A Planner-Guided Scheduling Strategy for Multiple Workflow Applications

Zhifeng Yu and Weisong Shi

Wayne State University

{zhifeng.yu, weisong}@wayne.edu

Abstract— **Workflow applications are gaining popularity in recent years because of the prevalence of cluster environments. Many algorithms have been developed since, however most static algorithms are designed in the problem domain of scheduling single workflow applications, thus not applicable to a common cluster environment where multiple workflow applications and other independent jobs compete for resources. Dynamic scheduling approaches can handle the mixed workload practically by nature but their performance has yet to optimize as they do not have a global view of workflow applications. Recent research efforts suggest merging multiple workflows into one workflow before execution, but fail to address an important issue that multiple workflow applications may be submitted at different times by different users. In this paper, we propose a planner-guided dynamic scheduling strategy for multiple workflow applications, leveraging job dependence information and execution time estimation. Our approach schedules individual jobs dynamically without requiring merging the workflow applications *a priori*. The simulation results show that the proposed algorithm significantly outperforms two other algorithms by 43.6% and 36.7% with respect to workflow makespan and turnaround time respectively, and it performs even better when the number of concurrent workflow applications increases and the resources are scarce.**

I. Introduction

Scheduling workflow applications in a cluster environment is a great challenge. A workflow application is typically represented as a direct acyclic graph (DAG), where nodes represent individual jobs and edges represent the inter-job dependence. Its performance is generally measured by makespan, the time difference between the start time and completion time of a workflow. We use DAG and workflow application interchangeably in this paper.

At a high level, DAG scheduling algorithms can be divided into two groups: *static* and *dynamic*. A static algorithm presumes knowledge of the whole structure of a DAG and its job execution time estimation, and resource mapping is made on DAG level before the execution. Conversely, a dynamic algorithm makes a decision only when an individual job is ready to execute. Comparative studies [1], [2], [3] from different perspectives show that static algorithms outperform dynamic ones in most cases.

However, static algorithms do not consider the real world situation where a cluster serves mixture of multiple workflow applications and other independent jobs. All of the 27 static algorithms surveyed in [4] and other ones [5], [6], [7] devised later are restricted to single DAG scheduling. Recent efforts [8], [9] attempt to schedule multiple DAGs, but they merely merge multiple DAGs into one unified DAG *a priori* and schedule as a single one. But this approach is not feasible when DAGs are submitted by different users at different time.

This work is in part supported by National Science Foundation CAREER grant CCF-0643521.

On the other hand, dynamic algorithms can handle multiple DAGs in a natural way because of their intrinsic adaptability to the dynamics of both workload and environment. From the scheduler's point of view, a ready-to-execute individual job within a DAG is no different than other ordinary independent jobs waiting in the queue. As the job interdependence is transparent to the scheduler, it can handle one or many workflow applications.

When a user submits a workflow application, a key question he or she wants to ask is what the turnaround time will be, which is measured by the time difference between submission and final completion of the application. In addition, the makespan is used to measure the workflow application performance. From a system management perspective, the concern is the overall resource utilization and throughput. While existing dynamic algorithms support dynamic workload allowably consisting of multiple DAGs, their performance is not yet evaluated and comparatively studied with any static counterpart to the best of our knowledge. Given the historic performance evaluation on single DAG scheduling [1], [2], [3], [10], it is not hard to envision that even with multiple DAGs the dynamic algorithms can be optimized if the DAG structure and job execution estimation are taken into account.

In this paper, we propose a planner-guided dynamic scheduling algorithm for multiple workflow applications in a cluster environment, inheriting both adaptability of dynamic approaches and performance advantages of static ones. With this approach, the workflow planner helps the executor to prioritize jobs globally across multiple DAGs so that the executor is able to assign the job of highest priority to the best resource to achieve better performance. More importantly, it is a practical solution capable of plugging into a real world workflow management system, and it can also be extended to grid environments with applications of mixed varieties. The contributions of this paper are:

1) Propose a planner-guided priority based dynamic scheduling algorithm for scheduling multiple workflow applications in a cluster environment;
2) Evaluate the performance of proposed algorithm using a published comprehensive test bench [11] and study

its effectiveness with respect to number of static and dynamic parameters; and
3) Observe that *RANDOM* and *FIFO* have very similar performance while the proposed algorithm improves the average makespan and turnaround time over the former two by 43.6% and 36.7% respectively.

The rest of the paper is organized as follows. Related work is discussed in Section II. Then we describe the planner-guided dynamic scheduling algorithm in Section III. Section IV elaborates the experiment design and evaluates the performance of proposed algorithm along with other popular ones. Finally, we summarize and lay out the future work in Section V.

II. Related Work

The DAG scheduling problem has drawn extensive attention in the last two decades and various algorithms have been proposed in literature with objective of achieving near optimal performance of single DAG. Several of them, static or dynamic, have been successfully implemented in number of workflow management systems like DAGMan [12], ASKALON [1], GidFlow [13], Pegasus [14], Taverna [15], GrADS [16], and so on.

By its nature a dynamic algorithm can support multiple DAGs as scheduling decision is made only when the individual job is ready to execute. For example, DAGMan [12] manages jobs of a DAG in an internal queue reflective of the job interdependence. When jobs in the internal queue become ready they will be forwarded to the Condor queue which contains jobs from different users. Each job can be assigned with user defined priority, otherwise an FIFO order will be used to schedule next available job in the Condor queue [17].

