Batch Queue Resource Scheduling for Workflow Applications

Yang Zhang, Charles Koelbel and Keith Cooper

Computer Science Department
Rice University
Houston, TX 77005
Email: {yzhang8,chk,keith}@rice.edu

Abstract

Workflow computations have become a major programming paradigm for scientific applications. However, acquiring enough computational resources to execute a workflow poses a challenge in a batch queue controlled resource due to the space-sharing nature of the resource management policy. This paper introduces a scheduling technique that aggregates a workflow application into several subcomponents. It then uses the batch queue to acquire resources for each subcomponent, overlapping resource provisioning overhead (wait time) of one with the execution of others. We implemented a prototype of this technique and tested it using five high performance computing centers job submission logs. The results show that our approach can eliminate as much as 70% of the wait time over more traditional techniques that request resources for individual workflow nodes or that acquire all the resources for the whole workflow at once.

I. Introduction

Workflow applications — high-level analyses structured as a set of inter-dependent tasks — have become an essential way for many scientists to compose and execute their computations on high performance computing resources. Workflow applications are widely used in scientific fields as diverse as astronomy [**?**], biology [**?**], [**?**] oceanography [**?**], and earthquake science [**?**].

At the same time, clusters (parallel computers with high-speed interconnects and shared file systems) have become the most common high-performance computing platform. Consequently, workflow applications are often executed on clusters. The workflow execution systems can get access to a cluster either locally, through a collaborative organizations such as TeraGrid [**?**], or through national supercomputing centers like TACC [**?**]. In any case, these clusters are shared and usually managed by a local resource management system that has its own resource sharing methodology and policy. Among them, commercial or open source batch queue scheduling software [**?**], [**?**], [**?**] is the most popular resource management system. Section **??** gives more details on the background of both workflow applications and batch schedulers.

The main goals of a site using batch queues are usually to achieve high throughput and maximize the system utilization. Consequently, many production resources have long queue wait times due to the high utilization levels. In addition, although it is not unusual for a single cluster to have several thousand processors, a single user usually can get only a small portion of the total available resources (without special arrangements). This creates performance problems for large scale workflow applications because each sub-task in the workflow could experience long delays in the job queue before it runs. The queue wait time overhead is sometimes much more than the workflow applications runtime [**?**]. Alternately, one could submit an entire workflow as a single batch queue job. However, this might cause an even longer wait for more resources to become available at once.

Our work seeks to reduce the workflow turnaround time by intelligently using batch queues without relying on reservations. We accomplish this by aggregating workflow tasks together and submitting them as a single job into the queue. Section **??** describes our method in greater detail. This approach can greatly reduce the number of jobs a workflow execution system submits to the batch queue. It also makes smaller placeholder requests than the virtual reservation approach. By overlapping some tasks' wait times with others executions, we further shorten the batch queue wait times for the workflow applications. As we will see in Section **??**, our scheduling reduces the queue wait time overhead while not disturbing the normal batch queue

operation. We conclude our presentation with a discussion of related work in Section **??** and our conclusions and future work in Section **??**.

II. Background

A. Batch Queues

Batch queues have become the most popular resource management method on computational clusters. A batch queue system is normally a combination of a parallel-aware resource management system (which determines "where" a job runs) and a policy based job scheduling engine (which determines "when" a job runs). We are mostly interested in the job scheduler component, treating the individual processors as homogeneous. To illustrate how this scheduler works, we describe the widely-used open-source Maui batch queue scheduler [**?**], [**?**]. The experiments in Section **??** are based on simulations of this scheduler.

The Maui scheduler, like many batch queue schedulers, is essentially a policy based reservation system. The key idea is to calculate a priority for each job in the queue based on aspects of the job and the policy of the queue system. The priority of each batch queue job is determined by job properties, such as the requested resource requirements (number of processors and total time), its owner's credentials, and the time it has waited in the queue. These properties are combined in a formula with weights configured by the system administrator. For example, to favor large jobs, a site would choose a high (and positive) weight for the resource requirements.

After all jobs' priorities are calculated, the Maui scheduler starts all the highest-priority jobs that it can run immediately. It then makes a reservation in the future for the next highest priority job according to the already running jobs' estimated finish time to ensure it will start to run as soon as possible. Given that reservation, a backfill mechanism attempts to find jobs that can start immediately *and* finish before the reservation time. Once a job begins execution, it runs to completion or until it exhausts its requested resources.

Maui, like some other schedulers [**?**], [**?**], [**?**], can provide advance reservation services at a user level. This allows the user to request a specific number of resources for a given period of time, effectively gaining a set of dedicated resources and eliminating the queue wait time. However, advance reservation is not available at all sites, usually involves system administrator assistance, and always requires notice beforehand. Furthermore, Snell et al. [**?**] showed that advance reservation can decrease the system utilization and has the potential to introduce deadlocks. We therefore avoid advance reservations in our work.

One advanced feature of Maui that we do use is the start time estimation functionality. A user can invoke the *showstart* command to get the estimated start time of a job in the queue or a new job (specified with number of processors and duration) to be submitted. This can be done by computing the job's priority, building (or querying) the queue's future schedule, and determining when the job would run. Note that, because new high-priority jobs could be submitted before the queried job runs, the estimate may not be exact. However, it is a useful piece of information to use in scheduling.

B. Workflow Application Execution

A workflow consists of a set of tasks that produce and consume data. The data transfer creates a dependence between the tasks. In scientific applications, there can be hundreds of such tasks, which can range from setting up simulation conditions to performing large computations to visualizing the results. In this paper, we represent such workflows as directed acyclic graphs (DAGs), where nodes represent the tasks and edges represent the dependences. Section **??** has a few examples.

