Acta Cybernetica **25** (2021) 131–150.

Execution Time Reduction in Function Oriented Scientific Workflows[*]

Ali Al-Haboobi[ab] and Gabor Kecskemeti[ac]

Abstract

Scientific workflows have been an increasingly important research area of distributed systems (such as cloud computing). Researchers have shown an increased interest in the automated processing scientific applications such as workflows. Recently, Function as a Service (FaaS) has emerged as a novel distributed systems platform for processing non-interactive applications. FaaS has limitations in resource use (e.g., CPU and RAM) as well as state management. In spite of these, initial studies have already demonstrated using FaaS for processing scientific workflows. DEWE v3 executes workflows in this fashion, but it often suffers from duplicate data transfers while using FaaS. This behaviour is due to the handling of intermediate data dependency files after and before each function invocation. These data files could fill the temporary storage of the function environment. Our approach alters the job dispatch algorithm of DEWE v3 to reduce data transfers. The proposed algorithm schedules jobs with precedence requirements to primarily run in the same function invocation. We evaluate our proposed algorithm and the original algorithm with small- and large-scale Montage workflows. Our results show that the improved system can reduce the total workflow execution time of scientific workflows over DEWE v3 by about 10% when using AWS Lambda.

Keywords: scientific workflows, cloud functions, serverless architectures, makespan

1 Introduction

Over the recent years scientific workflows have been a major area of interest within the field of complex scientific applications. Large-scale scientific workflows consist

[*]This work was supported in part by the Hungarian Scientific Research Fund under Grant agreement OTKA FK 131793

[a]Institute of Information Technology, University of Miskolc, Miskolc, Hungary

[b]University of Kufa, Najaf, Iraq, E-mail: al-haboobi@iit.uni-miskolc.hu, ali.alhaboobi@uokufa.edu.iq, ORCID: 0000-0001-7632-2485

[c]School of Computer Science and Mathematics, Liverpool John Moores University, Liverpool, UK, E-mail: kecskemeti@iit.uni-miskolc.hu, g.kecskemeti@ljmu.ac.uk, ORCID: 0000-0001-5716-8857

DOI: 10.14232/actacyb.288489

of a significant number of dependent jobs that rely on the output of other jobs (i.e., precedence constraints). Each job can be executed independently when its precedence constraints are met. Montage [11], CyberShake [10], and LIGO [1] are examples of scientific workflow applications. Workflow Management Systems (WMSs - such as Pegasus [8] and Kepler [2]) are used to ensure the precedence execution order and data constraints of every job in a scientific workflow are met during their runtime.

Cloud computing is fast becoming a key instrument in executing workflows. FaaS is a recent development in the field of cloud computing, and it has already incited significant interest in processing workflows. It promises a simple function-oriented execution environment for non-interactive tasks of web applications. Just like with other cloud computing technologies, there are commercial platforms (such as AWS Lambda and Google Cloud Functions) that were developed to provide FaaS functionalities. These allow functions to be executed in environments with a few limitations. First, there are resource limits on CPU, RAM, and temporary storage use. Second, the implemented functions are expected to have stateless behaviour: the execution environment will newly instantiate and terminate for each function invocation (i.e., will not remember state from previous invocations unless some persistence technology is applied). In addition, Amazon Kinesis shard acts as an independent queue that can send workflow tasks to its own function instance.

A number of studies [12, 18, 15] have proved the ability of cloud functions to execute small- and large-scale workflows. In spite of the previously discussed limitations, DEWE v3 have executed workflows even using functions. To avoid the temporary storage use limitation, it uses Amazon S3 to store intermediate workflow data. Therefore, the workflow data needs to be downloaded/uploaded for each function invocation when dependent jobs rely on the output data of other jobs. A large amount of transfer of dependent data can occur during workflow execution between S3 and the FaaS execution environment. Consequently, this could lead to an increased communication costs and a longer makespan.

In this paper, we propose to reduce the dependency transfers in workflows using FaaSs by improving the scheduling algorithm of DEWE v3. Our proposed algorithm exploits the internal queueing mechanisms of Amazon Kinesis shards that feed into AWS Lambda function instances. We choose to move some simple WMS behaviours inside the FaaS. Our approach schedules some dependent jobs on the same shard where their preceding jobs were scheduled. As a result, these dependent jobs can utilise the output files that generated from their precedence constrains in the same invocation. As there is no need for transfers, this step reduces the total workflow execution time as well. Due to Lambda's limitations in terms of temporary storage, the larger files cannot be processed in functions, these we scheduled in a sufficiently sized VM.

We evaluated the proposed and original algorithms with small- and large-scale Montage workflows. The large one is a 6-degree Montage workflow with over eight thousand jobs requiring the transfer of 38 GBs of inputs and outputs. This workflow size was chosen because the original DEWE v3 exhibits a significant amount of re-transfer data behaviour with this workflow. To show the limitations of our approach, we also used a smaller workflow (0.1-degree Montage) that does not have significant amounts of re-transfers even with the original approach.

The proposed algorithm outperforms the original in most cases. Our results show that the proposed approach can reduce the total workflow execution time over the original DEWE v3 approach by about 10%. Our improved scheduling algorithm schedules jobs with precedence constraints on the same shard to be executed in the same Lambda invocation. As a result, it can improve the execution time of scientific workflows on the Lambda platform. In contrast, our approach does not show significant differences in the performance when testing with smaller workflows.

The rest of this paper is organized as follows: the next section presents the background knowledge and related works. Section 3 includes the explanation of DEWE v3 and the proposed algorithm. Section 4 involves the evaluation of our approach with the original algorithm of DEWE v3. Section 5 concludes the paper and suggests some future works.

2 Background Knowledge and Related Works

This section first reviews scientific workflows for scheduling and challenging of real-world experiments as well as simulation frameworks. Then an overview is presented on the most popular FaaS platforms. Finally, the section concludes with a problem statement for the current related works.

2.1 Background Knowledge

A workflow can be formulated as a Directed Acyclic Graph (DAG) that contains a collection of atomic tasks. The nodes are a set of tasks $\{T_1, T_2, ..., T_n\}$ while the edges represent data dependencies among these tasks.

