again.

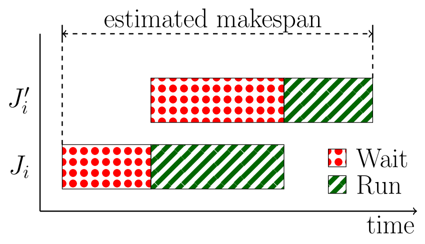


Fig. 4: Example execution of the J_i and J'_i jobs by GLUME.

For each $i = 0, \ldots, n-1$, GLUME computes the configuration (i.e., numbers of nodes and requested run times) for J_i and J'_i that minimize the estimated makespan. 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 levels into yet another two groups. In other words, every invocation of GLUME assumes a two-job execution but only submits the first job; thus, the execution of a workflow with n levels can entail up to n jobs. Unlike Zhang, GLUME never defaults to the one-job-per-task approach. It is thus never impacted by per-user caps on numbers of running jobs (since at most two workflow jobs are in the system at a time).

GLUME estimates the wait times and run times of J_i and J'_i for every possible number of nodes so as to pick the configuration that minimizes the (estimated) makespan. Run time is estimated based on task schedules computed using list-scheduling (since workflow levels can have more tasks than the number of requested compute nodes), and wait time estimates are obtained from the batch scheduler. This is similar to the approach used by the OneJob algorithm (see Section 5.2). The one difficulty here is obtaining wait time estimates for J'_i. GLUME obtains this estimate "now", but J'_i would be submitted in the future. By then, the state of the batch queue will have changed, further decreasing the accuracy of the wait time estimate. Nevertheless, given no knowledge of the future, GLUME uses this estimate as a best effort.

6.2 Detailed Description

The pseudocode for GLUME is shown in Algorithm 1. GLUME is invoked at the beginning of and repeatedly throughout workflow execution whenever a previously submitted job begins execution. GLUME takes as input a workflow (G) and a duration in seconds (*delay*). G only contains un-executed tasks (already executed tasks are ignored). *delay* is an amount of time before the next job to be submitted 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 each subsequent invocation, i.e., each time a previously submitted job begins executing, *delay* is this job's requested run time. Each invocation of GLUME returns a level (l), a number of nodes (n), and a duration in seconds (t). Based on this output, a job is submitted to the batch scheduler that requests n nodes for t seconds to execute all tasks in levels 0 to l (inclusive) of G.

Line 1 in Algorithm 1 sets variable *last* to the index of the last level of the workflow. Line 2 declares an array A, where $A[i]$ holds the best computed allocation for J_i. This allocation stores the job's wait time ($A[i].wait$), run time ($A[i].run$), leeway to be added to the run 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 J_i. As mentioned in Section 5.1, submitting the entire workflow as a single job is a safe option: once this job is submitted its wait time is bounded (provided the batch scheduler uses conservative backfilling), and thus also the workflow makespan. Therefore, we use the single-job option is used as a

Algorithm 1 GLUME$(G, delay)$

```
1: $last \leftarrow$ (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: $A[last] \leftarrow$ PickBestAllocation$(G, delay, 0, last)$
5: $M[last] \leftarrow A[last].wait + A[last].leeway + A[last].run$
6: for $l \leftarrow 0$ to $last - 1$ do
7:     $M[l] \leftarrow \infty$
8:     $A[l] \leftarrow$ PickBestAllocation$(G, delay, 0, l)$
9:     if $A[l].leeway > 0.1 \times A[l].run$ then
10:        continue
11:    end if
12:    $delay' \leftarrow A[l].leeway + A[l].run$
13:    $a' \leftarrow$ PickBestAllocation$(G, delay', l + 1, last)$
14:    if $a'.leeway > 0.1 \times a'.run$ then
15:        continue
16:    end if
17:    $t \leftarrow A[l].wait + a'.wait + a'.leeway + a'.run$
18:    if $t < M[last]$ then
19:        $M[l] \leftarrow t$
20:    end if
21: end for
22: $l \leftarrow$ argmin$(M)$
23: return $(l, A[l].nodes, A[l].leeway + A[l].run)$
```