Executing a workflow is conceptually simple. Whenever a node is ready to execute (i.e. all its predecessors have completed), it can be scheduled for execution. However, doing this naively in a batch queue environment could potentially create long waits for every task to begin. Nevertheless, this is common practice. There are two general ways [**?**] (other than advanced reservations) to reduce this batch queue overhead. One way is to aggregate the workflow tasks into larger groups [**?**]. The other method is to use virtual reservation technology [**?**], [**?**], [**?**], [**?**]. This provisioning technique enables users to create a personal dedicated resource pool in a space-shared computing environment. Although there are various implementations, the key idea is to submit a big placeholder job into the space shared resource site. When the placeholder job gets to run, it usually installs and runs a user-level resource manager on its assigned computing nodes. The user-level resource manager (in our case, the workflow execution system) then can schedule jobs onto the those computing nodes without going through the sites resource manager again. Our work draws inspiration from the virtual reservation implementation, but attempts to choose a more propitious size for the placeholder job.

III. Workflow Application Clustering

Traditional DAG clustering algorithms aggregate the workflow tasks into larger unit (to reduce the potential

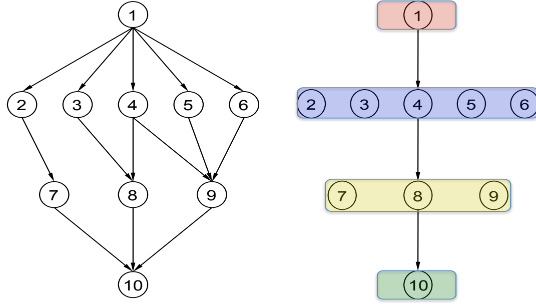


Fig. 1. Workflow Application Cluster

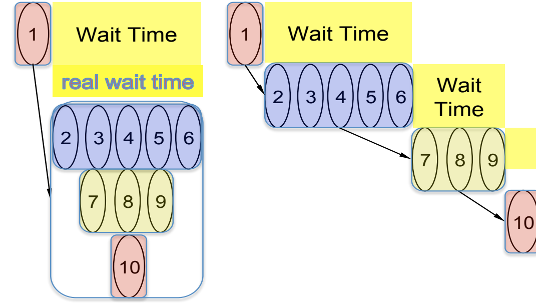


Fig. 2. Workflow Application Cluster by Level

communications). Figure **??** shows an example. The left side of the figure is the original DAG that represents a workflow application. The right side of the figure is a clustered version of the same DAG that we aggregate all the tasks in the same level into one aggregation. Our goal is to choose a clustering algorithm that will reduce the total batch queue wait time. The main idea behind our approach is that we can aggregate the workflow by level and submit a placeholder job for the later levels before their predecessor finishes. In this way, we can overlap the running time of the predecessor level with the wait time of the successor levels.

Figure **??** illustrates this idea. The left shows the possible result of grouping the workflow DAG in figure **??** into two aggregations and submitting them in turn. The yellow rectangles represent the wait time of the two placeholder jobs in the queue. A placeholder job, represented by a rectangle that contains one or more levels of tasks, is submitted into the queue as soon as its predecessor placeholder job starts. It asks for enough resources for the tasks it holds to run in full parallelism. The wait time seen by the users for the clustering on the left is the dark yellow area marked "real wait time". We can see from the figure that it is less than the queue wait time for the second task because of the overlap with task 1's execution. Ideally, if the first placeholder job gets to run immediately and the later jobs' wait times do not exceed their predecessor's run times, the queue wait time for the entire workflow application is eliminated, as shown on right side of figure **??**. However, this perfect overlap cannot be guaranteed. Furthermore, if the wait time for a placeholder job is less than its predecessor's run time (as is the case for task 10), it must pad its requested time to honor its dependences. In turn, this will affect the wait time of the placeholder job. Balancing these effects requires heuristic scheduling.

Our algorithm consists of two interrelated parts: an application manager shown in Figure **??**, and a "peeling" procedure shown in Figure **??**. The application manager is responsible for launching the workflow application and monitoring its progress. In general, it chooses partial DAGs and submits placeholder jobs to the batch queue system. Individual workflow tasks execute in the placeholder jobs when those jobs come to the front of the queue, with the application manager enforcing their dependences. The peeling procedure selects the partial DAGs to minimize the exposed waiting time. We now consider the parts in turn.

Figure **??** shows the application manager. After selecting and submitting the initial partial DAG (lines 1-5), the manager becomes an event-driven system. The primary events that it responds to are:

- A placeholder job starts to run (lines 7-16). The manger first starts all the workflow tasks associated with that job whose predecessors have finished. Then it invokes the peeling procedure to form the next placeholder job and submit it to the queue.
- A placeholder job finishes running (lines 17-25). Normally, no processing is needed. However, if the placeholder is terminated before all its tasks complete (i.e. because some predecessors were delayed in the batch queue), the manager must clean up. It cancels any placeholders that have not started, since some of their predecessors may be delayed. It also calls the peeling procedure to reschedule the unfinished DAG nodes (both interrupted tasks and those not yet run) and submits the new placeholder job into the queue.
- A DAG task finishes (lines 26-32). The manger starts all the successor tasks whose placehold job is already running. One subtlety in the application manager is that the successors of a DAG node may be in the same placeholder or a different one. In the latter case, the manager must handle the possibility that a placeholder starts without any runnable tasks (lines 28-30). If all of a placeholder's tasks are finished, the manger finishes the job to free the batch queue resource.