Iverson *et al.* [19] present a hierarchical matching and scheduling framework where multiple workflow applications compete for resources. A decentralized scheduling strategy is proposed where each application makes its own scheduling decision during the allocated time slots. Different scheduling time policies are compared for their impacts on overall resource utilization, but the discussion does not cover measurement of average turnaround and makespan, which are of most interest from the user's perspective.

Several static algorithms are also proposed for multiple DAG scheduling. Zhao *et al.* [8] propose composition approaches to merge multiple DAGs into a single DAG first before applying an algorithm designed for fairness. Similarly Hönig *et al.* [9] devise a meta-scheduler for multiple DAGs, which suggests to merge multiple DAGs into one to improve the overall parallelism. However, these efforts are limited by inability to deal with dynamics of workloads, i.e., multiple DAGs may come at different time.

In this paper, we envision that the key practical issue of scheduling multiple DAGs is that they may come in at different time dynamically. With the observation of adaptivity of dynamic strategies and potential benefits and serious limitations of static ones, we propose a planner-guided dynamic scheduling approach in order to deal with dynamic workload by leveraging on advantages of both approaches.

III. Planner guided dynamic scheduling

Inspired by some effective algorithms in the single DAG scheduling problem domain and the fact that workflow applications are gaining more popularity because of the prevalence of cluster environments recently, we explore practical solutions to the real world challenges of scheduling workflow applications in a cluster environment. In this section, we first discuss the motivation of our research, then describe how we approach the solution and finally define the proposed algorithm in detail.

A. Motivation

There have been extensive comparative studies for static and dynamic algorithms [1], [2], [3], [4], [10] in the context of scheduling single DAG. It is well concluded that static approaches outperform dynamic ones in most cases when resource and workload makeup do not vary over time. Nevertheless, the impracticality of static scheduling algorithms are recognized in researches [20], [21].

The design in our previous work [21], which builds collaboration between workflow *Planner* and *Executor* to adapt to a dynamic environment, can be extended to tackle dynamic workload issue as well. The *Planner* can facilitate the *Executor* to make wiser decisions with the information of DAG structure and job execution estimation available, including

1) *Use ready job pool to manage dynamic workloads.* The key difference between workflow applications and ordinary jobs is the job interdependence. If the workflow *Planner* submits individual jobs to the job pool only when they become ready to execute, the job dependence constraint is transparent to the *Executor*. The *Executor* applies a scheduling algorithm on the jobs in the pool, even though these jobs may belong to different DAGs. This mechanism offers the capability to schedule multiple workflow applications which arrive dynamically.
2) *Globally prioritize jobs in the pool to optimize dynamic scheduling performance.* The *Executor* can optimize the performance if it knows which job has the biggest impact on overall performance and therefore should be scheduled with highest priority. Each individual job is assigned priority locally by the *Planner* before being placed in the job pool, reflective of the job interdependence constraints and job execution time estimation. When the jobs from different users make the pool, the local priority of each job can be utilized as a base to form global priorities in the pool. The global priority of a job in the pool may change dynamically as the composition of the pool changes in real time.

B. Planner-guided dynamic scheduling

Most static scheduling algorithms are based on the so-called list scheduling technique, where a job list is maintained in order of priority [4]. HEFT(Heterogenous Earliest-Finish-Time) [5] is one of the most popular ones, and its effectiveness is proved by implementation in ASKALON [1] project. We will use the upward ranking mechanism introduced in [5] to locally prioritize jobs in a DAG.

A workflow application is represented by a direct acyclic graph, *G=(V, E)*, where V is the set of v jobs (nodes) and E is the set of e edges between jobs. Each edge $(i, j) \in E$ represents the precedence constraint such that job n_i should complete its execution before job n_j starts. $data$ is a $v \times v$ matrix of communication data, where $data_{i,k}$ is the amount of data required to be transmitted from job n_i to job n_k. The upward rank of a job n_i is recursively defined, starting from the job n_{exit}:

$$rank_u(n_i) = \overline{w_i} + \max_{n_j \in succ(n_i)} (\overline{c_{(i,j)}} + rank_u(n_j)) \tag{1}$$

where $succ(n_i)$ is the set of immediate successors of job n_i, $c_{(i,j)}$ is the average communication cost of edge (i, j), and $\overline{w_i}$ is the average computation cost of job n_i. For the exit job n_{exit}, the upward rank value is defined as

$$rank_u(n_{exit}) = \overline{w_{exit}} \tag{2}$$

In order to schedule dynamically and optimize the resource allocation decision, the proposed system consists of three core components: *DAG Planner*, *Job Pool* and *Executor*. The *DAG Planner* assigns each individual job local priority as defined above, manages the job interdependence and submits jobs whenever they are ready to execute into the *Job Pool*, which is an unsorted set containing all jobs from different users waiting to be scheduled. The *Executor* re-prioritizes the jobs in the *Job Pool* before it schedules in the order of job priorities. When a job finishes, the *Executor* notifies the *DAG Planner* which the job belongs to of the completion status. The collaboration is implemented by the continuous and dynamic event triggered communication among core components, as defined in Fig. 1.