baseline, which 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 run 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 makespan for J_i, $M[i]$, to infinity. Line 8 calculates 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 $M[i]$ as infinity) if the leeway is more than 10% of the run time. This is a heuristic for avoiding resource waste (since all 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 then calculates the allocation for J'_i. The delay for J'_i, $delay'$, is computed at Line 12 as J_i's total run 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 would 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. The overall workflow makespan is then estimated at Line 17, accounting for the overlap of J_i's run time with J'_i's wait time. Lines 18-21 simply updates the workflow makespan for this considered partition, but only if it is shorter than the baseline OneJob option. At Line 22 the algorithm computes the index of the partition, l, that leads to the shortest makespan (the argmin notation denotes the index of the minimum element in an array). The algorithm finally returns its decision that workflow levels 0 to l should be submitted to the batch scheduler as one job that requests $A[l].nodes$ nodes and $A[l].leeway + A[l].run$ seconds of run time.

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

```
1: $n \leftarrow$ PickBestNumNodes$(G, delay, l_{start}, l_{end})$
2: $run \leftarrow$ EstimateRuntime$(G, n, l_{start}, l_{end})$
3: $leeway, wait \leftarrow$
       PickBestLeeway$(G, delay, n, l_{start}, l_{end}, run)$
4: return $(n, run, leeway, wait)$
```

The pseudocode for PickBestAllocation is shown in Algorithm 2. It takes as 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 one job that requests n 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 by function PickBestNumNodes (described hereafter). run is computed at Line 2 by invoking function EstimateRuntime (pseudocode not shown), which estimates the run time of executing all tasks in levels l_{start} to l_{end} on n nodes. This is done using list-scheduling as for OneJob (see Section 5.2). At Line 3, another helper function, PickBestLeeway (pseudocode not shown), is called that returns $leeway$ and $wait$. PickBestLeeway computes the leeway (and the resulting wait time) in the interval $[0, delay]$ using binary search.

The pseudocode for PickBestNumNodes is shown in Algorithm 3. PickBest-NumNodes takes four inputs: 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 to request for a job that executes levels l_{start} to l_{end} of G. At Line 1, it computes the maximum number of nodes that can be used, $maxnodes$. It is simply the minimum of the number of 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 ($n = 0, \ldots, maxnodes$) 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 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, the index of the shortest makespan, i.e., the best number of nodes to use, is returned.

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

```
1: $maxnodes \leftarrow$ maximum usable number of nodes
2: let $M[1..maxnodes]$ be a new array                    ▷ Makespans
3: for $n \leftarrow 1$ to $maxnodes$ do
4:     $run \leftarrow$ EstimateRuntime$(G, n, l_{start}, l_{end})$
5:     $wait \leftarrow$ queue wait time estimate for a (n, run) job
6:     $M[n] \leftarrow \max(delay, wait) + run$
7: end for
8: return argmin$(M)$
```

GLUME has complexity $\mathcal{O}(N \cdot T \cdot M^2)$, where N is the number of workflow levels, T is the maximum number of tasks per level, and M is the number of compute nodes. Each invocation of GLUME requests $\mathcal{O}(M \cdot T + \log D)$ queue wait time estimates from the batch scheduler, where D is the maximum workflow job duration. The logarithmic term accounts for the leeway computation. Different batch schedulers may implement different algorithms for computing queue wait time estimates. This said, a batch scheduler that implements conservative backfilling must maintain a data structure that describes the current schedule and that allows scheduling decisions to be made with low complexity. This same data structure is used for obtaining queue wait time estimates (simply find the first "hole" in the schedule where a job could fit, but do not actually insert that job in the schedule). In the next section, we employ a simulator that uses an implementation of a batch scheduler that provides queue wait time estimates. Computing all necessary queue wait time estimates for each invocation of GLUME requires at most a few seconds on a single 2.5GHz core.

7 Results

In all experiments hereafter we fix the cap on the number of running workflow jobs to 16, which is a relatively high value as many production systems set the cap to 5 or 10 (see Section 4.2). With such lower cap values results for GLUME are further improved when compared to results presented hereafter.