We choose to submit a new placeholder job only after its predecessor begins running. There are several reasons for this design. In our experience with real queues, we discov-

```
Algorithm:runDAG (DAG dag, int sub_time)
1  task[] partial_dag ← levelize(dag);
2  int count ← 0;
3  Placeholder job ← peelLevel(partial_dag, sub_time, 0);
4  job. name ← count;
5  submit job;
6  while ( dag is not finished)
7  listen to batch queue and task events;
8      if (placeholder job_n starts to run at time t)
9         for all (task in job_n.getTasks())
10            if (all task predecessors have finished)
11               start task;
12        ear_finTime ← job_n.runTime;
13        partial_dag ← levelize(dag.unmappedTasks());
14        job ← peelLevel(partial_dag, t, ear_finTime );
15        job. name ← ++count;
16        submit job;
17     else if ( placeholder job_n finishes running at time t)
18           if ( job_n has unfinished tasks)
19              partial_dag ← levelize(job_n.unfinishedTasks());
20              for all ( pending placeholder job job_m )
21                  cancel job_m;
22                  add job_m.tasks() to partial_dag ;
23              Placeholder jobResub ← peelLevel(partial_dag, t, 0);
24              map all tasks in the partial_dag to jobResub;
25              submit jobResub;
26     else if ( task dagTask finishes running at time t)
27           delete the dagTask from its placeholder job
28           for all (dagTask's successor task chd_task)
29               if (chd_task's associated placeholder job is running)
30                  start chd_task;
31           if (dagTask's placehold Job has no more tasks to run)
32              stop dagTask's placehold Job
```

Fig. 3. The DAG Application Manager

ered that multiple outstanding jobs in the queue interfered with each other. In turn, this often caused the wait time for already-submitted jobs to lengthen, which both added overhead and invalidated our existing schedules. Therefore, we did not have a good estimate of the later placeholder's start time. Although our current design misses the potential of overlapping two placeholder jobs wait times with each other or with running jobs, we can calculate the earliest start time of all the remaining tasks. This is one key to the aggregate decision described in Figure **??**.

Figures **??** shows the peeling procedure used by the application manager. We refer to this process as "peeling" because it successively peels levels of the DAG off of the unfinished work list. First (lines 1-6), the main peelLevel function estimates the wait time to submit the entire DAG as a single placeholder job. It then invokes the groupLevel function (lines 7-8 and Figure **??**) to search for a better alternative. If groupLevel does not improve the wait time (lines 10-13), the peeling procedure chooses to submit the DAG either as a single placeholder job or as one job per task. The decision depends on whether the total wait time

```
Algorithm: peelLevel(levelized DAG, int sub_time, int ear_time)
1   int runTime_all, waitTime_all;
2   int peel_runTime[2], peel_waitTIme[2];
3   runTime_all ← est_runTime(DAG);
4   waitTime_all ← est_waitTime(runTime_all, DAG.width,sub_time);
5   peel_runTime[0] ← runTime_all;
6   peel_waitTIme[0] ← waitTime_all;
7   int level = groupLevel(DAG,sub_time, ear_time,
8                              peel_runTime, peel_waitTIme);
9   if ( level == DAG.height)
10     if (runTime_all * 2 < waitTime_all)
11        return the whole remaining DAG in a batch queue job
12     else
13        return submit the remaining DAG in individual mode
14  else
15     group levels to a partial_dag;
16     map each dag job to the batch queue job;
17     return the partial_dag in a placeholder job;
```

Fig. 4. The DAG Peeling Procedure

as a single job is twice the total run time of the DAG. The intuition for this is that individual submission can take advantage of the free resources or the backfill window. When the one giant placeholder job's wait time is twice as long as the run time, the individual submission has a better chance to finish earlier. This is a heuristic parameter chosen empirically. Otherwise, we use the partial DAG returned by groupLevel. The earliest job start estimation we used is a best effort approach like the *showstart* command in Maui. However, our experience shows it is a reliable indicator of the wait time with one experiment shows the mean difference is within 5% of the real wait time.

Figure **??** shows the key groupLevel procedure. Although the logic is somewhat complex, in essence we perform a greedy search for a aggregation of DAG that has enough granularity to hide later wait times and is wait-effective. We define the wait effectiveness of a job as the ratio between its wait time and its running time; a smaller ratio is better. The intuition behind this is that we want a job to either wait less or finish more tasks. However, we do not search for the globally best wait-effectiveness. This is because, once we group several layers of the DAG into a wait-effective aggregation, any later jobs wait time can be overlapped with run time of this aggregation. Continually adding levels onto the current aggregation negates this benefit.