1) *Job submission*. When a new DAG arrives, it is associated with an instance of *DAG Planner*. After ranking all individual jobs within the DAG initially, the *Planner* submits whichever job is ready to the *Job Pool*. In the very first time, only entry job(s) will be submitted. Afterwards, upon notification by the *Executor* of completion of a job, the *Planner* will determine if any successor job(s) become ready and submit them. The job rank information is submitted along with the job.
2) *Job scheduling*. Whenever there are resources available and a job is waiting in the *Job Pool*, the *Executor* will repeatedly do:
    a) Re-prioritize all jobs currently present in *Job Pool* based on individual job ranks.
    b) Remove the job with the highest global priority from *Job Pool*;
    c) Allocate the job to the resource which allows earliest finish time.
3) *Job completion notification*. When a job finishes successfully, the *Executor* will notify the corresponding *DAG Planner* of job completion status.

We name this design as *planner-guided dynamic scheduling*. As illustrated in Fig. 1, each DAG is associated with an instance of *DAG Planner* which ranks individual jobs in the DAG and forwards the ready jobs to the *Job Pool*. If we

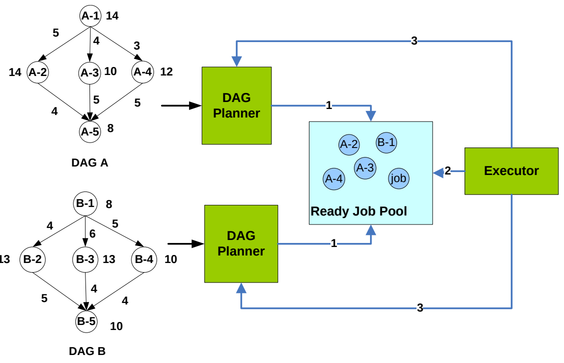


Fig. 1. An overview of planner-guided dynamic scheduling.

assume that DAG B arrives in the system right after the job A-1 finishes, the job A-2, A-3, A-4 and B-1 become ready and are submitted into *Job Pool* which may already have jobs from different users. In turn, the *Executor* will re-prioritize all jobs in *Job Pool* before picking the job with the highest global priority. Priority permutation may occur when *Job Pool* makeup changes, for example, a new job from a different user enters into the pool. The next section will detail how to globally prioritize the jobs in the pool.

C. Prioritization algorithms

Traditional DAG scheduling algorithms are developed for single DAG domain, directly applying them on multiple DAG scheduling is possible but with great practical limitation. It is merely equivalent to one of the composition processes discussed in [8]. It creates a composite DAG by making the nodes which do not have any predecessors of all DAGs the immediate successors of a new common entry node, and all the exit nodes of the DAGs immediate predecessors of a new common exit node. A node does not have any predecessor because either itself is an entry node or its predecessors are executing or have finished when composition process occurs. These two extra common nodes have no computation and no communication between them and other nodes. The major difference from [8] is that we consider that DAGs may arrive dynamically in different time. Reusing the examples in Fig. 1, the composition process will create a composite DAG as illustrated in Fig. 2. As job A-1 has finished, the common entry node makes itself an immediate predecessor of A-2, A-3 and A-4 from DAG A, B-1 from DAG B and another independent job.

One intuitive approach is to simply apply HEFT on the composite DAG by prioritizing jobs in non_increasing order of rank value. For discussion convenience, we refer to this algorithm as *RANK_HF* in the rest of the paper, which means *the highest rank first*. One can easily recognize that this approach is in favor of:

- The jobs from later arriving DAGs. If the DAGs are of similar complexity, the highest possible rank of a partially executed DAG is very likely smaller than the entry nodes of a newly arriving DAG.

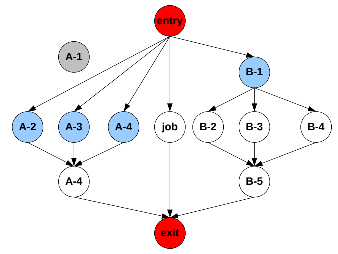


Fig. 2. An example of DAG composition.

```
T: a set of jobs in ready job pool
R: a set of free resources
rank: ranking values for all jobs

procedure schedule(T, R) {
  while $T \neq \emptyset$ and $R \neq \emptyset$ do
    boolean multiple = checkMultiple(T);
    if (multiple)
        sort T as an array L so that:
            for any $i<j$, $L[i] \in T$ and $L[j] \in T$, $rank(L[i]) \leq rank(L[j])$
    else
        sort T as an array L so that:
            for any $i<j$, $L[i] \in T$ and $L[j] \in T$, $rank(L[i]) \geq rank(L[j])$
    endif
    select $L[0] \in T$, where $L[0]$ is the job with the highest priority
    select $r \in R$, where job $L[0]$ has the earliest finish time
            if assigned to resource r
    schedule job $L[0]$ on resource r
    $T = T - \{L[0]\}$
    $R = R - \{r\}$
  endwhile
}

boolean checkMultiple(T) {
  //This function checks if the jobs in T belong to multiple DAGs
  // return true if jobs belong to multiple DAGs, otherwise return false
}
```

Fig. 3. The dynamic scheduling algorithm *RANK_HYBD*.

- The jobs that have bigger computation cost. It is obvious that bigger computation cost can help a job to earn higher rank. In an extreme case where all DAGs are a single job type, the priority is actually equivalent to *the longest job first* policy.

