An Adaptive Rescheduling Strategy for Grid Workflow Applications

Zhifeng Yu and Weisong Shi

Wayne State University
{zhifeng.yu, weisong}@wayne.edu

Abstract

Scheduling is the key to the performance of grid workflow applications. Various strategies are proposed, including static scheduling strategies which map jobs to resources before execution time, or dynamic alternatives which schedule individual job only when it is ready to execute. While sizable work supports the claim that the static scheduling performs better for workflow applications than the dynamic one, it is questioned how a static schedule works effectively in a grid environment which changes constantly. This paper proposes a novel adaptive rescheduling concept, which allows the workflow planner works collaboratively with the run time executor and reschedule in a proactive way had the grid environment changes significantly. An HEFT-based adaptive rescheduling algorithm is presented, evaluated and compared with traditional static and dynamic strategies respectively. The experiment results show that the proposed strategy not only outperforms the dynamic one but also improves over the traditional static one. Furthermore we observed that it performs more efficiently with data intensive application of higher degree of parallelism.

1 Introduction

Typically a workflow application is a set of jobs which are coordinated by control and data dependencies to accomplish a complex task. It is popular in scientific computation and becomes even more widely accepted thanks to the growing popularity of grid computation. In general, a scientific workflow application can be represented as a direct acyclic graph (DAG), where the node is the individual job and edge represents the inter-job dependence. Both nodes and edges are weighed for computation cost and communication cost respectively. The *makespan*, which is the total time needed to finish the entire workflow, is used to measure the performance of workflow applications.

One of the key functions of a workflow management system on grid is to schedule and manage the jobs on shared resources to achieve high performance. When it comes to system implementation, the workflow planner and executor are two core components in terms of how the resource mapping decision is made and how job is scheduled. Existing systems are going into two different extremes [6]. Some systems use static approaches, i.e., fully plan ahead, by which the planner makes the global decisions in favor of entire workflow performance relying on knowledge of the entire DAG and execution environment. Others depend on the workflow executor to make decision for each individual job only when it becomes ready to execute. This type of decision is also referred to as local just-in-time decision. Among the performance driven DAG-based grid workflow systems, DAGMan [8] and Taverna [13] support dynamic scheduling, GridFlow [3] supports static scheduling, and Pegasus [16] supports both. It is believed that static strategy can potentially perform near optimal, and this is also proven true with some real world workflow applications [20]. The simulation work [2] further suggests that static approaches still perform better than dynamic ones for data intensive workflow applications even with inaccurate information about future jobs.

However, in a grid environment static strategies may perform poorly because of the grid dynamics: resource can join and leave at any time; individual resource capability varies over time because of internal or external factors; and it is not easy to accurately estimate the communication and computation cost of each job, which is the foundation of any static scheduling. Recent work [7, 18, 11] shows that scheduling through resource reservation and performance modeling can help to ensure the resource availability during executing and theoretically make the grid more predictable, but their approaches do not solve all the problems.

We argue that the promising benefits of static strategies can be practically realized with collaboration between workflow planner and executor, which is currently missed in most system designs. This paper proposes an *HEFT* [19] based adaptive rescheduling algorithm to support such de-

1-4244-0910-1/07/$20.00 ©2007 IEEE.

sired collaboration. With this approach, the executor will notify the planner of any run time event which interests the planner, for example, resource unavailability or discovery of new resource. In turn, the planner responds to the event by means of evaluating the event and rescheduling the remaining jobs in the workflow if necessary. Planning is now an iterative (event-driven) activity instead of one time task. The experiment results, including simulation on both parametric randomly generated DAGs and two real application DAGs, show a considerable performance improvement by adaptive rescheduling.

The contributions of this paper are: (1) propose an adaptive rescheduling approach; (2) evaluate the performance of dynamic strategy when resource change is considered, and observe that traditional static strategy still performs better; and (3) study how the adaptive rescheduling improves over traditional static strategy when resource pool changes over time. In the simulation with two real world workflow applications, BLAST [17] and WIEN2K [21], we found that our approach outperforms the traditional static approach by 20.4% and 6.1% respectively.

The rest of the paper is organized as follows. Related work is discussed in Section 2. Then we describe an adaptive rescheduling approach in Section 3. Section 4 elaborates the experiment design, and evaluates the performance of adaptive rescheduling. Finally, we summarize and lay out the future work in Section 5.

2 Related Work

Job scheduling problem is an NP-Complete problem [9], it is extensively studied and various heuristics are proposed in the literature. *HEFT* (Heterogenous Earliest Finish Time) [19] is one of the most popular heuristics, it is implemented in the grid project ASKALON [20] and proven superior to other alternatives. Some other heuristics are studied in a comprehensive evaluation [10], and surprisingly they show a very similar behavior regarding the quality of the obtained results, exhibiting the same strengths and weaknesses, differing only by few percent. Based on these observations, *HEFT* heuristic is selected in this paper to implement the adaptive rescheduling algorithm.