Here is some more detailed explanation of our algorithm. After some initialization in lines 1-6, the main loop in lines 8-37 repeatedly moves one DAG level from the remaining work to the next placeholder job until the aggregation is less wait-effective than the previous round. For each candidate job, lines 9-18 adjust the placeholder's requested time to allow the workflow tasks to complete.

```
Algorithm: groupLevel (levelized DAG, int sub_time, int ear_sTime,
                       int peel_runTime[2], int peel_waitTIme[2] )
1  int real_runTime[2];
2  int runTime_all, waitTime_all,leeway;
3  runTime_all ← peel_runTime[0];
4  waitTime_all ← peel_waitTIme[0];
5  real_runTime[0] ← peel_runTime[0];
6  partial_dag ← level one of DAG;
7  boolean giant ← true;
8  while partial_dag ! = DAG
9     peel_runTime[1] ← est_runTime(partial_dag)
10    real_runTime[1] ← peel_runTime[1];
11    do
12       peel_runTime[1] ← peel_runTime[1]+ leeway/2;
13       peel_waitTIme[1] ←
14             est_waitTime(peel_runTime[1], DAG.width,sub_time);
15       leeway← ear _sTime + real_runTime[1] - peel_waitTIme[1];
16    while leeway > 10 mins
17    if (leeway > 0)
18       peel_runTime[1] ← peel_runTime[1] + leeway;
19    int real_WaitTime ← peel_waitTIme[1] - ear_sTime;
20    if ( real_WaitTime < 0)
21       real_WaitTime ← peel_waitTime[1];
22    if (giant)
23       if (real_WaitTime > real_runTime[1])
24          add one level to partial_dag;
25          continue
26    giant ← false;
27    if (peel_waitTime[1] - ear_sTime > 0 )
28       if ( peel_waitTime[1] / real_runTime[1]
29                   > peel_waitTime[0] / real_runTime[0] )
30          break;
31       if ( peel_waitTime[1] / real_runTime[1]
32                   > waitTime_all /runTime_all )
33          break;
34    peel_waitTime[0] ← peel_waitTime[1]
35    peel_runTime[0] ← peel_runTime[1]
36    real_runTime[0] ← real_runTime[1]
37    add one level to partial_dag;
38 if (giant)
39    return DAG.height;
40 else
41    return partial_dag.height-1;
```

Fig. 5. The Peel Level decision Procedure

As the left side of figure **??** shows, this is sometimes necessary because the (estimated) queue wait time is less than the time to complete the current job, creating what we term the leeway. A simple iteration adds the leeway to the job request until it is insignificant. (Of course, if the wait time is more than the time to execute predecessors, then no adjustment is needed, as in the right side of figure **??**.) The loop then operates in one of two modes based on whether a good aggregation has been identified. If no aggregation has been selected, more levels are added until the real run time is significant enough to create overlap for the next aggregation(lines 19-25). Once this happens, the current candidate is marked as a viable aggregation. From then on, levels are added only while the wait-effectiveness of the aggregation continues to improve (lines 27-37).

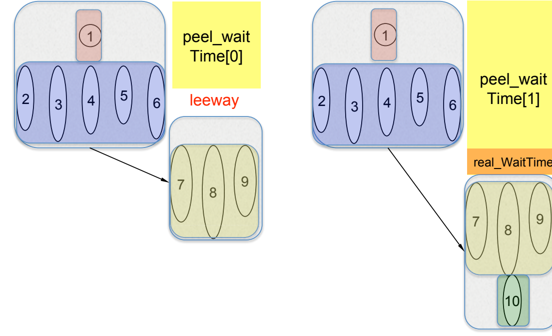


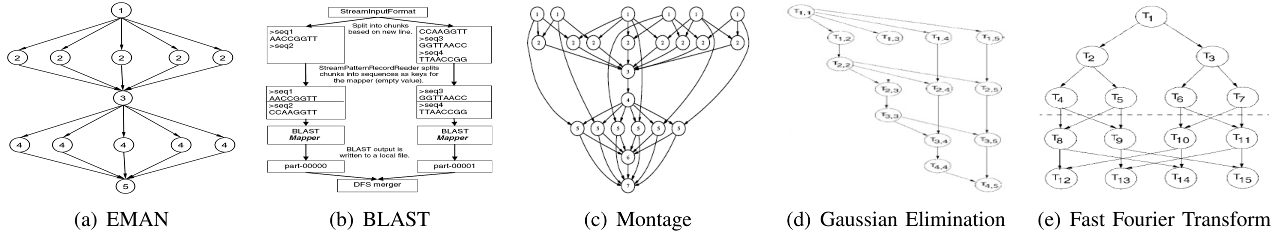
Fig. 6. Workflow Application Level Decision

IV. Experiments

A. Experimental Methodology

To test the performance of our algorithm, we developed a prototype batch queue system simulator that implements the core algorithms of the Maui batch queue scheduler described in [**?**]. The input of the system is a batch queue log obtained from a production high performance computing cluster and a batch queue policy configuration file. It simulates the batch queue execution step by step based on the input. We also implemented the job start time estimation function (the *showstart* command). The estimation is based on the batch queue policy and all the existing queued and running jobs' maximum requested time. It does not forecast any future job submissions. Therefore, it is a best effort estimation within the knowledge of a batch queue scheduler.

We implemented the methods of Section **??** to submit placeholder jobs to this simulator. We also implemented the runtime algorithm in that section, using events generated by our simulator to drive the workflow management. We also implemented two other ways to execute a workflow application on a batch queue based resources. The first is a straightforward way to submit each individual task to the batch queue when it is available to run, which we will refer to as the *individual* submission method. The second is to submit a giant placeholder job that requests enough resources for the entire DAG to finish, which we will refer to as the *giant* submission method. We compare our algorithm, which we will refer to as the *hybrid* submission method, to the individual and giant method by simulating a DAG submission into the queue using different methods with exactly the same experimental configuration.



(a) EMAN (b) BLAST (c) Montage (d) Gaussian Elimination (e) Fast Fourier Transform

Fig. 7. Workflow application DAGs

Cluster	Institution	Batch	Length
Lonestar	Texas Adv. Computing Center	LSF	12 Mon.
Ada	Rice University	Maui	12 Mon.
LeMieux	Pittsburgh SuperComp. Center	Custom	12 Mon.
RTC	Rice University	Maui	12 Mon.
Star	University of Arkansas	Moab	10 Mon.