However our later experiment shows that such intuitive extension turns out to be the worst performer compared to others in the evaluation. The reason is as follows. As a DAG starts to execute, its rank value of subsequent individual jobs decreases gradually to the lowest point when it reaches to the exit node. If a new DAG or an independent job with big computation cost is submitted in the middle of its execution, the DAG close to completion will not get any resource allocated due to the likely lower global priority until other DAGs are near completion as well. Such policy results in unnecessary longer turnaround and makespan if the resources are not rich enough, which is validated by the simulation results presented later in this paper.

Based on the observation above and *RANK_HF*'s well known efficiency in scheduling single DAG, we propose a hybrid prioritization algorithm, *RANK_HYBD*, which calculates the global priority based on the rank value of each job in the way as described in Fig. 3. If there is only one DAG

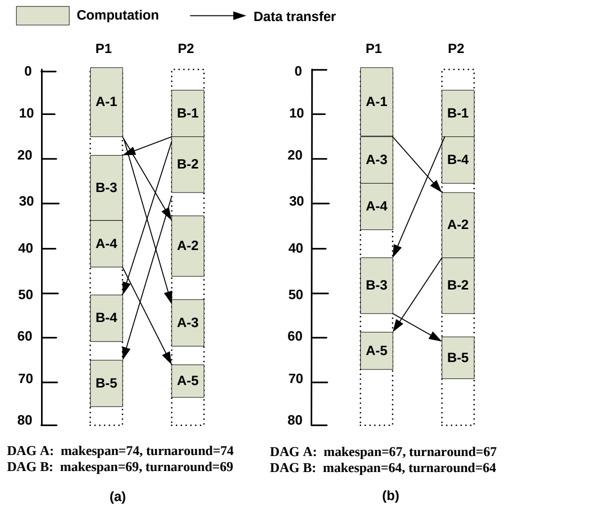


Fig. 4. Scheduling results: (a) scheduling result for algorithm RANK_HF; (b) scheduling result for algorithm RANK_HYBD.

present in the system, *RANK_HYBD* is identical to *RANK_HF*. Otherwise, it prioritizes the jobs in the opposite order. The *Executor* first checks if the jobs in the pool belong to different DAGs. If the jobs come from multiple DAGs, the *Executor* sorts the jobs in a queue (array) which holds the jobs in a non-decreasing order of job ranking value, i.e., the first job has the smallest rank value. If all the jobs belong to the same DAG, the jobs are sorted in opposite order, same as the HEFT algorithm. Then the *Executor* picks the first job in the queue (array) and assigns it to the resource which offers earliest finish time. In an extreme case where all DAGs are actually single jobs, the algorithm is equivalent to the *shortest job first* policy. The later simulation results show that *RANK_HYBD* improves the average makespan and turnaround time very impressively while maintaining similar resource utilization and throughput.

We use the two example DAGs in Fig. 1 to illustrate how algorithm *RANK_HF* and *RANK_HYBD* work. First, a local priority will be assigned to each job by calculating the upward rank values. The job ranking result for each DAG is:

- DAG A: A-1(45); A-2(26); A-3(23); A-4(25) and A-5(8).
- DAG B: B-1(41); B-2(28); B-3(27); B-4(24) and B-5(10).

In this example, we assume that there are two processors, P1 and P2, and DAG B is submitted 6 time units later than DAG A. Fig. 4 shows the scheduling results.

One may notice that, with the algorithm *RANK_HF*, job A-3 and A-4 are scheduled later as they have lower priorities compared with the jobs from DAG B. As DAG B comes in the midst of execution of DAG A, the jobs on the top level certainly have higher rank values. It supports our observation that *RANK_HF* favors DAGs of later arrival and jobs of more complexity. Basically, the algorithm *RANK_HF* penalizes whichever DAG gets close to completion and results in sub-optimal performance from user's perspective.

Conversely, the *RANK_HYBD* assigns higher priority to the jobs of smaller rank value, which implies that either the job is closer to DAG exit point or the job is less complex, as shown in Fig. 4(b). When DAG A has started, the remaining jobs, A-2, A-3 and A-4, are the ones on the lower level and therefore have comparatively smaller rank values, compared with entry jobs from DAG B. The *RANK_HYBD* allows the DAG which gets closer to completion higher priority to obtain required resources, at the expense (delaying) of DAGs arriving later or jobs of more complexity though. However, it helps to reduce the majority's turnaround and better satisfy users. Finally, it is very fair that when a user submits a new DAG into an already well loaded cluster environment or his DAG request is very complex he would reasonably expect a longer turnaround time.

IV. Experiment Design and Evaluation Results

In this section, we present the experiment design for evaluating the effectiveness of *RANK_HYBD*. We comparatively evaluate *RANK_HYBD*, *RANDOM*, *FIFO* and *RANK_HF* with published workflow application test bench and analyze the results under an arrange of system parameters.

A. Algorithms to evaluate

In order to evaluate the effectiveness of *RANK_HYBD*, we compare it with two practically popular algorithms: *RANDOM* and *FIFO*, along with *RANK_HF*. As a matter of fact, the initial study of *RANK_HF* leads us to design and define *RANK_HYBD* in this paper.