Workflow scheduling is an increasingly important area regarding WMSs. It plays a critical role to achieve an optimal resource allocation for all tasks. The problem of scheduling in distributed environments is known to be NP-hard [20]. Therefore, no algorithms can achieve an optimal solution within polynomial time while some algorithms can provide approximate results in polynomial time.

Running real-world experiments for workflows is a challenge and especially for execution of large-scale. Therefore, WMS simulation has been studied by many researchers using different simulator extensions such as WorkflowSim [7] and WRENCH [6]. WorkflowSim extends the CloudSim [3] simulator, while WRENCH extends the SimGrid [5] framework. However, to date, FaaSs are not simulated in these simulator extensions for running scientific workflows. As a result, we need to restrict our experiments to smaller-scale and larger-scale with considering data transfers, but real-world executions of workflows on commercial FaaSs like Lambda.

Lambda[1] has been presented by AWS in 2014 while cloud functions (GCF[2]) have introduced by Google in 2016. In [12] they stated that Google Cloud Functions, in

[1]`https://aws.amazon.com/lambda/`
[2]`https://cloud.google.com/functions/`

its current form, is not suitable for executing scientific workflow applications due to its limited inbound and outbound socket data quota. There are two benefits when workflows are executed on FaaS systems. First, resource management is provided by the platform in a scalable way. It means the number of concurrent invocations into the infrastructure can more closely follow the actual workflow's demands without the burden on the WMS to deal with the infrastructure's management. Second, due to the nature of the lightweight functions used, the user pays for the much less overheads on computing resource consumption in contrast to more traditional Infrastructure as a Service systems. Lambda functions are stateless, thus their execution environment is initialized and ended for each function invocation. In addition, other commercial solutions also appeared on the FaaS landscape, like Microsoft Azure Functions[3] and IBM OpenWhisk Functions[4].

The above mentioned four FaaS providers were evaluated in [16, 9]. The authors proposed multiple hypotheses concerning the expected performance of cloud functions and designed several benchmarks to confirm them. Their function platforms have tested by invoking CPU, memory, and disk-intensive functions. In addition, data transfer times were also measured for these function providers. They observed different resource allocation policies at the providers. The execution performance of Lambda and GCF is based on the size of memory that is allocated for the invocation. They identified that at the time of writing, Amazon's was more flexible and performant. Moreover, they also reported that computing with cloud functions is more cost-effective than virtual machines due to practically zero delay in booting up new resources. They also indicated that due to the more fine grained invocation patterns to functions virtual machines would have to sit idle in between invocations. This behaviour results in more costs incurred by virtual machine based function oriented solutions. Consequently, we expect more users would prefer Lambda based workflows due to its efficiency and effectiveness comparing with other platforms.

2.2 Related Works

Nowadays, most scientific workflows have been processed in clouds, especially on IaaSs. Only a few related works have studied the use of FaaS platforms to execute workflows. In [17], Malawski et al. proposed five architectural alternatives to run scientific workflows on clouds. One of them introduced a system for serverless computing that integrated the HyperFlow engine with GCFs and AWS Lambda. They examined the viability of running large-scale scientific workflows on cloud functions by evaluating their implementation with a 0.25-degree and a 0.4-degree Montage workflow. They found the approach highly promising. In addition, in [18], they further tested the prototype a 0.6-degree Montage workflow as well. They stopped their experiment at a 0.6-degree workflow as they had faced problems with the temporary storage's 500 MB limitation. However, their approach already exhibits the deficiency of increased transfer of dependent data on these workflows.

[3]`https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview`

[4]`https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-getting-started`

In [12], Jiang et al. designed a WMS called DEWE v3 that can process scientific workflows on three various modes: (*i*) traditional clusters, (*ii*) cloud functions, and (*iii*) a hybrid mode that combines the two. It was tested with large-scale Montage workflows. They have proven that cloud functions can be used in large-scale scientific workflows with complex precedence constrains. However, their job dispatch algorithm schedules jobs to Lambda without considering on their precedence constraints to be executed in the same Lambda invocation. Consequently, more transfer of dependent data can occur during the execution between the storage service and the Lambda invocation's execution environment. This can lead to increased communication costs.

Next, Kijak et al. [15] summarized the challenges for running scientific workflows on a serverless computing platform. They presented a serverless Deadline-Budget Workflow Scheduling (SDBWS) algorithm that was transformed to support function platforms. It was tested with a small-scale 0.25-degree Montage workflow on AWS Lambda. The algorithm used different memory sizes for Lambda based on the deadline and budget constraints assigned by the user. In addition, the function resource is selected depending on the combination of cost and time. This approach was only tested on small scale and likely exhibits transfer of dependent data issues.

In contrast to the above works, [19] proposed an approach which utilised three different cloud function platforms which were Lambda, GCF, and OpenWhisk. They evaluated the platforms with a large-scale (over 5000 jobs in parallel) bag-of-tasks style workflow. The experimental results showed that Lambda and GCF can provide more computing power if one requests more memory, while OpenWhisk's performance is indifferent from this factor. Consequently, they have shown that cloud functions can provide a high level of parallelism for workflows with a large number of parallel tasks at the same time. However, they experimented with a bag-of-tasks approach where they did not consider transfer of dependent data.

In [4], they built Wukong, a new serverless parallel computing framework. It's a cost-effective, serverless, decentralized, locality-aware parallel computing framework. Its key insight is that partitioning the work of a centralized scheduler (i.e., tracking task completions, identifying and dispatching ready tasks, etc.) across a large number of Lambda executors, can greatly improve performance by permitting tasks to be scheduled in parallel, reducing resource contention during scheduling, and making task scheduling data locality-aware, with automatic resource elasticity and improved cost effectiveness. However, their approach already exhibits the deficiency for the data transfers of the precedence constraints between the different jobs of workflow.

3 Our DEWE v3 extension

To uncover the possibilities in dependency transfer optimisation, we have chosen DEWE v3 as a base WMS for our work. Our choice was due to three factors: (*i*) its scheduling technique was closest to our envisioned approach, (*ii*) it is an open source WMS, and (*iii*) it already has the implementation of Lambda as our target execution environment. To understand our extension, we first give a general overview of DEWE v3's behaviour in the following few paragraphs.