Fig. 8. The Clusters

B. Experimental Setting

We generate DAG configurations for five high performance computing applications that represent typical parallel computing paradigms, as shown in Figure **??**. EMAN [**?**] is a computational biology application that has two parallel phases connected with single execution steps. BLAST [**?**] is a bioinformatic application that has a sequence of parallel executions. Montage [**?**] is an astronomical application consisting of several inter-leaved layers of parallel executions. We also use two traditional high performance algorithms, Fast Fourier Transform (FFT) and Gaussian elimination. For each application, we generate 25 configurations for different data sizes. The total number of tasks in a workflow ranges from dozens to thousands, maximum parallelism ranges from 5 to 256, and total running time ranges from several hours to a week.

We gathered batch queue logs from four production high performance computing sites with different capacity and batch queue management systems. Figure **??** lists the five clusters we studied at those sites. From each log, we collected all the jobs that finished and their requested number of processors, requested running time, submission time and user id (used only for the user fair share computation). We also obtained the start time and finish time of each job to compute the real job run time. Since most sites don't publish the details of their queuing policy and it can change from day to day, we generate three policies that favored large jobs (*FL*), small jobs (*FS*) or jobs that stay in the queue the longest (*FCFS*). These policies are modified from real site policies which all have a cap value on the resource component of the priority. For example, the FL policy does not assign a higher priory for a large job beyond certain size. Each policy has a queue wait time component which does not have a cap value to avoid starvation. The FCFS policy has a particularly large weight on the wait time component.

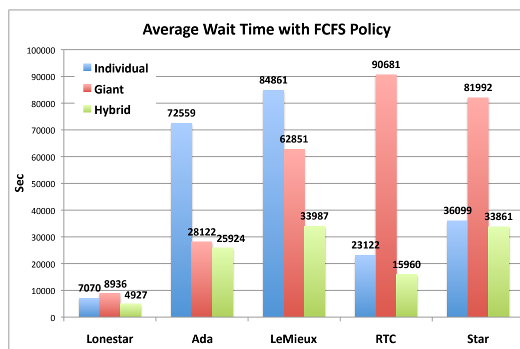
Figure **??** shows our experiment settings. Since the batch queue loads and number of jobs in the queue fluctuate widely, the results of our algorithms depend highly on the time we simulate the submission. Therefore, we run each experimental configuration combination starting at 100 random times during the batch queue log's available time and report the mean results. In total, we ran over 700,000 experiments.

- Algorithms = {individual, giant, hybrid}
- Workflow Application = {EMAN, Montage, BLAST, FFT, Gaussian}
- DAG = { 25 for each workflow application}
- Batch Queue Logs = {Lonestar, Ada, LeMieux, RTC, Star}
- Batch Queue Policies = {FL, FS, FCFS}

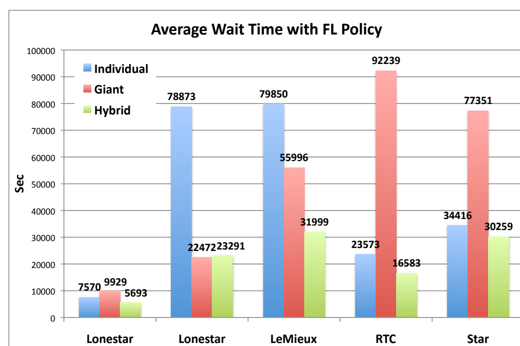
Fig. 9. The Experiment Settings

C. Result Analysis

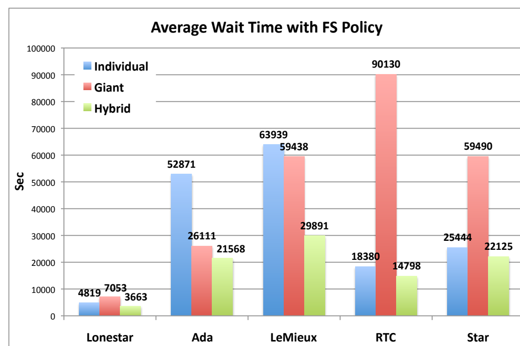
Figure **??** shows the average wait time of all workflow applications on five clusters. All but one of the differences between averages are statistically significant on a two-tailed paired t-test with p-value set at 0.05. We can see that our hybrid scheduling and submission method consistently has the least average wait time among three execution methods. The single exception is on cluster Ada with queuing policy that favors large jobs, and that is the only statistical tie. In addition, our results indicate that although the batch queue policy determines each batch queue job's priority, it does not affect our experiment significantly. However, the average wait time from each cluster varies greatly. For example, the average application wait time on the Lonetar cluster is only a fraction of the other four clusters. Furthermore, while the individual submission method waits significantly more time on the Ada and



(a) FCFS Policy Results



(b) FL Policy Results



(c) FS Policy Results

Fig. 10. Overall Average Wait time

LeMieux clusters than the giant method, it waits much less time than the giant method on the RTC and Star clusters.

Since we ran the same set of experiment on each cluster, we hypothesized that the differences in the outcomes were the results of each cluster's unique combination of its configuration and usage pattern. Therefore, we further analyzed the characteristics of each cluster's batch queue jobs. We calculated the number of jobs submitted each day, processors a job requests, time a job runs, the CPU hours a job requests, the actual load and the requested load of the system over the duration of each log file. The actual load is calculated by dividing the total CPU hour used by the cluster's maximum capacity and the requested load is calculated by using the total CPU hour requested. Figure **??** presents each cluster's configuration and our calculations. The results clearly show each cluster has its own unique usage pattern, and we can use this to explain the variance in our experiment results. For example, Lonestar cluster has the largest computing capacity among the five clusters. This explains why the average wait time of workflow application on Lonestar is much less than on the other clusters since the it's much easier for Lonestar cluster to fulfill the resource demand of the same workflow application than other clusters. The batch queue usage pattern can also affect the execution results in more subtle ways.