The algorithm details of *RANK_HF* and *RANK_HYBD* are described in Section III. As the name suggests, the *RANDOM* algorithm randomly picks up a job from the *Job Pool* without any priority consideration. With *FIFO*, the *Executor* maintains a queue in the order of job entry time and always chooses the job at the the first place of the queue to schedule. Once a job is selected, the four algorithms adopt the same resource selection process by assigning the job to the free resource which offers earliest finish time.

B. Workload simulation

The published test bench [11] for workflow applications is used to evaluate the algorithms. It consists of randomly generated DAGs and is structured according to several DAG graph properties [11]:

- *DAG Size*: the number of nodes in a DAG. As our goal is to evaluate the algorithm performance with complex workload, we use the DAG group with the most jobs only, where each DAG consists of 175 to 249 jobs.
- *Meshing degree*: the extent to which the nodes are connected with each other. It is subdivided into four subcategories: high, medium, low and random.
- *Edge-length*: the average number of nodes located between any two connected nodes. It consists of four subcategories: high, medium, low and random.
- *Node- and Edge-weight*. These two parameters describe the time required for a jobs computation and communication cost. It is related to *CCR*, the communication to computation ratio, but is purposely broken down into Node-high/Edge-high, Node-high/Edge-low, Node-low/Edge-low, Node-low/Edge-high, Node-random/Edge-random subcategories, rather than different CCR values. Obviously, Node-high/Edge-low corresponds to low CCR and Node-low/Edge-high means high CCR.

There are 25 randomly generated DAGs for each combination of subcategories, and they make up totally 2,000 unique test DAGs in our experiment.

The test bench also assumes that each of the available computing nodes, named as target processing elements (TPE) in paper[11], executes just one job at a time and that we have accurate estimates for the computation and communication times of the corresponding DAG scheduling problems[11]. TPE can represent a CPU resource in most contexts of this paper.

Besides the graph properties defined by [11] as above, we add another set of properties to model the dynamic workload:

- *Number of concurrent DAGs*. This is the total number of DAGs concurrently execute in a cluster. We simulate 5, 10, 15, 20 and 25 number of concurrent DAGs respectively in the experiment.
- *Arrival interval*. We are interested in the arrival interval at which the DAGs are submitted into the environment. This is used to mimic the workload dynamics. In the simulation, we assume the the arrival interval follows a Poisson distribution with mean value of 0, 100, 200, 500, 1000, 2000, 3000 and 6000 time units respectively.

With all possible combination of DAG graph properties with dynamic workload characteristics, the experiment involves totally 16,000 test cases based on 2,000 unique DAGs. Finally, the simulation is developed on top of SimJava [22], an event based simulation framework. It is worth noting that we are targeting a cluster environment in this study, but the proposed scheduling algorithm can be used in a grid of one site with large number of computing nodes, or multiple sites that are connected with a high-speed network, such as TeraGrid [23] and Open Science Grid [24].

C. Performance metrics

Since the objective of our algorithm is to improve the workflow application performance, we use the following three metrics to comparatively evaluate all four algorithms:

- *Makespan*: the total execution time for a workflow application from start to finish. It is used to measure the performance of a scheduling algorithm from the perspective of workflow applications.
- *Turnaround time*: the total time between submission and completion of a workflow application, including the real execution time and the waiting time. It measures the performance of a scheduling algorithm from users' perspective.
- *Resource utilization percentage*: the ratio of the time for each resource spending on computation to the total time span to finish all DAGs. This metric is used to measure the algorithm efficiency with respect to resource usage from a system perspective.

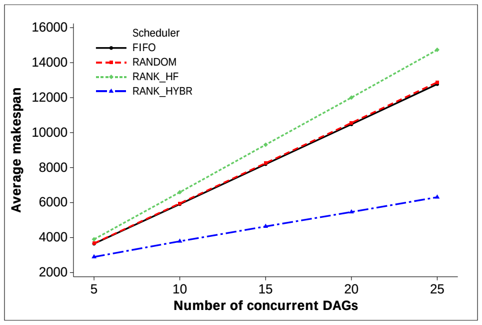


Fig. 5. Average makespan vs. the total number of concurrent DAGs.

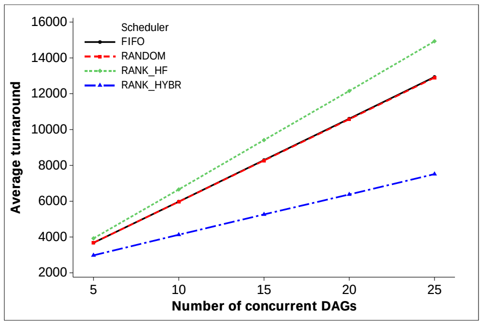


Fig. 6. Average turnaround vs. the total number of concurrent DAGs.

D. Simulation results and analysis

The simulation results of these four algorithms are compared and analyzed with respect to the evaluation metrics described in previous section against various parameters, including DAG graph characteristics and workload dynamic characteristics of arrival interval and concurrency.

Fig. 5 and Fig. 6 show how the algorithms perform with different number of concurrent DAGs. As a result, *RANDOM* and *FIFO* have almost identical performance with respect to average makespan and turnaround, and they both perform better than the *RANK_HF* algorithm. *RANK_HYBD* always outperforms others and improves even more significantly when the computing environment has more DAGs execute concurrently, with respect to average makespan. The average makespan improvement rate of *RANK_HYBD* over *FIFO* increases from 20.6% to 50% when total number of concurrent DAGs increases from 5 to 25.