The challenge of scheduling grid workflow application with static strategy is discussed in research [6], but few research efforts address them. Rescheduling is implemented in the GrADS [1], where it is normally activated by contract violation. However, the efforts are all conducted for iterative applications, allowing system to perform rescheduling decisions at each iteration [5]. The *plan switching* approach [22] is to construct a family of activity graphs beforehand and investigates the means of switching from one member of the family to another when the execution of one activity graph fails, but all the plans are made without knowledge about the future environment change.

Another rescheduling policy is proposed in [14], which considers rescheduling at a few, carefully selected points during the execution. The research tackles one of the shortcomings that static scheduling always assumes accurate prediction of job performance. After the initial schedule is made, it selectively reschedules some jobs if the run time performance variance exceeds predefined threshold. However, this approach deals with only the inaccurate estimation and does not consider the change of resource pool.

As a complementary research to the above, we focus on how the workflow planner adapts to the resource pool change. For example, when the new resources are discovered, the planner will evaluate whether these extra resources can be utilized to achieve better performance and reschedule the remaining jobs if necessary.

3 Adaptive Rescheduling

Even though theoretically static scheduling performs near to optimal, its effectiveness in a dynamic grid environment is questioned. We discuss and analyze these issues in the beginning of this section, and propose a static strategy based novel *adaptive rescheduling* algorithm by which the workflow planner can adapt to the grid dynamics to achieve its strength practically.

3.1 Issues with Static Scheduling

Planning is a one time activity in the traditional static scheduling paradigm. It does not consider the future change of grid environment after the resource mapping is made. On the other hand, rescheduling in execution phase is proposed but mainly used to support fault tolerance. Overall, the issues with traditional static scheduling are: (1) *Accuracy of estimation*. Estimating communication and computation costs of a DAG is the key success factor but practically difficult. The deviation in run time is detrimental to scheduling based scheduling. (2) *Adaptation to dynamic environment*. Most static scheduling approaches assume that resource set is given and fixed over time. The assumption is not always valid even with the reservation capability in place. Moreover, the static scheduling approach can not utilize new resources after the plan is made; and (3) *Separation of workflow planner from executor*. Fundamentally the above two issues are related to the lack of collaboration between the workflow planner and executor. With collaboration, a planner will be aware of the grid environment change, including the job performance variance and resource availability, and is able to adaptively reschedule based on the increasingly accurate estimations. This approach can both continuously improve performance by considering the new resources and minimize the impact caused by unexpected resource downgrade or unavailability.

3.2 System Architecture

We propose a system design which adapts the *Planner* to dynamic grid environment via collaboration with the *Executor*, as shown in Fig. 1. The system consists of two main components *Planner* and *Executor*. The *GRID Services* on top of which the *Executor* is built, is a collection of essential services for any grid system and not the focus of this paper.

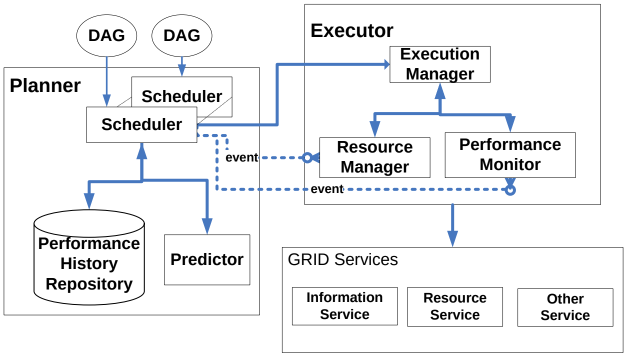


Figure 1. The diagram of the system design.

The capabilities of each system components are defined as below:

Planner. The *Planner* has a collective set of subcomponents, including *Scheduler*, *Performance History Repository* and *Predictor*. For each workflow application represented as a DAG, the *Planner* instantiates a *Scheduler* instance. Based on the performance history and resource availability, the *Scheduler* inquires the *Predictor* to estimate the communication and computation cost with the given resource set. It then decides on resource mapping, with the goal of achieving optimal performance for entire workflow, and submits the schedule to the *Executor*. During the execution, the *Scheduler* instance listens to the pre-defined events of interest, for example, addition of new resources and significant variance of job performance, evaluates the event and reschedules the application if necessary. In the mean time, the *Scheduler* updates the *Performance History Repository* with the latest job performance information to improve the estimation accuracy subsequently. As each instance of *Scheduler* manages an individual DAG, it will focus on the events specifically associated with the DAG it manages to continuously improve the *makespan*. By collectively working with other instances of *Scheduler*, this multiple-instance scheme can further improve the overall system performance.