Figure **??** shows that the Ada cluster users tend to submit small jobs both in terms of processors and CPU hours. However, the Ada's actual load is not particularly light and it has a large number of jobs get submitted each day. This explains why the giant method is more effective on Ada than the individual method when the queue policy favors large job, see figure **??**. It is because the giant placeholder job would usually be the job with the highest priority in the queue and thus could start early. On the other hand, the individual job submission is less effective not only because the queue policy favors large jobs but also, since most jobs in the queue are small jobs, there are fewer opportunities to schedule an individual job by backfilling. However, figure **??** does not show a very clear picture of why the giant method still performs relatively well when the policy favors small jobs (although the difference is much less). Figure **??** depicts more clearly the effect of the queue policy on the outcome for each method. We calculated the average of the *relative* wait time In figure **??** by dividing each application's wait time by its running time before we computed the mean. In this way, we give each workflow's wait time an equal weight in the final result. Now, we can see that giant method actually performs worse when the queue policy favors small jobs in terms of relative wait time. Nevertheless, our hybrid method performs the best in terms of relative wait time under all three queue policies since it uses feedback from the batch queue scheduler.

We can also deduce from Figure **??** that the users of the Star cluster request long run times but not as many processors. In addition, we notice that the average requested load on Star is almost five times more than the actual load, the highest among all clusters we tested. This means the Star users tend to request many more CPU hours than they actually use. This can partially explain why the individual submission method works well on Star since the system reserves resources for the next highest priority job by basing its start time on the running jobs' requested time. When a job finishes early, it creates a backfill window, so

Cluster	Cluster Size	Mean Jobs per Day	Mean Job Width	Mean Job Run Time	Mean Job Request Size	Actual Load	Request Load
Lonestar	5000 core	932	26.18 core	3.03 hour	274 hour	0.81	2.13
Ada	520 Core	1342	3.57 core	3.57 hour	25 hour	0.81	2.76
LeMieux	2048 Core	251	43.80 core	3.30 hour	329 hour	0.91	1.68
RTC	270 Core	108	2.43 core	13.69 hour	112 hour	0.57	1.87
Star	1200 Core	108	13.16 core	16.93 hour	1050 hour	0.83	3.94

Fig. 11. Cluster Configuration and Batch queue Job Characteristic

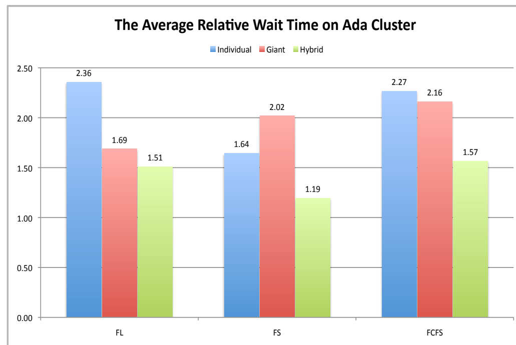


Fig. 12. The Effect of Queue Policy on Ada

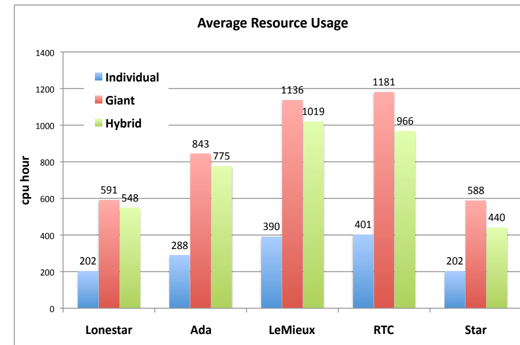


Fig. 13. The CPU Hour Usage

Star would have many backfill opportunities based on its usage pattern. Small jobs, as generated by the individual method, are more likely to be able to use these backfill slots. However, this does not explain why the giant method works better under a queue policy that favors small jobs on Star cluster.

We computed the average resource usage for our workflow applications on the clusters with FS queue policy. The resource usage for a workflow application is the sum of the actual running times for all placeholder jobs submitted into the queue. The wait time is not included. Figure **??** shows that the giant submission method uses almost three times more resources than individual method while our hybrid submission method uses 10-20% less than the giant method. In both the hybrid and giant method, the additional CPU usage is mainly due to resources allocated to the placeholder according to the level with the maximum parallelism but not used on the other levels. On the Star cluster, we can see the average giant placeholder job uses less than 600 CPU hours while Figure **??** shows the average job on Star requests over 1000 CPU hours. This means the giant jobs are actually small compared to other jobs' requests (although, again referring to Figure **??**, not their actual run time). This explains why all the execution methods work better under the queue policy that favors small jobs on the Star cluster. At same time, we can see that the idle processor overhead for both the giant and hybrid methods can be substantial. Despite the large job size and inaccurate job request on the Star cluster, our hybrid method again has the lowest mean wait time.