DEWE v3 can execute scientific workflows on three different approaches (traditional clusters, cloud functions, and a hybrid mode that combines the two). The FaaS platform supports AWS Lambda and Google Cloud Functions. It has executed large-scale workflows on a hybrid approach that combines traditional clusters with the FaaS platform. DEWE v3 runs a workflow engine on virtual machine. When using AWS Lambda, DEWE v3 reads the workflow definition from an XML file and based on the information found in them loads the job binaries and input files to the object storage Amazon S3. Given that Lambda has a temporary storage limit of 500MBs in the execution environment, some jobs cannot be sent to Lambda due to their large size. Jobs that are ready for execution (i.e., according to their precedence constraints) are scheduled to Amazon Kinesis shards.

Each shard acts as an independent queue that can send tasks to its own function instance. The number of tasks that a function can process in a single invocation is determined by the batch size of Kinesis. This can be configured before the workflow's execution. Next, the Lambda function will pull a batch of tasks from its own shard to execute them sequentially in a single function invocation. The number of running function instances and accompanying kinesis shards are also configurable before the workflow's runtime and this directly influences the maximum level of parallelism the workflow's execution can exhibit.

When a function instance starts to process a job, DEWE v3 needs to download its input data from Amazon S3. Similarly, when the job's processing has finished this must be also uploaded to S3 to make sure other jobs in the workflow can be scheduled due to their input data being ready. This could result in a large amount of transfer dependent data during the execution of the workflow. The transfers take place between S3 and the FaaS environment and directly increase the workflow's communication costs.

To avoid these transfers, we have focused our improvement on the scheduling algorithm of DEWE v3 which targets the Lambda platform as its execution environment. In order to reduce data transfers, during the scheduling, we not only considered the currently ready jobs, but also their successors allowing their sequential execution in a single function instance given that they would not violate Lambda's temporary storage limitation. The next subsection discloses our changes in details.

3.1 The Proposed Scheduling Algorithm

To enhance DEWE v3's data transfers, we moved some workflow management system behaviours inside Amazon's FaaS platform. We exploited the sequencing behaviour of shards and Lambdas. First, some jobs and their successors are scheduled to the same shard and function instance. The ordering of the schedule in the shard is kept in line with the job order in the workflow as prescribed by job precedence constraints. Additionally, we used the *SequenceNumberForOrdering* parameter that guarantees the order of jobs on a shard[5]. This will allow the consecutive jobs to be executed in the same Lambda invocation avoiding the need to transfer outputs and inputs if they are only used in between the given jobs. This behaviour is due to Lambda pulling a batch of jobs based on the batch size of Kinesis to execute them sequentially in an invocation. When the first job in the batch starts its processing, it will read its input data from Amazon S3. We used Amazon S3 because it makes data available through an Internet API that can be accessed anywhere. The intermediate data will be uploaded to S3 that might be needed by other jobs out of batch jobs. Finally, the Lambda will finish processing the batch by uploading the final datafiles to S3 as well.

We have extended the *LambdaWorkflowScheduler* class of DEWE v3 [6]. Our proposed algorithm mainly focuses its changes to the *setJobAsComplete* method, and our changes are depicted in algorithm 1. This algorithm changes the decision on which jobs to schedule at a particular time, while it also alters the shard selection for the jobs that have predecessors. First, we discuss these new choices through the algorithm, then we will disclose two illustrative examples which help to clarify the behaviours even further.

Algorithm 1 shows the pseudo-code of the proposed scheduling algorithm for scientific workflows. We assume that before the application of this algorithm, all jobs without predecessors were scheduled to shards already. Then, this function is invoked by each completed job (T) to release its successor jobs. In step 5, we initialise $jobsNum$ to make sure our allocations of any given shard are balanced in step 10. In step 6, we initialise $alertMax$ which will be used to determine if the current shard received sufficient jobs to fill a complete Lambda invocation batch. Next, in step 7 we initialise the array ($loadBalancing$) that will maintain the job counts on each shard. This will allow us to see if a particular shard is less used and prioritize it for future occasions to equalise the load on all of our lambda instances. Step 9 is the basic behaviour of DEWE v3, where it forgets about jobs that have been completed (called T in our case). This step allows us to determine what job is available to schedule at the moment as jobs without predecessors will become eligible to schedule. In step 10, we choose a shard that has received the minimal number of jobs so far. In step 12, the algorithm checks if the successor job T_i has no more predecessor jobs, then in step 13, the algorithm will schedule T_i to the Kinesis shard determined in the previously discussed step 10. Next, we process all successor jobs (T_j) of our just scheduled T_i. Step 16 checks if T_j has no other predecessor job but T_i. If so, then in Step 17 the algorithm will remove T_i as a predecessor job from T_j (to allow its premature schedule to the same shard that we used for T_i - this is disclosed in Step 18). To ensure the balanced use of all our function instances, step 21-24 checks if we have scheduled sufficient jobs for the next lambda invocation (i.e., the currently selected shard is allocated a complete batch worth of jobs). If so, we don't pursue scheduling any further successors to T_i. We will also remember that we exceeded the batch size of the shard, so the next

[5] `https://docs.aws.amazon.com/kinesis/latest/APIReference/API_PutRecord.html`
[6] `https://github.com/Ali-Alhaboby/DEWE.v3`