7.1 Results for $*$-$*$-medium Workflows

We discuss results for the $*$-$*$-medium workflow configurations and the KTH and SDSC workloads. We picked these results because results for the HPC2N and CTC workloads are similar to that for the KTH workload, unlike results for the SDSC workload. Also, results for the short and long workflow configurations are more clear-cut and thus easily summarized (see the next section).

Figure 5 shows results obtained assuming accurate requested job run times (see Section 4.2). Positive values correspond to cases in which a particular strategy outperforms ZHANG. For the KTH workload, ZHANG beats GLUME for 2 of the 12 configurations (by at most 1.5%), while GLUME beats ZHANG for 8 of these configurations (by up to 36.6% and by 10.6% on average). For the SDSC workload, results are similar, with ZHANG beating GLUME for 3 configurations (by at most 0.5%), and GLUME beating ZHANG for 9 configurations (by up to 55.1% and by 26.6% on average). ONEJOBPERTASK and LEVELBYLEVEL fare poorly, while ONEJOB fares better, especially for the SDSC workload, but not as well as GLUME. The performance of GLUME and ONEJOB relative to that of ZHANG tends to improve as workflows comprise more tasks. This is because ZHANG often defaults to ONEJOBPERTASK and is thus limited by the cap on the number of running workflow jobs.

Figure 6 shows results for when requested job run times are taken directly from the batch logs, and are thus inaccurate in practical situations. For the KTH workload, results are actually improved : GLUME beats ZHANG for 9 of the 12 workflow configurations (by up to 47.6% and by 16.2% on average) but is never beaten by it. However, results are drastically different for the SDSC workload: ZHANG beats GLUME for 11 of the 12 workflow configurations (by up to 55.3% and by 22.42% on average). A reason for the poor performance of GLUME on the SDSC workload is that jobs in the SDSC workload request on average 3h20m more than needed. In the KTH workload, jobs request on average only 1h01m more than needed. As a result, wait time estimates provided by the batch scheduler are less accurate for the SDSC workload than for the KTH workload, which can negatively impact GLUME. This said, in the HPC2N and CTC workloads jobs request on average 4h27m and 3h56m more than needed, respectively. And yet, results with those workloads are consistent with those obtained with the KTH workload. There is thus some feature of the SDSC workload, which we were not able to pinpoint, that makes wait time estimates less accurate, which in turns penalizes GLUME. Although ZHANG also uses wait time estimates, in these results it often defaults to ONEJOBPERTASK, and is thus less impacted by inaccurate wait time estimates.

7.2 Overall Results

Table 2 summarizes results across all four workloads and 36 workflow configurations, assuming accurate (left hand-side) and real (right-hand side) requested job run times in the workloads. Each cell in the table shows the number of wins and losses of GLUME vs. one of its competitors for a given workflow computational demand (short, medium, and long). Each cell aggregates results over 48 experimental scenarios (4 workloads, 4 workflow types, 3 workflow sizes in number of tasks). Boldface is used for cells in which GLUME records more wins than losses. We define a "win", resp. "loss", as an average makespan decrease, resp. increase, by more than 5% when compared to a competitor. Otherwise, we

Table 2: Wins/losses of GLUME against competitors, aggregated for each workflow computational demand, assuming accurate (left) and real (right) requested job run times.

	Accurate requested run times			Real requested run times		
	Worklows			Workflows		
	short	medium	long	short	medium	long
OneJobPerTask	14/18	**40/0**	**37/0**	18/19	**27/8**	**37/0**
LevelByLevel	**22/6**	**36/1**	**35/0**	**21/10**	**33/1**	**34/0**
OneJob	**9/4**	**10/1**	**5/0**	**17/6**	**17/1**	**9/0**
Zhang	5/18	**17/1**	**9/0**	13/20	10/11	**14/0**

declare a draw. Without this threshold, the table would include many wins and losses for cases in which two competitors achieve results that would make little difference to end users. We pick 5% arbitrarily, but note that different values (say between 1% and 10%) lead to similar conclusions from the results.