Executor. The *Executor* is an enactment environment for workflow applications, can be further decomposed into *Execution Manager*, *Resource Manager* and *Performance Monitor* according to their respective roles in the run time environment. The *Executor* supports advance reservation of resources. The *Execution Manager* receives the DAG and executes it as scheduled. It is also responsible for getting job input file ready and executing the job on mapped resource. Upon arrival of a schedule, the *Resource Manager* will reserve the resource as per the schedule. If the arriving schedule is a result of rescheduling, it revokes resource reservation for replaced schedule before making new reservations. As part of the collaboration, the *Resource Manager* and *Performance Monitor* update the *Planner* with information and event subscribed by the *Planner*.

3.3 Adaptive Scheduling

We present the basic idea of adaptive scheduling in this subsection, followed by a detailed algorithm based on HEFT [19] in the next subsection. For a given DAG and a set of currently available resources, the *Planner* makes the initial resource mapping as any other traditional static approaches do. The primary difference is that our approach requires the *Planner* listens for and adapts to the significant events in the execution phase, such as:

- *Resource Pool Change*. If new resource is discovered after the current plan is made, rescheduling may reduce the *makespan* of a DAG by considering the resource addition. When resource fails, fault tolerant mechanism is triggered and it is taken care of by *Execution Manager*. However, if the failure is predictable, rescheduling can minimize the failure impact on overall performance.

- *Resource Performance Variance*. The performance estimation accuracy is largely dependent on history data, and inaccurate estimation leads to a bad schedule. If the run time *Performance Monitor* can notify the *Planner* of any significant performance variance, the *Planner* will evaluate its impact and reschedule if necessary. In the meantime, the *Performance History Repository* is updated to improve the estimation accuracy in the subsequent planning.

The *Planner* reacts to event by evaluating if *makespan* can be reduced by rescheduling. For example, if a new resource becomes available, the *Planner* will evaluate if a new schedule with the extra resource in consideration can produce smaller *makespan*. If so, the *Planner* will replace the current one with new one by submitting it to the *Executor*.

The evaluation can be further extended to support online system management function by answering the “*What...if...*” type query, for example, “*What* will be the expected performance *if* an additional resource A is added (removed)?” The query result, as evaluation output, will help one to tune up the application and system performance in a proactive way, and this will be our future work.

A generic adaptive rescheduling algorithm is described in Fig. 2. For a given DAG, at initial time or when an event occurs during its execution, the *Planner* schedules or evaluates the event by (re)scheduling. The *Planner* retrieves the latest resource information and job performance history data first, and estimates the cost of each job in the DAG. Based on the estimation, the *Planner* applies a specific static heuristic, for example *HEFT*, either makes an initial schedule for the entire DAG or a new schedule for the remaining jobs. If the schedule is an initial one or it is expected to perform better than the current one, the *Planner* submits it to the *Executor* to execute, otherwise the *Planner* does not take any action. Until the DAG is executed successfully, the *scheduler* keeps listening for the event of interest, evaluate it and reschedule the DAG if necessary. S_0 and S_1 denotes the scheduler and rescheduler respectively.

```
T - set of the jobs in the DAG
R - set of all available resources
P - performance estimation matrix
H - Heuristic employed by scheduler
S - Schedule
set initial schedule $S_0$ = null
while (($S_0$==null OR any event) AND DAG  not finished) do
    #R is updated via the communication with Resource Manager
    update Resource Set R;
    update Performance History Repository;
    #Predicator component will update performance estimation matrix P
    call P=estimate(T, R);
    #New schedule is made by applying the heuristics H on execution
        status snapshot of $S_0$ and P
    call $S_1$=schedule($S_0$, P, H);
    if ($S_0$==null OR $S_0$.makespan>$S_1$.makespan)
        $S_0$=$S_1$;
        submit $S_0$;
    endif
endwhile
```

Figure 2. A generic adaptive rescheduling algorithm.