Algorithm 1 The proposed scheduling algorithm.

```
Function jobCompleted($T$)
1: $T_i$ = successor job, $T_j$ = dependent job, $KS$ = Kinesis shard
2: $L$ = Lambda instance, $batchSize$ = the batch size of jobs in Lambda
3: $n$ = the number of Lambda instances equals the number of Kinesis shards
4: $m$ = the shard number that has received the minimum number of jobs
5: $jobsNum$ = the number of scheduled jobs to shard.
6: $alertMax$ = alerting the number of scheduled jobs equals to $batchSize$
7: $loadBalancing[n]$ := an array to count the number of sent jobs to each shard
8: for $i = 1, 2, \ldots, p$ do                    // $p$ is the number of successors of $T$
9:    Remove $T$ as a predecessor job from $T_i$
10:   $m$ := find the shard number that has received the minimum number of jobs
11:   $jobsNum := 0$
12:   if $T_i$ has no precedence constraints then
13:      Schedule $T_i$ to $KS_m$ to run in $L_m$
14:      for $j = 1, 2, \ldots, q$ do   // $q$ represents the number of successor jobs of $T_i$
15:         $jobsNum := jobsNum+1$
16:         if $T_j$ has only $T_i$ as a precedence constraint then
17:            Remove $T_i$ as a predecessor job from $T_j$
18:            Schedule $T_j$ to $KS_m$ to run in $L_m$
19:            $jobsNum := jobsNum+1$
20:         end if
21:         if $jobsNum==batchSize$ then
22:            $alertMax$ := true
23:            break
24:         end if
25:         if $alertMax$==true then
26:            $loadBalancing[m] := loadBalancing[m]+jobsNum$
27:            $m$ := find the shard number that has received the minimum number
               of jobs
28:            $alertMax$ := false
29:            $jobsNum := 0$
30:         end if
31:      end for
32:   end if
33: end for
```

shard's schedule can be influenced according to our load balancing rules denoted by steps 26-29. Step 26 maintains the *loadBalancing* array, while step 27 selects a new shard that has received the minimum number of jobs to proceed with the scheduling of further jobs.

To further clarify how the proposed algorithm works, we apply its steps on two simple but carefully selected and crafted sample workflows. Although these

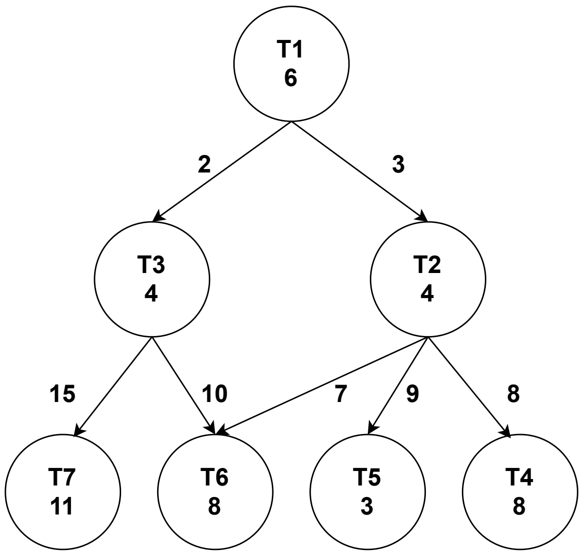

Figure 1: A sample workflow

workflows are simplified, they capture well known DAG patterns that often occur in more complex workflows. As a result, through them, we can demonstrate the applicability of our algorithm to other more complex workflows.

3.2 First illustrative example

In this subsection, we will discuss the workflow fragment, shown in Figure 1. This consists of seven tasks in the graph's nodes: $T1 - T7$. The number inside each task's node represents its estimated execution time (in seconds). On the edges between the nodes, we have also depicted the estimated data transfer time between the storage service (Amazon S3) and the FaaS execution environment.

In the following paragraphs, we will discuss how the original and our new algorithms would be applied to execute the workflow. Before we begin, we will assume the following: (i) there are two Kinesis shards with two Lambda function instances behind that can execute the workflow's jobs; (ii) each invocation downloads/uploads data files sequentially from/to Amazon S3; (iii) Amazon S3 will be used to store all workflow data.

First, the original algorithm would schedule T1. Once T1 completes, it will enable the schedule of T2 and T3 using both available shards. Once they complete, T4, T5, T6 and T7 will be scheduled on two shards as two invocations. Table 1 shows our analysis of the expected execution time with the original algorithm. The colouring of the Table also shows concurrent invocations (i.e., steps coloured the same execute in parallel). When we have parallel invocations, the largest execution

Table 1: The Execution Time (ET) and Transfer Time (TT) of each Lambda invocation of the original algorithm on the sample workflow of Figure 1.

Step	Tasks	ET	TT S3 to FaaS	TT FaaS to S3	Total Time
1	T1	6	-	5	11
2	T2	4	3	24	31
2	T3	4	2	25	31
3	T4, T5	11	17	-	28
3	T6, T7	19	32	-	51
					83

time of the parallel steps will be the component to be considered for the total workflow execution time (i.e., 11s for the white-, 31s for the yellow- and 51s for orange-steps). Finally, for DEWE v3's original algorithm, the Table also discloses our estimated total execution time of 83s in bold.

Now let's compare this approach to our improved scheduling algorithm. We first schedule all tasks that have no predecessor tasks such as T1 which is the same behaviour as before. The commonalities stop here though. Next, when T1 completes, T2 and T3 will become ready. Then, to reduce data transfers, our algorithm will schedule their successor tasks (T4, T5, and T7) as well. It will schedule T2, T4, and T5 on the same shard to be executed in the same function invocation. Also, it will schedule T3 and T7 on the same shard to run on the same invocation. At this time T6 is still left out of schedule because it has two predecessor tasks and we would need both of their outputs before we could start executing T6. Finally, when T2 and T3 complete, they will release T6 to be ready. In Table 2, we computed the Transfer Time (TT) FaaS to S3 in Step 2 because T2 and T3 have a child task T6 which is not scheduled. Therefore, all the data dependency files generated from T2 and T3 need to be uploaded to Amazon S3 in order to make them available to T6. Due to our algorithm's load balancing behaviour, T6 will execute in the same shard T3 and T7 did (as that shard executed the fewest jobs thus far). Similarly to the original algorithm's analysis Table, we have presented our analysis for the new algorithm as well in Table 2. We have concluded that the total workflow execution time of our improved algorithm on this workflow is expected to be significantly better at 68s.

3.3 Second illustrative example

In this subsection, we will discuss the workflow fragment, shown in Figure 1. This fragment has taken from a 0.1-degree Montage workflow that we used in our experiment.

In our second illustrative example, we explain how the proposed algorithm relies on the structure of workflow. We used a workflow fragment that has taken from a 0.1-degree Montage workflow that we used in our experiment. This workflow

Table 2: The Execution Time (ET) and Transfer Time (TT) of each Lambda invocation of the proposed algorithm on the sample workflow of Figure 1.