Figure **??** also explains the giant method's ineffectiveness on the small RTC cluster. When virtual reservations in the giant method request more than 128 processors (which about 30% of the total workflows do), it takes more than half the cluster. Even when the queue policy favors large jobs, such a job cannot run until all the already running jobs on RTC finish. Figure **??** presents the average wait time of the workflows that require less (small DAG) or more (large DAG) than 128 processors on the RTC cluster. It shows the giant method indeed suffers the most when a single workflow application requires too much of the entire cluster. The same would be true for placeholders generated by the hybrid method, but the estimated wait times prevent our scheduler from generating such pathologies. As a result, our hybrid method outperforms both giant and individual under any policy on the RTC when the DAGs are small and it virtually submits all the big DAGs in individual mode.

Figure **??** shows why the hybrid method performs the best on the LeMieux cluster. We can see that the LeMieux cluster's ratio of requested load to actual load is the lowest, which means that users do a good job in estimating their jobs' running time. That greatly improves the accuracy of the batch queue start time estimation and in turn reduces the opportunities for individual jobs to be backfilled. In short, the individual method has no leverage to schedule

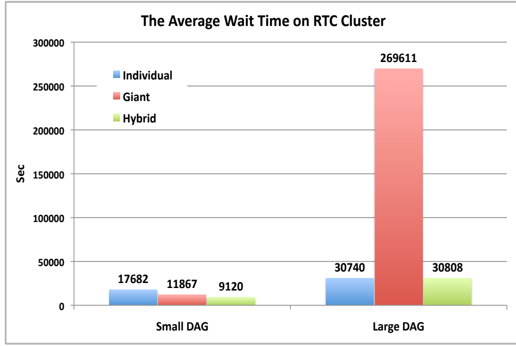


Fig. 14. The Average Wait Time of Small DAGs on RTC Cluster

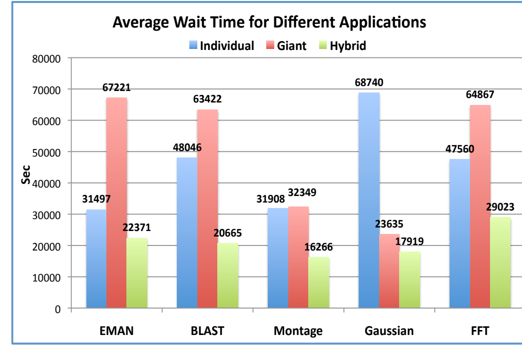


Fig. 15. The Average Wait Time for Different Applications

its small tasks. On the other end of the spectrum, the accurate wait time estimation helps the hybrid method avoid submitting large requests that would endure long waits, as the giant method is prone to do. As a result, we see a better advantage for the the hybrid method on LeMieux than any other cluster.

The type of workflow application can also affect the performance of the execution methods. Figure **??** shows the average wait time of the five workflows we tested averaged across all the clusters under the FL policy. While the giant method is best for Gaussian elimination, it is worst for the other four applications. The difference lies in the application configuration as shown in figure **??**. The Gaussian elimination workflow has the most levels relative to the number of tasks among our test cases. For example, EMAN and Montage both have a constant number of levels, and FFT grows logarithmically to a total of level 20 in our test while the longest Gaussian DAG has over 100 levels. Since the tasks in the individual submission method have to wait for the previous level of task to finish before it can be submitted into the queue, there are more stalls for the Gaussian workflow than other applications. Another reason is the maximum parallelism for a Gaussian placeholder is 55 while other applications have up to 256 in our experiment settings. As we saw in Figure **??**, the giant method performs better than the individual method when a DAG's maximum parallelism is small relative to the cluster size. The giant method results on RTC cluster alone increase the average wait time for all the applications but the Gaussian workflow. Again, we see that our hybrid algorithm consistently has the least wait time for any workflow applications we tested.

V. Related Work

Brevik et al. [**?**] provided upper bound prediction of the queue wait time for an individual job. They used a binomial model and historical traces of job wait times in the queue to produce a prediction for a user specified quantile at a given confidence level without knowing the exact queuing policy of the resource. We use the estimate provided by the system itself, but in principle we could use any predictor.

There are several techniques for a user to reserve resources in a batch queue system without using the system's advanced reservation function. Condor glide-in [**?**] is used to create condor [**?**] pools in a remote resource. Nurmi et al. [**?**] implemented probabilistic based reservations for batch-scheduled resources. The basic idea is to use their wait time prediction [**?**] to choose when to submit a job so that it runs at a given time. Walker et al. [**?**] developed an infrastructure that submits and manages job proxies across several clusters. A user can create a virtual login session that would in turn submit the user's jobs through a proxy manager to a remote computing cluster. Kee et al. [**?**] developed a virtual grid system that allows a user to specify a number of resource reservations. Our work is inspired by these techniques to get a personal cluster from a batch queue controlled resource for each aggregation of tasks in the workflow application.

Limited research has been done on scheduling a workflow application on a batch queue controlled resources. Nurmi et al. [**?**] took into account the queue wait time when each individual task in a workflow application is scheduled. Singh et al. [**?**] demonstrated the effectiveness of clustering a workflow application using the Montage [**?**] application. Our approach builds on top of their ideas by dynamically choosing the clustering for the workflow, whereas they use static mappings.

VI. Conclusions and Future Work

In this paper, we presented an algorithm that clusters a workflow application and submits them when the previous aggregation begins to run in the batch queue. The aggre-

gation granularity is computed so that it can minimize the total wait time experienced by the workflow by overlapping most of the wait time and running time between the aggregations. By using system-provided estimates of the current queue wait time, we were able to substantially improve turnaround time over standard strategies of submitting many small jobs or a single large job. The results that we collected from running over half a million experiments using logs from five production HPC resources showed that our hybrid execution method consistently results in less overall wait time in the batch queue. We were able to accomplish this without modifying the site policies or software.