3.4 HEFT-based Adaptive Rescheduling: AHEFT

Next we define our own adaptive scheduling strategy, which is an *HEFT*-based adaptive rescheduling algorithm, referred to as *AHEFT* hereafter. Specifically, we use *HEFT* heuristic to implement the $schedule(S_0, P, H)$ method in the generic algorithm described in Fig. 2. For consistence purposes, we directly use the scheduling system model defined in paper [19] with revision and extension. 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. R is a set of resources which represent computation units. The variable *clock* is used as logical clock to measure the time span of DAG execution, it is initially set as 0 when the DAG starts to execute. When the DAG finishes successfully, the *clock* reads as the *makespan* of the DAG.

we define the symbols used by *AHEFT* in Table 1, and formulate the calculation of these attributes by these three equations, whose rationale are described next.

$$FEA(n_m,n_i,r_j,S_0,clock)=\begin{cases} AFT(n_m), & Case1 \\ clock+c_{m,i}, & Case2 \\ SFT(n_m), & Case3 \\ SFT(n_m)+c_{m,i}, & \text{otherwise.} \end{cases} \tag{1}$$

wherein,

Case 1: If job n_m finished **and** S_0 maps job n_m to r_j;

Case 2: If job n_m finished **and** S_0 does **not** map job n_m to r_j.

Case 3: If job n_m has not finished **and** S_1 maps job n_m to r_j.

$$EST(n_i,r_j,S_0,clock,R)= \\ max\{avail[j],\max_{n_m \in pred(n_i)}(FEA(n_m,n_i,r_j,S_0,clock))\} \tag{2}$$

and

$$EFT(n_i,r_j,S_0,clock,R)=w_{i,j}+EST(n_i,r_j,S_0,clock,R) \tag{3}$$

A job can not start without all required inputs available on the resource on which the job is to execute. Such inputs are in turn the outputs from immediate predecessor jobs. If a job n_i will execute on resource r_j and requires output data from an immediate preceding job n_m, the Equation (1) calculates the earliest time when output data arrives on resource r_j. By the time of (re)scheduling, *clock*, if job n_m already finishes on resource r_j then its output is ready there as input for job n_i, and the FEA equals $AFT(n_m)$. If job n_m finishes but on different resource, then its output has to transfer to resource r_j. As the file transmission can not be earlier than *clock*, the FEA is equal to $clock + c_{m,i}$. If a job n_m has not finished by $clock$, it will be rescheduled by rescheduler S_1, as applied to the rest of cases.

On the other hand, a job can not execute before the earliest available time $avail[j]$ for resource r_j. These constraints are indicated by the inner *max* block in Equation (2). It is easy to get the earliest finish time of job n_i by

Table 1. Definition of attributes in *AHEFT*

Attribute	Definition
$EST(n_i, r_j, S_0, clock, R)$	the earliest start time for not-started job n_i on resource r_j with available resource set R when the schedule S_0 is executed to the time point of *clock*
$EFT(n_i, r_j, S_0, clock, R)$	the earliest finish time for not-started job n_i on resource r_j with available resource set R when the schedule S_0 is executed to the time point of *clock*
$FEA(n_m, n_i, r_j, S_0, clock)$	the earliest time for file output of job n_m being available on resource r_j ready for job n_i after schedule S_0 has been executed to the time point of *clock*
$SFT(n_i)$	scheduled finish time of job n_i when it is mapped to a resource
$AST(n_i)$	actual start time of job n_i
$AFT(n_i)$	actual finish time of job n_i
$avail[j]$	the earliest time when resource r_i is ready for job execution
$w_{i,j}$	the computational cost of job n_i on resource r_j
$c_{i,j}$	the communication cost for data dependence of job n_j on n_i
$pred(n_i)$	the set of immediate predecessor jobs of job n_i

adding the estimated execution time $w_{i,j}$ to its earliest start time. After a job n_i is scheduled on resource r_j, the earliest finish time for job n_i on resource r_j is denoted as $SFT(n_i)$, the scheduled finish time of job n_i. Finally the *makespan* is defined as,

$$makspan = max\{SFT(n_{exit})\} \tag{4}$$

where n_{exit} is the exit job in a DAG. There can be one or multiple exit jobs in one DAG.

It is obvious that *AHEFT* is identical to *HEFT* [19] when $clock = 0$ or it is the initial scheduling, i.e. S_0 is not defined yet. The primary difference comes to the rescheduling when *AHEFT* considers the fact that workflow has been executed partially. In Equation (1), the FEA can be actual available time if by the time of rescheduling, i.e., $clock$, the immediate predecessor job finishes and output file is moved or to be moved to the resource as previously scheduled. However, previous schedule may direct the output file to different resource, then the file needs to be retransmitted to this resource regardless, which falls into the second situation in Equation (1). The third one is the same as *HEFT*, if either this is initial scheduling or the immediate predecessor job has not started yet. With these equations now we can define the procedure $schedule(S_0, P, H)$ of *AHEFT*, see Fig. 3.