Let us first consider the left-hand side of the table, i.e., assume that requested job run times in the workloads are accurate. GLUME outperforms LevelByLevel across the board, which was already noted in the previous section. GLUME clearly outperforms OneJobPerTask for medium and long workflows, but as expected does not fare as well for short workflows. GLUME outperforms OneJob as well, although not as drastically as it does LevelByLevel and OneJobPerTask. Finally, GLUME outperforms Zhang for medium and long workflows, only recording one loss for one medium workflow configuration. For short workflows, GLUME is outperformed by Zhang. This is because for these configurations Zhang often defaults to OneJobPerTask after executing only a few levels of the workflow, which turns out to be a winning strategy.

The results in the right-hand side of the table are for actual requested run times. While one might expect GLUME to perform worse since it should be more sensitive to wait time estimate inaccuracies, trends are similar. The one glaring difference is for medium workflow configurations, where GLUME experiences 11 losses to Zhang. 10 of these losses are seen in Figure 6 (right-hand side) and are for the SDSC workload, as discussed in Section 7.1).

Our main conclusion is that GLUME beats its competitors provided workflow computational demands are high enough, even with inaccurate queue wait time estimates. GLUME partitions workflow levels into jobs more effectively than Zhang, allowing it to outperform the baseline OneJob approach. Note that our "medium" workflow configurations only correspond to 500 hours of computation, which is at best modest given current workflow and platform trends [1]. It is thus only for very short workflows that GLUME loses to the approach that consists in running at most a few workflow levels as a single job before devolving to OneJobPerTask. For these workflows it is difficult to beat this approach as it benefits from backfilling opportunities without suffering from cascading wait times. But current trends make it clear that longer workflows are more broadly relevant to current practice in most scientific application domains [2].

8 Conclusion

We have proposed GLUME, a strategy for reducing the makespan of workflow executions on batch-scheduled platforms. GLUME partitions workflow levels into batch jobs and explicitly attempts to minimize the makespan. Simulation results show that GLUME outperforms the strategy in [25] as well as baseline strategies, provided workflow computational demands are moderate to high. GLUME relies on queue wait time estimates provided by batch schedulers and achieves good results in spite of the inaccuracy of these estimates.

There are three broad future directions for this work. The first is to allow GLUME to default to OneJobPerTask so that it can be effective even for very short workflows. This will require the development of a heuristic for deciding when to default to OneJobPerTask, which should take into account the platform's per-user cap on the number of running jobs. The second is to account for other metrics besides the makespan, such as resource utilization. A straightforward enhancement to GLUME would be to enforce that each workflow job achieves a minimum, user-provided resource utilization (by bounding the number of nodes allocated to each job). Much more challenging would be to consider a bi-objective makespan/utilization scheduling problem. The third is to augment GLUME so that it can partition workflows vertically as well as horizontally, e.g., splitting a single level into multiple jobs. This would be useful for cases in which a single workflow level would already be a large job relative to the target platform and would thus experience long wait time. Ultimately, our goal is to implement GLUME as part of a software tool for executing workflows on platforms managed by standard batch schedulers.