The same observation holds too when the performance is measured by the average turnaround time, that *RANK_HYBD* outperforms others better when the system serves more DAGs concurrently, as shown in Fig. 6. The improvement rate of *RANK_HYBD* over *FIFO* increases from 19% to 41.9% when the total number of concurrent DAGs increases from 5 to 25. With the page limitation and the fact that turnaround and makespan almost share the identical pattern in the evacuation, in the rest of the paper we will discuss the algorithm evaluation result of both but do not include all turnaround metric related figures.

Fig. 7 helps us to understand how the algorithms respond

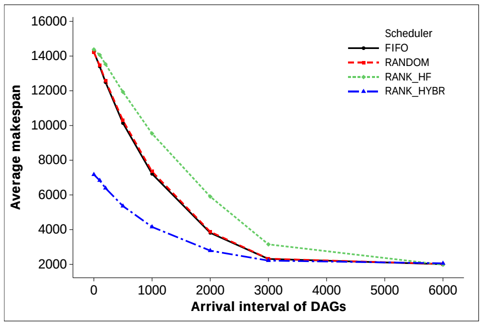


Fig. 7. Average makespan vs. the arrival interval of DAGs.

to the workload intensity measured by interval between DAG submission. It can be easily seen that the more intensively DAGs are submitted, i.e., the smaller arrival interval, the better *RANK_HYBD* outperforms other three algorithms in terms of both makespan, as shown in Fig. 7. When all the DAGs are submitted at the same time, *RANK_HYBD* outperforms *FIFO* by 40% for both average makespan and average turnaround. Interestingly, once again, *RANDOM* and *FIFO* algorithms have very similar performance. When DAGs arrive at an interval of about 6000 time units, it is almost equivalent to the case that one DAG comes in after another one finishes. In this situation, all of these four algorithms have similar performance. However, *RANK_HF* is the best one and outperforms *RANK_HYBD* by 4% slightly with respect to both average makespan and average turnaround. But in reality, most high performance computing centers are overloaded.

We also investigate how these algorithms perform in terms of resource sufficiency, i.e., the number of TPEs, as shown in Fig. 8. We once again find out that *RANDOM* and *FIFO* algorithms have similar performance in all scenarios. Fig. 8 also show that all algorithms do not perform much differently when the cluster environment has sufficient resources, more than 16 TPEs in this experiment. When there are only limited resources available, *RANK_HYBD* is the algorithm of best performance. However, its advantage diminishes in a fast pace when there are more resources available. As shown in Fig. 8, its makespan improvement rate over *FIFO* drops quickly from 52.6% in the case of two TPEs, to 31.5% in the case of eight TPEs. Same pattern is observed with turnaround time, and the improvement rate drops from 43.6% to 27.7% accordingly.

Fig. 9 shows our further evaluation of *RANK_HYBD* with respect to several DAG graph properties. One can easily observe that *RANK_HYBD* outperforms the other three algorithms in all categories significantly. This evaluation also leads to the discovery that *RANK_HYBD* is less sensitive to DAG graph properties and therefore a relatively fair algorithm in terms of makespan. Fig. 9(a) shows that *RANK_HYBD* results in similar makespan for DAGs of different edge lengths. And its performance does not vary much for the DAGs with different meshing degrees either, demonstrated by Fig. 9(b). For the different communication to computation ratios, as shown in Fig. 9(c), it has similar performance for DAGs of Node-

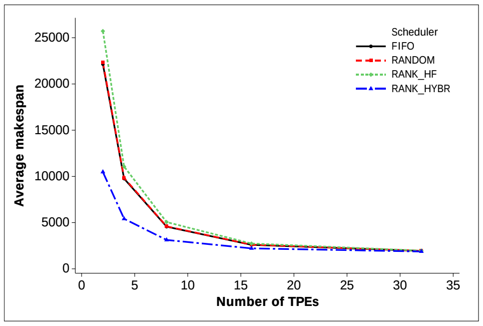


Fig. 8. Average makespan vs. the number of TPEs.

TABLE I
SENSITIVITY TO DAG GRAPH PROPERTIES

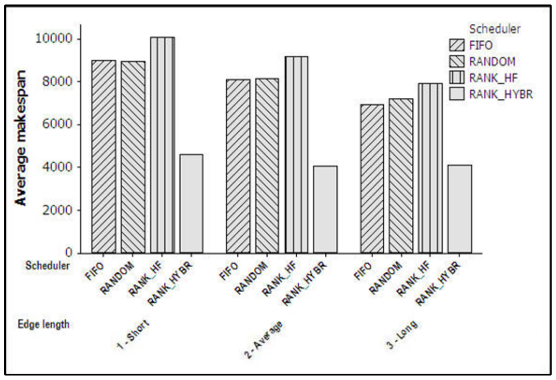
DAG property	Statistic attribute	Makespan		
		RANK_HYBD	FIFO	RANDOM
Mesh degree	Std_Dev	106.6	544.3	503.9
	Mean	4612.3	8228.0	8302.6
Edge length	Std_Dev	296.0	1032.0	890.6
	Mean	4240.3	8017.8	8121.0
CCR	Std_Dev	1704.1	3181.8	3215.5
	Mean	4489.4	7932.8	8007.4

high/Edge-low Node-low/Edge-high, where the former implies low CCR while the latter indicates high CCR. In terms of average turnaround time, all algorithms respond to the each DAG property in a similar way.