Except for how EFT is calculated, the procedure $schedule(S_0, P, H)$ defined in Fig. 3 is very similar to *HEFT*. Based on the cost estimation obtained, i.e., line 5 in Fig. 2, the upward rank of a job n_i is recursively defined, starting from the job n_{exit}, by [19]

$$rank_u(n_i) = \overline{w_i} + \max_{n_j \in succ(n_i)} (\overline{c_{(i,j)}} + rank_u(n_j)) \tag{5}$$

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

$$rank_u(n_{exit}) = \overline{w_{exit}} \tag{6}$$

```
   T - set of the jobs of status not started in DAG
   R - set of all available resources
   P - performance estimation matrix
   H - HEFT heuristic employed by scheduler
   S0 - Initial schedule
   clock - the time point of scheduling
1. procedure schedule(S0, P, H)
2.   compute rank_u for all jobs by traversing graph upward, starting
        from the exist job
3.   sort the jobs in a scheduling list by nonincreasing order of rank_u
4.   while there are unscheduled jobs in the list do
5.       select the first job, n_i from the list of scheduling
6.       for each resource r_k in R do
7.          compute EFT(n_i, r_k , S0, clock, R)
8.          assign job n_i to the resource r_j that minimizes EFT of job n_i
9.    endwhile
```

Figure 3. Procedure $schedule(S_0, P, H)$ of *AHEFT*.

As indicated by line 2 and 3 in Fig. 3, the upward rank is calculated for each job and sorted in nonincreasing order which corresponds to significance order how the individual job affects the final *makespan*. The basic concept of this algorithm is to select the “best” resource which minimizes the earliest finish time of the job currently with highest upward rank and remove the job from unscheduled job list once it is assigned with resource. The resource selection process repeats until the list is empty.

As an illustration, we use a sample DAG and resource set, shown in Fig. 4, to compare schedule performance of traditional *HEFT* and *AHEFT*. Fig. 5 shows the schedule obtained from traditional *HEFT* and *AHEFT* respectively. Resources r_1, r_2 and r_3 are available from the beginning, while resource r_4 emerges at time point of 15. *HEFT* produces the schedule with makespan as 80 without considering the addition of resource r_4 at later time. For *AHEFT*, the initial schedule made at time point of 0 is identical as the one by *HEFT*. However, when resource r_4 is added, *HEFT* reschedules the rest of the workflow, i.e. all jobs but n_1 which is finished by the time of rescheduling. The new schedule reduces the *makespan* to 76.

4 Experiment Design and Results

In this section, we present the experiment design for evaluating the effectiveness of *AHEFT*. We first evaluate it with randomly generated DAGs. Then we specifically compare it with traditional *HEFT* in the context of two real world applications, namely BLAST [17] and WIEN2K [21].

4.1 Experiment Design

The following important assumptions are made for the experiment design: (1) *Accuracy of estimation*. As other studies [2, 14, 19], the estimation of communication and computation cost is assumed accurate and job will start and finish on time; (2) *File transferring*. For static approaches, when a job finishes, the output file of the job is transmitted immediately to the resources where the immediate succeeding jobs are scheduled to execute on. But for dynamic one the output file is not transmitted until the *Executor* decides on which resource to run the depending job. In both cases, the file transmission is time consuming only activity and does not incur computation cost; and (3) *Advance resource reservation*. We assume the advance reservation capability ensures resource availability during the reserved time window. On the other hand, *HEFT* and *AHEFT* react identically to the resource failure while job is executing, as if rescheduling is the fault tolerance mechanism. Therefore, to simplify the experiment design, we can reasonably only consider the situation that new resources come available during the execution of workflow.

4.2 Results of Parametric Randomly Generated DAGs

In order to evaluate the performance and stability of *AHEFT*, i.e., whether it always performs better than *HEFT* and dynamic one in all kinds of cases, we use parametric randomly generated DAGs in the experiment. For the purpose of fair comparison, we directly follow the heterogeneous computation modeling approach defined in [19] to generate representative DAG test cases. The input parameters and the corresponding values are very similar as used in [19] as well. These input parameters are also suggested in the workflow test bench work[10], as listed below:

- The number of jobs in the graph (v).
- The maximum out edges of a node, out_degree, represented as percentage of total nodes in a DAG.
- Communication to computation ratio (CCR). A data-intensive application has a higher CCR, while a computing-intensive one has a lower value.
- The resource heterogenous factor, β. A higher value of β suggests the bigger difference of resource capability. The resources are homogeneous when β is 0. The average computation cost of all jobs in a DAG is $\overline{\omega_{DAG}}$, then the average of each job n_i in the graph, represented as $\overline{\omega_i}$, is selected randomly from a uniform distribution with range $[0, 2 \times \overline{\omega_{DAG}}]$. Then, the computation cost of each job n_i on each resource r_j in the system, i.e., $\omega_{i,j}$, is randomly selected from the following range: $\overline{\omega_i} \times (1 - \frac{\beta}{2}) \leq \omega_{i,j} \leq \overline{\omega_i} \times (1 + \frac{\beta}{2})$.