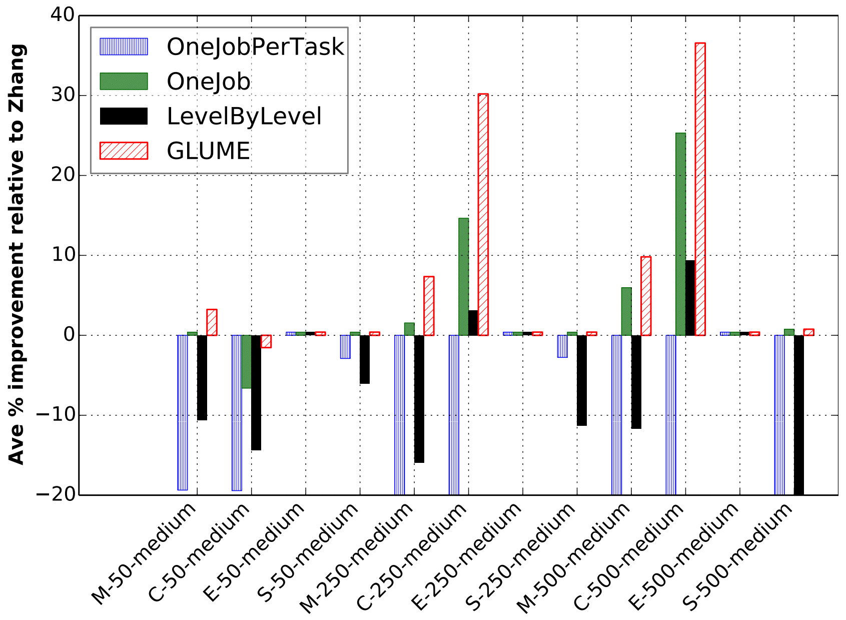
Acknowledgments

This work was partially funded by NSF contracts #1923539 and #1923621.

References

1. Ahn, D., Bass, N., Chu, A., Garlick, J., Grondona, M., Herbein, S., Koning, J., Tapasya, P., Scogland, T., Springmeyer, B., Taufer, M.: Flux: Overcoming Scheduling Challenges for Exascale Workflows. In: Proc. IEEE/ACM Workshop on Workflows in Support of Large-Scale Science (WORKS). pp. 10–19 (11 2018)
2. Atkinson, M., Gesing, S., Montagnat, J., Taylor, I.: Scientific workflows: Past, present and future. Future Generation Computer Systems **75**, 216–27 (2017)
3. Batsim-compatible algorithms implemented in C++. `https://gitlab.inria.fr/batsim/batsched` (2017)
4. Casanova, H., Ferreira da Silva, R., Tanaka, R., Pandey, S., Jethwani, G., Koch, W., Albrecht, S., Oeth, J., Suter, F.: Developing Accurate and Scalable Simulators of Production Workflow Management Systems with WRENCH. Future Generation Comp. Sys. **112**, 162–175 (2020). https://doi.org/10.1016/j.future.2020.05.030
5. Dutot, P.F., Mercier, M., Poquet, M., Richard, O.: Batsim: a Realistic Language-Independent Resources and Jobs Management Systems Simulator. In: Proc. 20th Workshop on Job Scheduling Strategies for Parallel Processing (2016)