Not every batch queue resource management softwares provide the earliest job start time estimation yet so in the future we would like to integrate this feature into open source systems. Moreover, we believe that providing support for workflow DAGs directly in the batch queue software would be a valuable service to users, particularly when coupled with intelligent scheduling techniques such as those we have presented.

Acknowledgments

This material is based on work supported by the National Science Foundation under Cooperative Agreement No. CCR-0331645 (the VGrADS Project). This work was supported in part by the Shared University Grid at Rice funded by NSF under Grant EIA-0216467, and a partnership between Rice University, Sun Microsystems, and Sigma Solutions, Inc. We would also like to thank Roger Moye from Rice University, Jeff Pummill, Dr. Amy Apon, Wesley Emeneker from University of Arkansas, Rich Raymond, Chad Vizino from Pittsuburgh Supercomputing Center and Warren Smith from Texas Advanced Computing Center for providing the batch queue log data.

References

[1] Brett Bode, David M. Halstead, Ricky Kendall, Zhou Lei, and David Jackson. The portable batch scheduler and the maui scheduler on linux clusters. In *ALS'00: Proceedings of the 4th annual Linux Showcase & Conference*, pages 27–27, Berkeley, CA, USA, 2000. USENIX Association.

[2] John Brevik, Daniel Nurmi, and Rich Wolski. Predicting bounds on queuing delay for batch-scheduled parallel machines. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 110–118, New York, NY, USA, 2006. ACM.

[3] Inc. Cluster Resources. http://clusterresources.com/.

[4] Ewa Deelman, Scott Callaghan, Edward Field, Hunter Francoeur, Robert Graves, Nitin Gupta, Vipin Gupta, Thomas H. Jordan, Carl Kesselman, Philip Maechling, John Mehringer, Gaurang Mehta, David Okaya, Karan Vahi, and Li Zhao. Managing large-scale workflow execution from resource provisioning to provenance tracking: The cybershake example. In *E-SCIENCE '06: Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, page 14, Washington, DC, USA, 2006. IEEE Computer Society.

[5] James Frey, Todd Tannenbaum, Miron Livny, Ian Foster, and Steven Tuecke. Condor-g: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[6] Bill Howe, Peter Lawson, Renee Bellinger, Erik Anderson, Emanuele Santos, Juliana Freire, Carlos Scheidegger, António Baptista, and Cláudio Silva. End-to-end escience: Integrating workflow, query, visualization, and provenance at an ocean observatory. In *ESCIENCE '08: Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pages 127–134, Washington, DC, USA, 2008. IEEE Computer Society.

[7] David B. Jackson, Quinn Snell, and Mark J. Clement. Core algorithms of the maui scheduler. In *JSSPP '01: Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing*, pages 87–102, London, UK, 2001. Springer-Verlag.

[8] Gideon Juve and Ewa Deelman. Resource provisioning options for large-scale scientific workflows. *eScience, IEEE International Conference on*, 0:608–613, 2008.

[9] Yang-Suk Kee, C. Kesselman, D. Nurmi, and R. Wolski. Enabling personal clusters on demand for batch resources using commodity software. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–7, April 2008.

[10] Arun Krishnan. Gridblast: a globus-based high-throughput implementation of blast in a grid computing framework: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(13):1607–1623, 2005.

[11] Load Sharing Facility (LSF). http://www.platform.com/.

[12] S. Ludtke, P. Baldwin, and W. Chiu. EMAN: Semiautomated software for high resolution single-particle reconstructions. *J. Struct. Biol*, pages 82–97, 1999.

[13] W. Gentzsch (Sun Microsystems). Sun grid engine: Towards creating a compute power grid. In *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 35, Washington, DC, USA, 2001. IEEE Computer Society.

[14] Daniel Nurmi, Anirban Mandal, John Brevik, Chuck Koelbel, Rich Wolski, and Ken Kennedy. Evaluation of a workflow scheduler using integrated performance modelling and batch queue wait time prediction. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 119, New York, NY, USA, 2006. ACM.

[15] Daniel Charles Nurmi, Rich Wolski, and John Brevik. Varq: virtual advance reservations for queues. In *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*, pages 75–86, New York, NY, USA, 2008. ACM.

[16] Open PBS. http://www.openpbs.org/.

[17] Condor Research Project. http://www.cs.wisc.edu/condor.

[18] G. Singh, E. Deelman, and G. Bruce Berriman et al. Montage: a Grid enabled image mosaic service for the National Virtual Observatory. *Astronomical Data Analysis Software and Systems*, 2003.

[19] Gurmeet Singh, Mei-Hui Su, Karan Vahi, Ewa Deelman, Bruce Berriman, John Good, Daniel S. Katz, and Gaurang Mehta. Workflow task clustering for best effort systems with pegasus. In *MG '08: Proceedings of the 15th ACM Mardi Gras conference*, pages 1–8, New York, NY, USA, 2008. ACM.

[20] Quinn Snell, Mark J. Clement, David B. Jackson, and Chad Gregory. The performance impact of advance reservation meta-scheduling. In *IPDPS '00/JSSPP '00: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 137–153, London, UK, 2000. Springer-Verlag.

[21] TeraGrid. http://www.teragrid.org/about.

[22] Texas Advanced Supercomputing Center. http://www.tacc.utexas.edu/.

[23] E. Walker, J.P. Gardner, V. Litvin, and E.L. Turner. Creating personal adaptive clusters for managing scientific jobs in a distributed computing environment. In *Challenges of Large Applications in Distributed Environments, 2006 IEEE*, pages 95–103, 2006.