Step	Tasks	ET	TT S3 to FaaS	TT FaaS to S3	Total Time
1	T1	6	-	5	11
2	T2, T4, T5	15	3	24	42
2	T3, T7	15	2	25	42
3	T6	8	7	-	15
					68

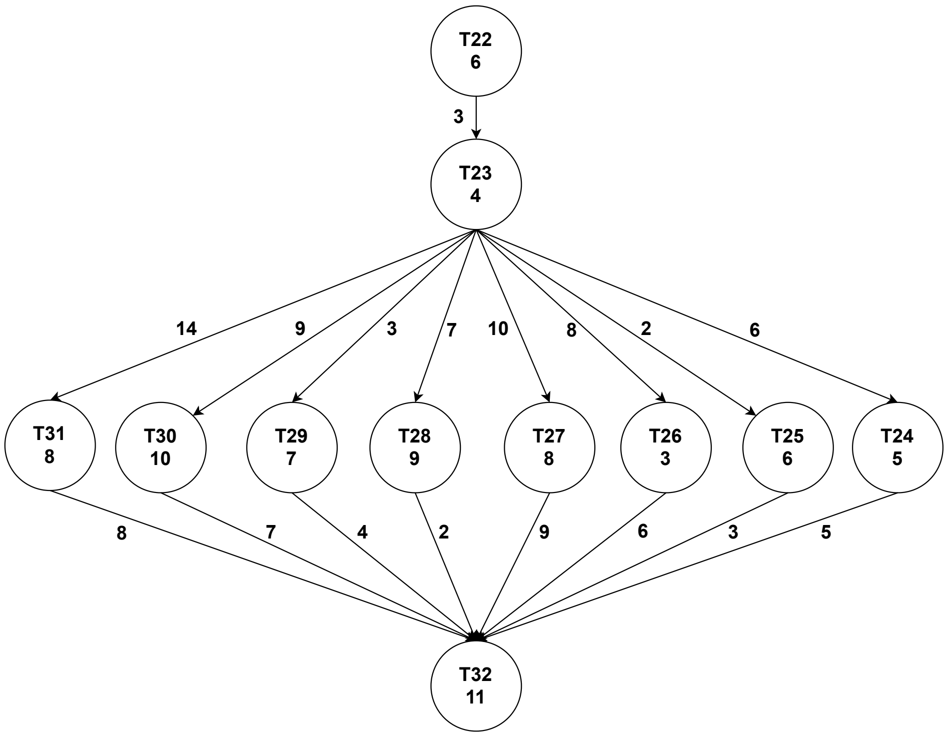

Figure 2: A workflow fragment of a 0.1-degree Montage workflow

(shown in Figure 2) consists of eleven tasks ($T22 - T32$). We will use the same assumptions of the previous example, while also having a batch size of ten. Now we apply both algorithms as follows.

Again, the original algorithm schedules T22 then waits for its completion. Afterwards, it will schedule T23 on one of the two shards. Next, when this task completes, T24-31 will be scheduled on one of the two shards because the batch

Table 3: The Execution Time (ET) and Transfer Time (TT) of each Lambda invocation of the original algorithm on the sample workflow of Figure 2.

Step	Tasks	ET	TT S3 to FaaS	TT FaaS to S3	Total Time
1	T22	6	-	3	9
2	T23	4	3	59	66
3	T24, T25, T26, T27, T28, T29, T30, T31	56	59	44	159
4	T32	11	44	-	55
					289

Table 4: The Execution Time (ET) and Transfer Time (TT) of each Lambda invocation of the proposed algorithm on the sample workflow of Figure 2.

Step	Tasks	ET	TT S3 to FaaS	TT FaaS to S3	Total Time
1	T22	6	-	3	9
2	T23, T24, T25, T26, T27, T28, T29, T30, T31	60	3	59	122
3	T32	11	44	-	55
					186

size of each Lambda instance is 10. Finally, when they complete, they will release T32 to be ready. The total workflow execution time of the original algorithm is estimated to be 289s based on our analysis of Table 3.

With the proposed algorithm a few steps change again. First, as T22 does not have a predecessor, we proceed as the original algorithm. Once it completes, T23-31 will be notified of the completion of one of their predecessors. As our algorithm also schedules successor tasks, T24-31 will also be scheduled to reduce data dependency transfers. All the tasks will be allocated to one of the shards because the batch size of each Lambda instance is 10. They will allocate to the same shard. Finally, when they complete, they will release T32 to be ready. In Table 4, we estimate the total workflow execution time of our algorithm to be 186s which is a significant improvement over the original approach.

With these two illustrative examples we have demonstrated the potential of our algorithm. In the following section, we will evaluate it on both smaller and larger scale real-life workflow executions.

4 Scheduling experiment

In our experiment, we have evaluated our proposed algorithm as well as the original from DEWE v3 on three different approaches (with/without data dependencies on smaller and larger scale). In all three cases, we choose to evaluate through the well known Montage workflow as this makes our results comparable to the previous studies in the related works. Montage is a compute-intensive astronomy workflow for generating custom mosaics of the sky. Montage was also used for different benchmarks and performance evaluation in the past [13]. To ensure good quality data collection, we have repeated all experiments described in this section three times and we reported the average measurement result for each experiment. Each experiment was repeated three times because we obtained the relative consistency of the results by three executions. In addition, we calculated the boxplot visualization that displays the data distribution based on five-number summary (i.e., minimum, first quartile, median, third quartile and maximum) on Figures 3, 4, 5 and 6.

4.1 Evaluation without processing data transfers

First, we have evaluated both algorithms with 2.0 and 4.0 degree Montage workflows (these are medium and larger scale workflows). In this first experiment, we wanted to demonstrate that our algorithmic changes have only negligible influences on the execution time when data transfers play little or no role in a workflow's makespan. Without data transfers our approach should not be able to make its gains. As a result, this experiment can only differ due to execution time circumstances or due to algorithmic changes. This experiment will show the variance of the results without any influence from data transfers. Consequently, we can use the observed differences between the original and the new algorithm as the baseline (i.e., if we see proportionally similar results for the later experiments then the later results would not be significant). The configurations of the experiment are as follows:

1. The Lambda Memory sizes were 512, 1024, 1536, 2048 and 3008 MB
2. The Lambda execution duration limit was 900 seconds.
3. The batch size of the Lambda function was 30.
4. The number of Kinesis shards was set to 5.
5. The VM was t2.micro instance as a free tier with 1 vCPU 2.5 GHz, Intel Xeon Family, and 1 GiB memory.