6. Fox, W., Ghoshal, D., Souza, A., Rodrigo, G.P., Ramakrishnan, L.: E-HPC: A Library for Elastic Resource Management in HPC Environments. In: Proc. 12th Workshop on Workflows in Support of Large-Scale Science. pp. 1–11 (2017)
7. Hataishi, E.: Efficient Execution of Scientific Workflows on Batch-Scheduled Clusters. Master's thesis, University of Hawai'i at Mānoa (2020)
8. Lee, C., Schwartzman, Y., Hardy, J., Snavely, A.: Are User Runtime Estimates Inherently Inaccurate? In: Proc. 10th international conference on Job Scheduling Strategies for Parallel Processing. pp. 253–263 (2004)
9. Liew, C.S., Atkinson, M.P., Galea, M., Ang, T.F., Martin, P., Hemert, J.I.V.: Scientific workflows: moving across paradigms. ACM Computing Surveys **49**(4), 1–39 (2016)
10. Lifka, D.A.: The ANL/IBM SP Scheduling System. In: Proc. Workshop on Job Scheduling Strategies for Parallel Processing. vol. 949, pp. 295–303 (1995)
11. Liu, J., Pacitti, E., Valduriez, P., Mattoso, M.: A survey of data-intensive scientific workflow management. Journal of Grid Computing **13**(4), 457–493 (2015)
12. Mu'alem, A.W., Feitelson, D.G.: 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)
13. The OAR Scheduler. `http://oar.imag.fr` (2020)
14. Parallel Workloads Archive. `https://www.cs.huji.ac.il/labs/parallel/workload` (2020)
15. Rodrigo, G.P., Elmroth, E., Östberg, P.O., Ramakrishnan, L.: Enabling Workflow-Aware Scheduling on HPC Systems. In: Proc. 26th Intl. Symp. on High-Performance Parallel and Distributed Computing. pp. 3–14 (2017)
16. Ferreira da Silva, R., Chen, W., Juve, G., Vahi, K., Deelman, E.: Community resources for enabling and evaluating research on scientific workflows. In: 10th IEEE International Conference on e-Science. pp. 177–184. eScience'14 (2014)
17. Ferreira da Silva, R., Filgueira, R., Pietri, I., Jiang, M., Sakellariou, R., Deelman, E.: A Characterization of Workflow Management Systems for Extreme-Scale Applications. Future Generation Computer Systems **75**, 228–238 (2017)
18. Task-Clustering Batch Simulator. `https://github.com/wrench-project/task_clustering_batch_simulator` (2020)
19. Slurm Resource Limits. `https://slurm.schedmd.com/resource_limits.html` (2020)
20. Sonmez, O., Yigitbasi, N., Abrishami, S., Iosup, A., Epema, D.: Performance Analysis of Dynamic Workflow Scheduling in Multicluster Grids. In: Proc. of 19th ACM Intl. Symp. on High Performance Distributed Computing. pp. 49–60 (2010)
21. Tovar, B., da Silva, R.F., Juve, G., Deelman, E., Allcock, W., Thain, D., Livny, M.: A Job Sizing Strategy for High-Throughput Scientific Workflows. IEEE Transactions on Parallel and Distributed Systems **29**(2), 24–253 (2017)
22. Pegasus Workflow Gallery. `https://pegasus.isi.edu/workflow_gallery` (2020)
23. Yoo, A.B., Jette, M.A., Grondona, M.: SLURM: Simple Linux Utility for Resource Management. In: Proc. 9th International Workshop on Job Scheduling Strategies for Parallel Processing. pp. 44–60 (Jun 2003)
24. Yu, Z.F., Shi, W.S.: Queue Waiting Time Aware Dynamic Workflow Scheduling in Multicluster Environments. J. Comput. Sci. Technol. **25**, 864–873 (2010)
25. Zhang, Y., Koelbel, C., Cooper, K.: Batch queue resource scheduling for workflow applications. In: Proc. IEEE Intl. Conf. on Cluster Computing. pp. 1–10 (2009)