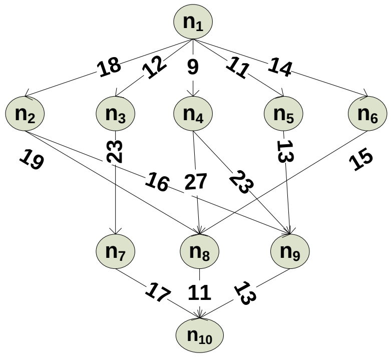
Table 2. Parameter values of random generated DAGs.

Parameter	Value
v	20, 40, 60, 80, 100
CCR	0.1, 0.5, 1.0, 5.0, 10.0
out_degree	0.1, 0.2, 0.3, 0.4, 1.0
β	0.1, 0.25, 0.5, 0.75, 1.0
R	10, 20, 30, 40, 50
Δ	400, 800, 1200, 1600
δ	0.10, 0.15, 0.20, 0.25

To model the dynamic change of resources, we introduce three additional parameters as following: (1) Initial resource pool size, R; (2) Interval of resource change, Δ. The higher value of Δ indicates the lower frequency of resource change; and (3) Percentage of resource change, δ, to measure the resource change percentage each time compared with the initial resource pool. The value set for each parameter of this empowerment is listed in Table 2.

With combination of v, CCR, out_degree and β, we have totally 625 different DAG types. For each type we create 10 instances with randomly assigned computation and communication cost, so there are totally 6250 DAGs used in the experiment. Then we apply 80 different types of resource models, combining the R, Δ and δ, so we finally generate 500,000 test cases. For each DAG, we simulate *HEFT* [19], *AHEFT* and dynamic *Min-Min* [4] heuristic and obtain the respective *makespan*. The simulation for dynamic *Min-Min* is implemented on top of the event-driven simulation framework SimJava [15].

The average *makespan* for *HEFT*, *AHEFT* and *Min-Min* are 4075, 3911 and 12352 respectively. It shows that both *HEFT* and *AHEFT* achieve much better performance than *Min-Min*, and *AHEFT* is slightly better than *HEFT*. We further compare *AHEFT* and *HEFT* to identify which type of



Computation Cost

Job	Resource			
	r_1	r_2	r_3	r_4
n_1	14	16	9	*14*
n_2	13	19	18	*17*
n_3	11	13	19	*14*
n_4	13	8	17	*15*
n_5	12	13	10	*14*
n_6	13	16	9	*16*
n_7	7	15	11	*15*
n_8	5	11	14	*20*
n_9	18	12	20	*13*
n_{10}	21	7	16	*15*

Figure 4. A sample DAG, the weight of each edge represents its communication cost.

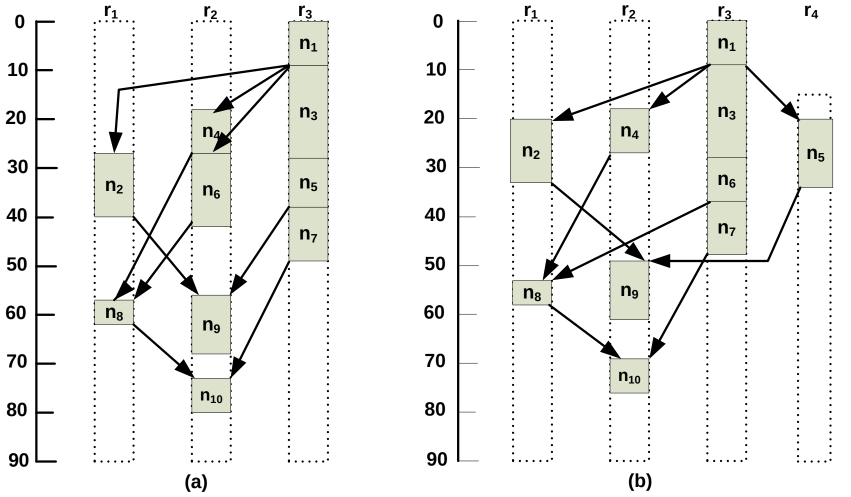


Figure 5. Schedule of the DAG in Fig. 4 using *HEFT* and *AHEFT* algorithms: (a) *HEFT* schedule (*makespan*=80) and (b) *AHEFT* schedule with resource adding at time 15 (*makespan*=76).

Table 3. Improvement rate with various *CCR*s.

CCR	0.1	0.5	1.0	5.0	10.0
Imprv. rate	0.4%	0.5%	0.7%	3.2%	7.7%

Table 4. Improvement rate with various total number of jobs.