Figure 3 shows the total execution time of both systems with 2.0-degree workflow on five different memory sizes of Lambda. The differences between the original and the new algorithms have a mean absolute percentage error (MAPE) of 9.96%. While Figure 4 illustrates the total execution time of both systems with 4.0-degree workflow on five different memory sizes of Lambda. In the second case, the MAPE of the total execution time have been calculated as 2.19%. Thus we can conclude

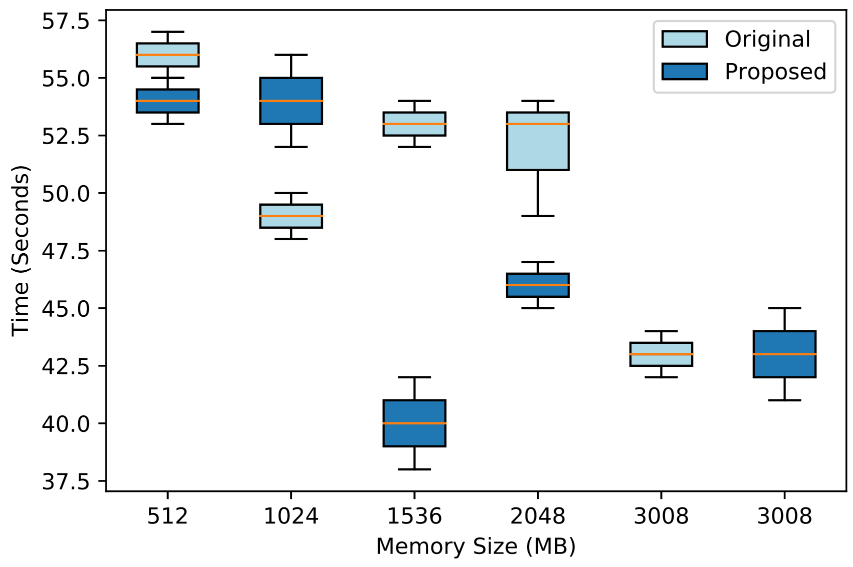

Figure 3: The boxplot visualization of total Execution Time (ET) of both systems with a 2.0-degree Montage workflow without data transfers running on different Lambda memory sizes.

that our changes could manifest in a $\sim$ 6% (average) MAPE. Therefore, in the rest of our experiments results with higher average error values than 6% show that our measurements can be considered as a significant difference. We repeated some memory sizes on the X-axis of Figures 3 and 4 because the boxplot visualization has similar results for both systems.

4.2 Small-scale evaluation

Next, we have evaluated both the original and the new algorithm with a 0.1-degree Montage workflow that also processed its data transfers. We have selected the 0.1 degree one to validate that testing with smaller Montage workflows does not show significant differences with regards to the total execution time (i.e., we show that our approach does not introduce execution time penalties even on smaller workflows where transfers are marginal). The 0.10-degree Montage workflow is sufficiently small for this as it consists of 33 tasks only. The configurations of the experiment are as follows:

1. The Lambda Memory sizes were: 512, 1024, 1536, 2048 and 3008 MB
2. The Lambda execution duration was 900 seconds.
3. The batch size of the Lambda function was 10.

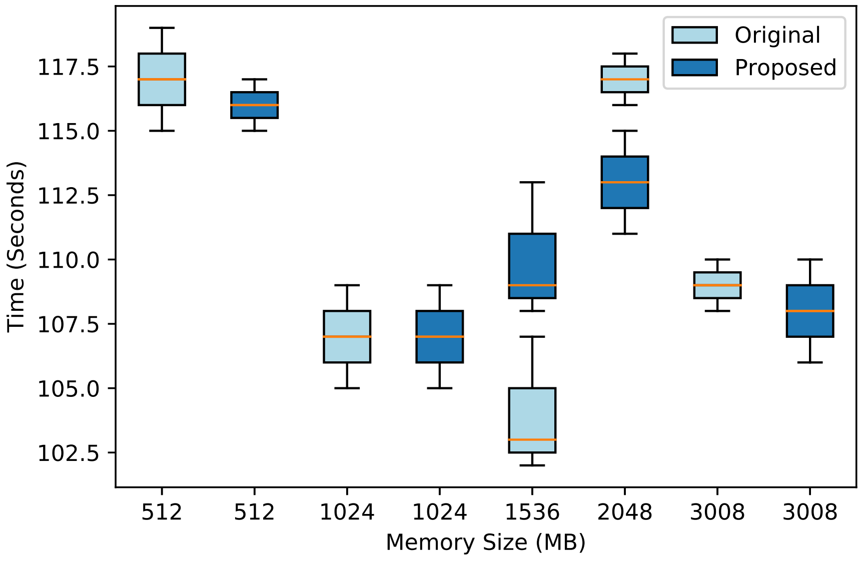

Figure 4: The boxplot visualization of total Execution Time (ET) of both systems with a 4.0-degree Montage workflow without data transfers running on different Lambda memory sizes.

4. The number of Kinesis shards was set to 2.
5. The VM was t2.micro instance as a free tier with 1 vCPU 2.5 GHz, Intel Xeon Family, and 1 GiB memory.

Figure 5 shows the total execution time of both systems with five different memory sizes of Lambda. The MAPE for this series of measurements was 13.95%. This shows that our algorithm has some minimal positive effects already for small-scale workflows as we have arrived to a MAPE value which is over 10% that we have seen in our control experiment in the previous subsection. The results about the lambda with the smallest memory configuration are inconclusive and needs further experimentation to clarify the exact reasons, however it is likely to be caused by the significantly weaker computing performance of those lambda memory configurations.