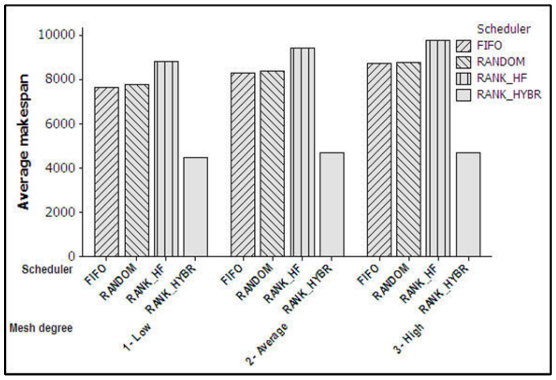
In addition, we compare *RANK_HYBD*, *FIFO* and *RADOM* with respect to their sensitivity to DAG graph properties measuring in the form of the standard deviation (`Std_Dev`), as shown in Table I. It shows that *RANK_HYBD* is much less sensitive to different DAG properties than *FIFO* and *RANDOM* with respect to average makespan. Its sensitivity to turnaround is also less than *FIFO* and *RANDOM*, but not significantly. We attribute this to the fact that the turnaround time is more related to the system workload, rather than the scheduling algorithm when the system is considerably busy.

Finally, we study the algorithm performance from the systems' perspective. Fig. 10 illustrates the empirical cumulative distribution function(CDF) of resource utilization percentage when the simulation is based on 32 TPEs. The figure shows that all algorithms result in very similar resource utilization percentage. Combined all test results, we conclude that *RANK_HYBD* is the best algorithm, it outperforms *FIFO* and *RANDOM* algorithms by 43.6% and 36.7% with respect to average makespan and turnaround time respectively.

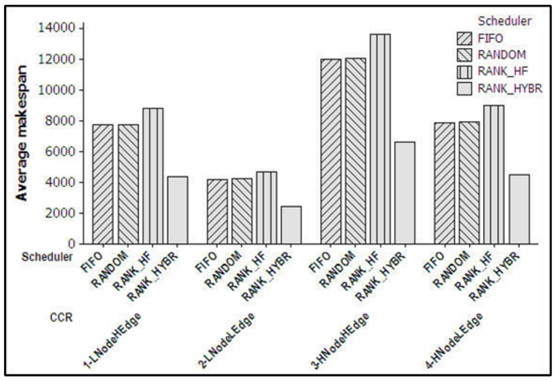
However, we admit that the conclusion with respect to resource utilization efficiency is not solid as the simulation is performed in a relatively small scale. Moreover, the lack of a proper model of the dynamic workload also makes it difficult to evaluate system performance. For the same reason, we do not further evaluate the throughput metric, as the total number of workflow applications alone does not suffice to properly quantify the real complexity of the work requests. We envision that as more and more workflow applications have been developed and executed on cluster environments, the community will have a better idea of the properties of workflow applications. At that time, it will make more sense to evaluate the scheduling algorithms from the perspective of systems. We also evaluate the algorithm in terms of fairness, but do not include it due to page limit.



(a) Average makespan vs. edge length



(b) Average makespan vs. mesh degree



(c) Average makespan vs. CCR

Fig. 9. Effects of DAG properties on the average makespan.

V. SUMMARY AND FUTURE WORK

This paper attacked the problem of scheduling multiple workflow applications in a cluster environment. When considering that multiple workflow applications arrive dynamically and execute concurrently, we found that the most of algorithms

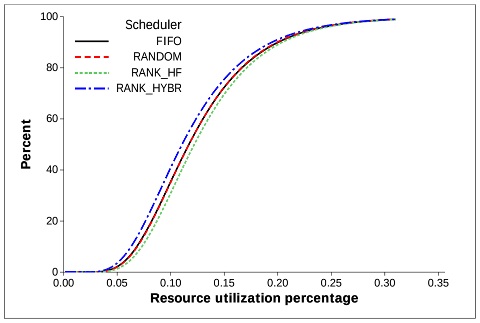


Fig. 10. CDF for resource utilization percentage when TPE=32.

designed for a single workflow application are impractical and very little work has contributed to this problem domain. A new practical solution, planner-guided dynamic scheduling algorithm, is proposed to improve dynamic scheduling performance by guiding it with information about workflow structure and job execution time estimation. A comprehensive simulation has been conducted based on a published workflow application test bench. The evaluation results show that the proposed algorithm outperforms significantly over the two other algorithms by an average of 43.6% and 36.7% with respect to makespan and turnaround respectively.

Discussion We use one popular ranking mechanism [5] in this paper to locally prioritize individual jobs of a DAG. On one hand, it illustrates how the local priority information can help the *Executor* to globally re-prioritize jobs from different users. On the other hand, it demonstrates a two layered scheduling architecture. While the *Planner* determines the local priority for individual jobs in a DAG by utilizing any applicable static algorithm, the *Executor* re-prioritizes all jobs ready to execute in a real time fashion. Even though this research is inspired for scheduling multiple workflow applications, it is generically applicable to the cluster and grid platforms of mixed workloads. Finally, this solution is implementable in practice. For example, DAGMan can accept job submissions with a user's predefined priority. Rather than allowing users to define these priorities without any guidance, we can let the *Planner* determine the order in a unified and systematic way. What we need to do additionally is to add a global prioritization functionality to the Condor job queue manager so it can dynamically re-prioritize the jobs in the queue, instead of the current *FIFO* implementation.