Job number	20	40	60	80	100
Imprv. rate	2.9%	3.9%	4.3%	4.2%	4.1%

workflow applications can benefit more from *AHEFT* by studying the effect of different parameters. Given the limited space, we show the results of CCR and the number of jobs in Table 3 and Table 4 respectively. One can easily notice that *AHEFT* favors data-intensive workflow application by Table 3. When CCR increases, i.e., application is more data-intensive, *AHEFT* outperforms *HEFT* better. Another observation is, with the total number of jobs increases, the improvement rate jumps initially and becomes stable later, as Table 4 shows.

It is worth noting that these observations are drawn from the experiments with randomly generated DAGs of limited scale (less or equal to 100 jobs). To better understand the correlation between *AHEFT* and workflow application characteristics, we evaluate with two real world applications in the next subsection.

4.3 Results of BLAST and WIEN2K

We attribute the less significance of the performance improvement in randomly generated DAGs to two observations below: 1) *DAG shape*. Typically a scientific workflow application is designed to accomplish a complex task by means of job parallelism, its DAG is hence uniquely shaped. The DAGs of many real world workflow applications are well balanced and highly parallel, like Montage [12], BLAST [17] and WIEN2K [21], and so forth. Moreover, the DAG shape decides the job parallelism degree to some extent; 2) *Types of jobs in the DAG*. Despite of the fact that one scientific workflow is composed of hundreds individual jobs if not thousands, there are only handful unique operations. For example, Montage has totally 11 unique executable operations. The same operation appears as different individual jobs in the DAG when it is executed in different context with different inputs. This observation holds same true with BLAST and WIEN2K applications. Fig. 6 gives a six-step BLAST workflow example with two-way parallelism. This workflow represents a set of function calls that specify inputs such as genome sequence files, output files from comparative analysis tools, and textual parameters. We conduct the simulation with 200-, 400-, 600-, 800- and 1000-way parallelism respectively. With these two observations we choose BLAST and WIEN2K DAGs to evaluate how well adaptive rescheduling may improve practically and how its effectiveness is related to the DAG characteristics. BLAST and WIEN2K are implemented in grid system GNARE [17] and ASKALON [20] respectively.

WIEN2k [21] is a quantum chemistry application developed at Vienna University of Technology. WIEN2k workflow contains two parallel sections $LAPW1$ and $LAPW2$, with possibly multiple parallel tasks. The DAG we used for experiment is a full-balanced graph, with equal number of parallel jobs in these two sections, as shown in Fig. 7. In the experiment, we set the number of parallel tasks as 200, 400, 600, 800, 1000 respectively. The parallelism factor used in both BLAST and WIEN2K actually decides the total number of jobs in the DAG. We define the value set for each parameter of experiment with both BLAST and WIEN2K in Table 5. Table 6 shows the average *makespan* improvement by *AHEFT* over BLAST and WIEN2K respectively. The results again assert that the effectiveness of adaptive rescheduling is very sensitive to the parallelism degree of DAGs, which in turn relates to the DAG shape, well corresponding to our observation mentioned earlier of this section. For the DAGs with shape like BLAST, *AHEFT* can help to reduce *makespan* by 20.4% on average when new resources are added to the system periodically. But it only improves a little with the WIEN2K DAG. The difference is understandable if one notices that the parallelism degree of WIEK2K is obviously lower than that of BLAST, so that any additional resource is less likely utilized and contributes less to the performance improvement. Despite of

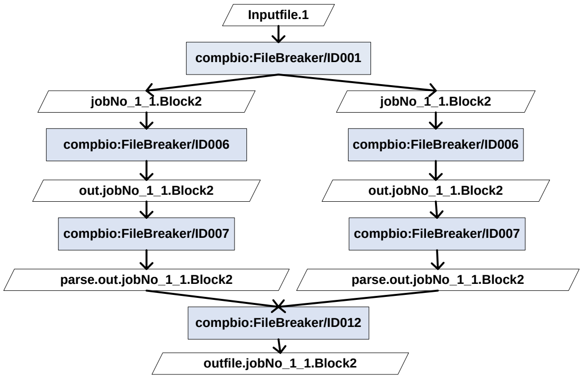

Inputfile.1
compbio:FileBreaker/ID001
jobNo_1_1.Block2
jobNo_1_1.Block2
compbio:FileBreaker/ID006
compbio:FileBreaker/ID006
out.jobNo_1_1.Block2
out.jobNo_1_1.Block2
compbio:FileBreaker/ID007
compbio:FileBreaker/ID007
parse.out.jobNo_1_1.Block2
parse.out.jobNo_1_1.Block2
compbio:FileBreaker/ID012
outfile.jobNo_1_1.Block2

Figure 6. A six-step BLAST workflow with two-way parallelism [17]. The rectangle represents a job and the parallelogram represents data file.