4.3 Large-scale evaluation

Finally, we have concluded our experiments by evaluating both systems with a 6.0-degree Montage workflow with processing data transfers. This workflow has over eight thousand jobs requiring total data transfers with the size of 38GBs. We have selected this workflow size because in our past analysis, DEWE v3 has already

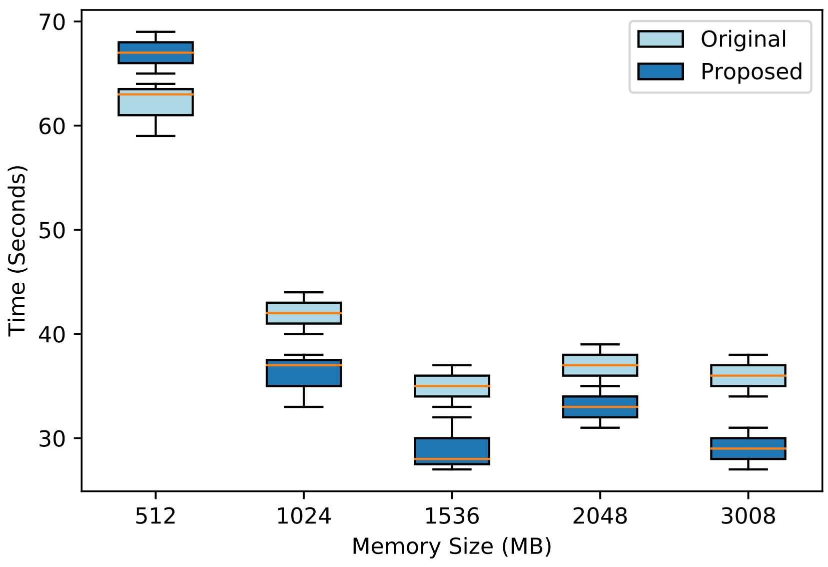

Figure 5: The boxplot visualization of total Execution Time (ET) of both systems with a 0.1-degree Montage workflow with data transfers running on different Lambda memory sizes.

shown a large amount of re-transfer data behaviour. Ideally, our improved DEWE v3 does not have this issue with such large-scale re-transfer-prone workflows. Due to the large expected dependency files of some of the workflow's jobs (namely mAdd), this experiment also used a larger Virtual Machine (VM) alongside the usual lambda functions (as such, all mAdd jobs were executed on the VM). The configurations of the experiment are as follows:

1. The Lambda Memory size was 3008 MB
2. The Lambda execution duration was 900 seconds.
3. The batch size of the Lambda function was 20.
4. The number of Kinesis shards was set to 30.
5. The virtual machine was t2.xlarge that has the following features: 16 GiB of memory and 4 vCPUs.

Figure 6 shows the total execution time of both systems. The proposed algorithm has reduced the total execution time of the large-scale workflow over DEWE v3 by approximately 10%. Thus, this experiment demonstrates that our algorithm is beneficial to be applied for larger scale workflows where the typical data dependency files are still within the 500 MB limit of the Lambda temporary storage limit

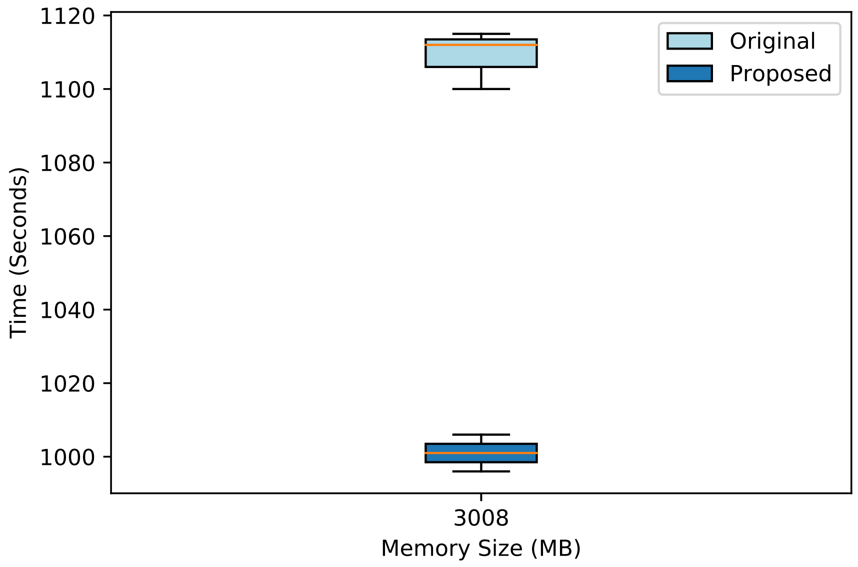

Figure 6: The boxplot visualization of total Execution Time (ET) of both systems with a 6.0-degree Montage workflow with data transfers running on Lambda.

(if this limit would be often breached, the virtual machine count would need to be extended and the cost and elasticity benefits of FaaS systems would be mostly lost). In conclusion both data transfer inducing measurements demonstrate a significantly better result over the original algorithm when we consider the control experiment in subsection 4.1.

5 Conclusion

In this paper, we have changed the job dispatch algorithm of DEWE v3 to reduce its data transfers. The main issue was that DEWE v3 has duplicated data transfers when it executes workflows on FaaSs. It was due to the uploading of intermediate data dependency files after the completion of each function invocation to allow the deletion of temporary files. Otherwise it would fill the Lambda temporary storage space over time because it has an Amazon 500 MB limit. Our proposed algorithm schedules jobs with precedence requirements on the same shard to run in the same function invocation. As a result, the dependent jobs can use the intermediate files that are produced from their predecessor jobs in the same function invocation. We have evaluated our proposed- and the original algorithms with small- and large-scale Montage workflows. Our results show that the improved system can reduce the total workflow execution time of scientific workflows over the original DEWE v3 approach by about 10% when targeting FaaS systems.

In our future work, we will extend the improved system to run on heterogeneous memory sizes of cloud functions to reduce the execution time and cost. In addition, we will study the behaviour of other scientific workflows to make the results more generally applicable. Moreover, we will introduce a Workflow Management System (WMS) simulation for the DISSECT-CF [14] simulator in order to enable the simulation and the execution of scientific workflows on different, reproducible environments. This would foster the creation of more efficient, multi target (i.e., cloud, FaaS, fog etc) workflow scheduling. Finally, we will consider Amazon Elastic File System (EFS) instead of Amazon S3 for storage workflows' data to investigate it in terms of performance, availability, and cost.