In the next step, we will further refine the dynamic workload model to make it more realistic. We also plan to study the algorithm efficiency from the system management perspective. In addition, we intend to implement the proposed algorithm *RANK_HYBD* in a real workflow scheduler, such as DAGMan/Condor-G [12].

References

[1] M. Wieczorek, R. Prodan, and T. Fahringer, "Scheduling of scientific workflows in the askalon grid environment," *SIGMOD Record*, vol. 34, no. 3, pp. 56–62, 2005.

[2] M. Lopez, E. Heymann, and M. Senar, "Analysis of dynamic heuristics for workflow scheduling on grid systems," in *Proceedings of the 5th International Symposium on Parallel and Distributed Computing (ISPDC'06)*. IEEE, 2006, pp. 199–207.

[3] J. Blythe *et al.*, "Task scheduling strategies for workflow-based applications in grids," in *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05)*, Cardiff, UK, 2005.

[4] Y. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Computing Surveys*, vol. 31, no. 4, pp. 406–471, 1999.

[5] H. Topcuouglu, S. Hariri, and M. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distribution Systems*, vol. 13, no. 3, pp. 260–274, 2002.

[6] A. Radulescu and A.Gemund, "Fast and effective task scheduling in heterogeneous systems," in *Proceedings of the 9th Heterogeneous Computing Workshop (HCW'00)*. Washington, DC, USA: IEEE, 2000, p. 229.

[7] R. Sakellariou and H. Zhao, "A low-cost rescheduling policy for efficient mapping of workflows on grid systems." *Scientific Programming*, vol. 12, no. 4, pp. 253–262, 2004.

[8] H. Zhao and R. Sakellariou, "Scheduling multiple dags onto heterogeneous systems," in *Proceedings of the 15th Heterogeneous Computing Workshop (HCW)*, Rhodes Island, Greece, April 2006.

[9] U. Hönig and W. Schiffmann, "A meta-algorithm for scheduling multiple dags in homogeneous system environments," in *Proceedings of the 18th International Conference on Parallel and Distributed Computing and Systems (PDCS'06)*. IEEE, 2006.

[10] A. Mandal *et al.*, "Scheduling strategies for mapping application workflows onto the grid," in *Proceeding of the 14th International Symposium on High Performance Distributed Computing (HPDC'05)*. IEEE, 2005, pp. 125–134.

[11] U. Hönig and W. Schiffmann, "A comprehensive test bench for the evaluation of scheduling heuristics," in *Proceedings of the 16th International Conference on Parallel and Distributed Computing and Systems (PDCS'04)*. IEEE, 2004.

[12] "Dagman." [Online]. Available: http://www.cs.wisc.edu/condor/dagman/

[13] J. Cao, S. Jarvis, S. Saini, and G. Nudd, "Gridflow: Workflow management for grid computing," in *Proceeding of 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2003, pp. 198–205.

[14] E. Deelman *et al.*, "Mapping abstract complex workflows onto grid environments," *Journal of Grid Computing*, vol. 1, pp. 25–36, 2003.

[15] T. Oinn *et al.*, "Taverna: A tool for the composition and enactment of bioinformatics workflows," *Bioinfomatics*, vol. 20, no. 17, pp. 3045–3054, 2004.

[16] F. Berman *et al.*, "New grid scheduling and rescheduling methods in the grads project," *International Journal of Parallel Programming*, vol. 33, no. 2, pp. 209–229, 2005.

[17] G. Malewicz, A. Rosenberg, and M. Yurkewych, "Toward a theory for scheduling dags in internet-based computing," *IEEE Transactions on Computers*, vol. 55, no. 6, pp. 757–768, 2006.

[18] G. Malewicz, I. Foster, A. Rosenberg, and M. Wilde, "A tool for prioritizing dagman jobs and its evaluation," in *Proceedings of the 15th International Symposium on High Performance Distributed Computing (HPDC'06)*. IEEE, 2006, pp. 156–167.

[19] M. Iverson and F. Özgüner, "Hierarchical, competitive scheduling of multiple dags in a dynamic heterogeneous environment." *Distributed Systems Engineering*, vol. 6, no. 3, pp. 112–120, 1999.

[20] H. Dail *et al.*, "Scheduling in the grid application development software project," in *Grid Resource Management: State of the Art and Future Trends*, J. Nabrzyski, J. Schopf, and J. Weglarz, Eds. Norwell, MA, USA: Kluwer Academic Publishers, 2004.

[21] Z. Yu and W. Shi, "An adaptive rescheduling strategy for grid workflow applications,," in *Proceeding of 21st International Parallel and Distributed Processing Symposium (IPDPS'07)*, Long Beach, Florida, USA, March 2007.

[22] "SimJava." [Online]. Available: http://www.dcs.ed.ac.uk/home/hase/simjava/

[23] "NSF TaraGrid." [Online]. Available: http://www.teragrid.org/

[24] "Open Science Grid." [Online]. Available: http://www.opensciencegrid.org/