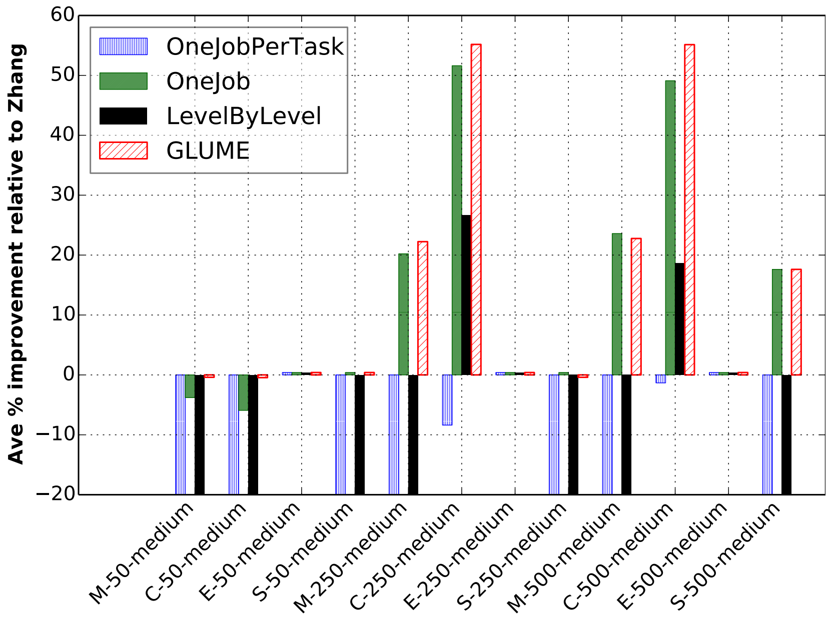
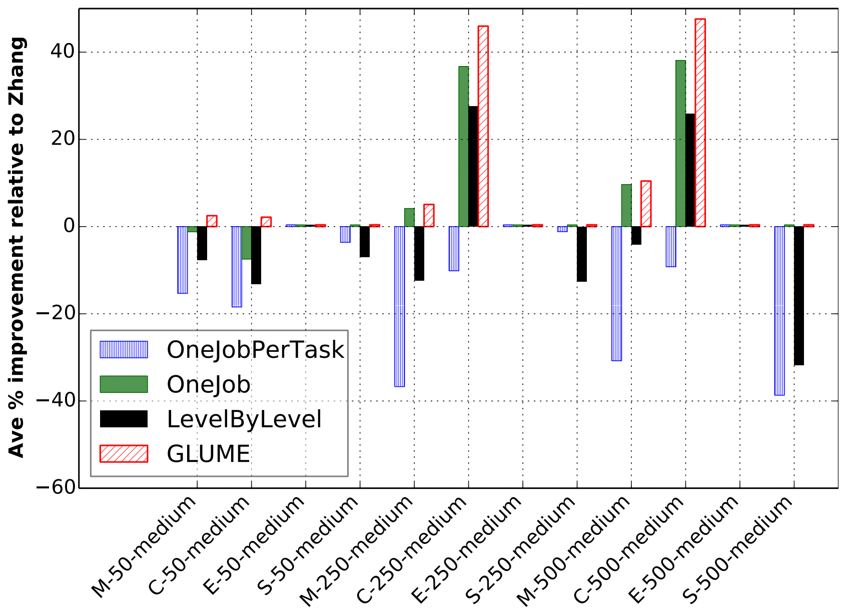


Fig. 5: Average percentage improvement relative to ZHANG for ∗-∗-medium workflows on the KTH (top) and SDSC (bottom) workloads, with the number of simultaneously running workflow jobs capped at 16 and assuming accurate job requested run times.



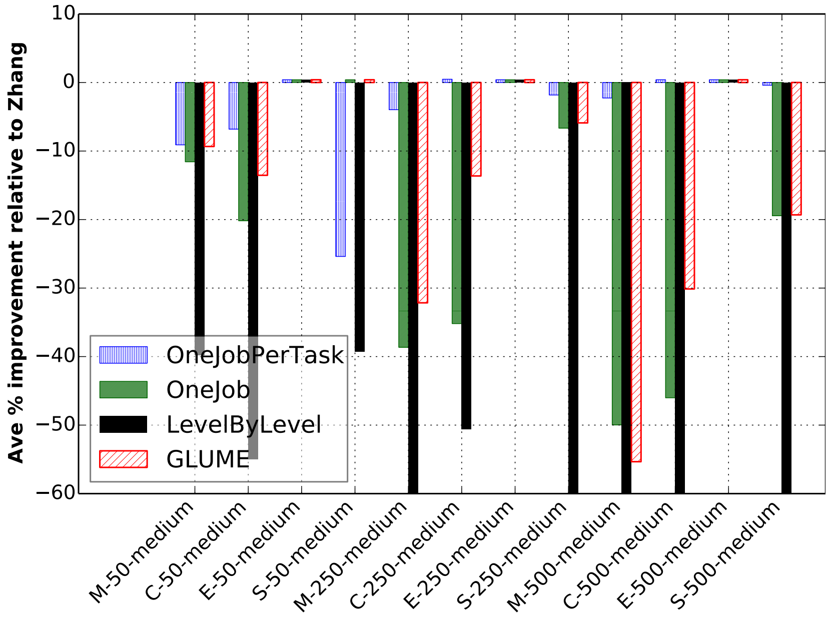


Fig. 6: Average percentage improvement relative to Zhang for ∗-∗-medium workflows on the KTH (top) and SDSC (bottom) workloads, with the number of simultaneously running workflow jobs capped at 16 and realistic job requested run times.