References

[1] Abramovici, Alex, Althouse, William E, Drever, Ronald WP, Gürsel, Yekta, Kawamura, Seiji, Raab, Frederick J, Shoemaker, David, Sievers, Lisa, Spero, Robert E, Thorne, Kip S, et al. LIGO: The laser interferometer gravitational-wave observatory. *Science*, 256(5055):325–333, 1992. DOI: `10.1126/science.256.5055.325`.

[2] Altintas, Ilkay, Berkley, Chad, Jaeger, Efrat, Jones, Matthew, Ludascher, Bertram, and Mock, Steve. Kepler: An extensible system for design and execution of scientific workflows. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, pages 423–424. IEEE, 2004. DOI: `10.1109/SSDM.2004.1311241`.

[3] Calheiros, Rodrigo N, Ranjan, Rajiv, Beloglazov, Anton, De Rose, César AF, and Buyya, Rajkumar. CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and experience*, 41(1):23–50, 2011. DOI: `10.1002/spe.995`.

[4] Carver, Benjamin, Zhang, Jingyuan, Wang, Ao, Anwar, Ali, Wu, Panruo, and Cheng, Yue. Wukong: A scalable and locality-enhanced framework for serverless parallel computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 1–15, 2020. DOI: `10.1145/3419111.3421286`.

[5] Casanova, Henri, Giersch, Arnaud, Legrand, Arnaud, Quinson, Martin, and Suter, Frédéric. Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917, 2014. DOI: `10.1016/j.jpdc.2014.06.008`.

[6] Casanova, Henri, Pandey, Suraj, Oeth, James, Tanaka, Ryan, Suter, Frédéric, and da Silva, Rafael Ferreira. Wrench: A framework for simulating workflow management systems. In *2018 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, pages 74–85. IEEE, 2018. DOI: `10.1109/WORKS.2018.00013`.

[7] Chen, Weiwei and Deelman, Ewa. Workflowsim: A toolkit for simulating scientific workflows in distributed environments. In *2012 IEEE 8th international conference on E-science*, pages 1–8. IEEE, 2012. DOI: `10.1109/eScience.2012.6404430`.

[8] Deelman, Ewa, Blythe, James, Gil, Yolanda, Kesselman, Carl, Mehta, Gaurang, Patil, Sonal, Su, Mei-Hui, Vahi, Karan, and Livny, Miron. Pegasus: Mapping scientific workflows onto the grid. In *European Across Grids Conference*, pages 11–20. Springer, 2004. DOI: `10.1007/978-3-540-28642-4_2`.

[9] Figiela, Kamil, Gajek, Adam, Zima, Adam, Obrok, Beata, and Malawski, Maciej. Performance evaluation of heterogeneous cloud functions. *Concurrency and Computation: Practice and Experience*, 30(23):e4792, 2018. DOI: `10.1002/cpe.4792`.

[10] Graves, Robert, Jordan, Thomas H, Callaghan, Scott, Deelman, Ewa, Field, Edward, Juve, Gideon, Kesselman, Carl, Maechling, Philip, Mehta, Gaurang, Milner, Kevin, et al. Cybershake: A physics-based seismic hazard model for southern California. *Pure and Applied Geophysics*, 168(3-4):367–381, 2011. DOI: `10.1007/s00024-010-0161-6`.

[11] Jacob, Joseph C, Katz, Daniel S, Berriman, G Bruce, Good, John, Laity, Anastasia C, Deelman, Ewa, Kesselman, Carl, Singh, Gurmeet, Su, Mei-Hui, Prince, Thomas A, et al. Montage: A grid portal and software toolkit for science-grade astronomical image mosaicking. *International Journal of Computational Science and Engineering*, 4(2), 2009. DOI: `10.1504/IJCSE.2009.026999`.

[12] Jiang, Qingye, Lee, Young Choon, and Zomaya, Albert Y. Serverless execution of scientific workflows. In *International Conference on Service-Oriented Computing*, pages 706–721. Springer, 2017. DOI: `10.1007/978-3-319-69035-3_51`.

[13] Juve, Gideon and Deelman, Ewa. Resource provisioning options for large-scale scientific workflows. In *2008 IEEE Fourth International Conference on eScience*, pages 608–613. IEEE, 2008. DOI: `10.1109/eScience.2008.160`.

[14] Kecskemeti, Gabor. DISSECT-CF: a simulator to foster energy-aware scheduling in infrastructure clouds. *Simulation Modelling Practice and Theory*, 58:188–218, 2015. DOI: `10.1016/j.simpat.2015.05.009`.

[15] Kijak, Joanna, Martyna, Piotr, Pawlik, Maciej, Balis, Bartosz, and Malawski, Maciej. Challenges for scheduling scientific workflows on cloud functions. In *11th IEEE International Conference on Cloud Computing (CLOUD)*, pages 460–467. IEEE, 2018. DOI: `10.1109/CLOUD.2018.00065`.

[16] Lee, Hyungro, Satyam, Kumar, and Fox, Geoffrey. Evaluation of production serverless computing environments. In *11th IEEE International Conference*

on Cloud Computing (CLOUD), pages 442–450. IEEE, 2018. DOI: `10.1109/CLOUD.2018.00062`.

[17] Malawski, Maciej. Towards serverless execution of scientific workflows-hyperflow case study. In *WORKS 2016 Workshop*, pages 25–33. CEUR-WS.org, 2016.

[18] Malawski, Maciej, Gajek, Adam, Zima, Adam, Balis, Bartosz, and Figiela, Kamil. Serverless execution of scientific workflows: Experiments with hyperflow, AWS lambda and Google cloud functions. *Future Generation Computer Systems*, 2017. DOI: `10.1016/j.future.2017.10.029`.

[19] Pawlik, Maciej, Figiela, Kamil, and Malawski, Maciej. Performance considerations on execution of large scale workflow applications on cloud functions. *arXiv preprint arXiv:1909.03555*, 2019. `https://arxiv.org/abs/1909.03555`.

[20] Ullman, Jeffrey D. NP-complete scheduling problems. *Journal of Computer and System sciences*, 10(3):384–393, 1975. `https://core.ac.uk/reader/82723490`.