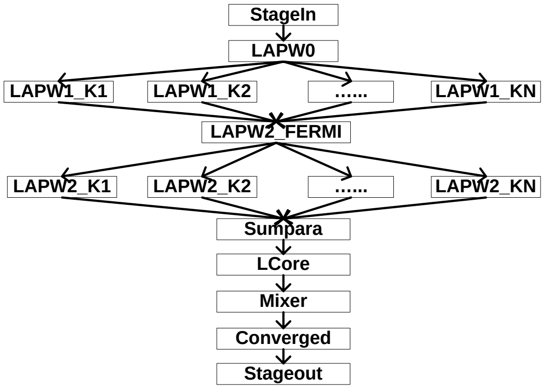


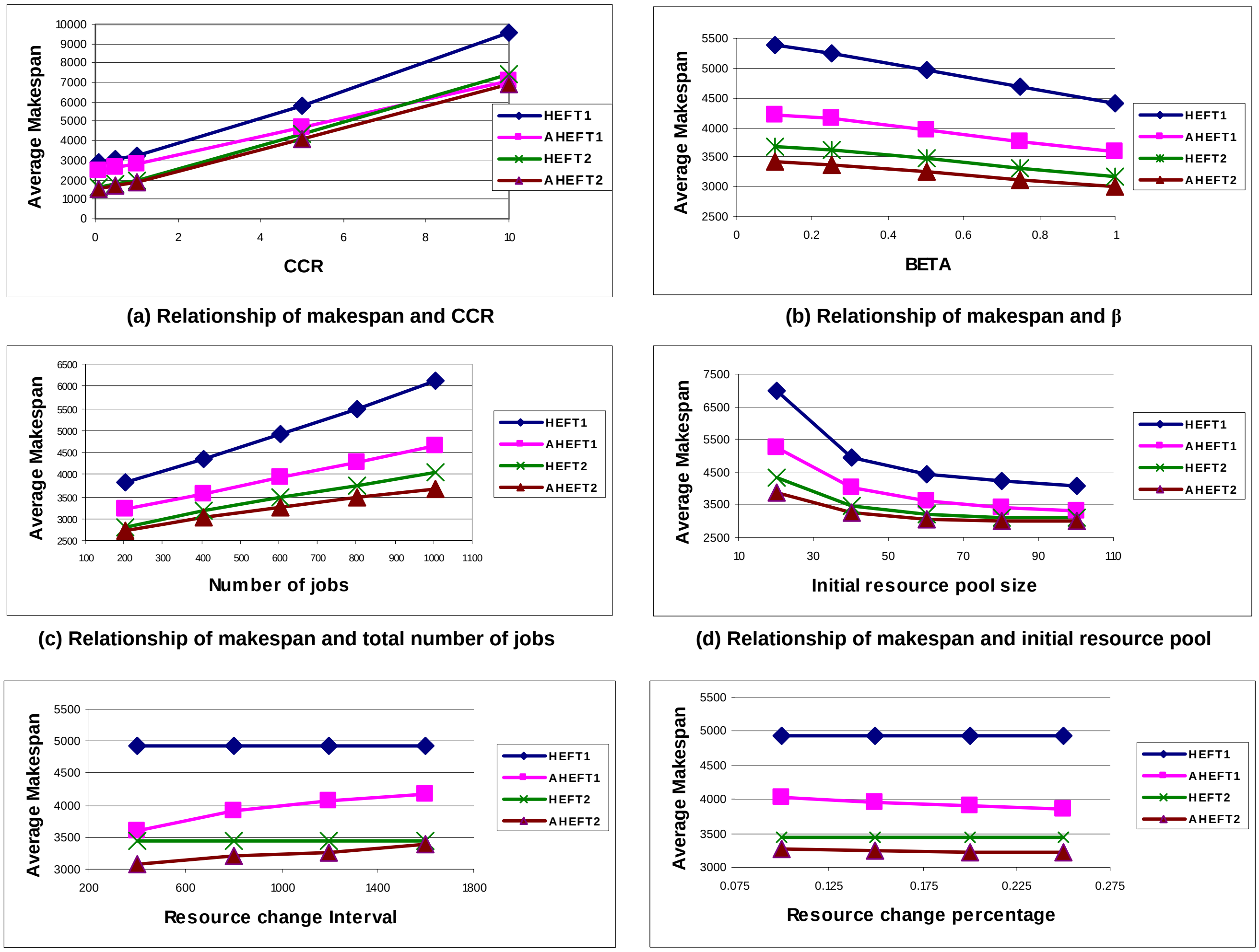
Figure 7. A full-balanced WIEN2K DAG example [20].

Table 5. Parameter values of BLAST and WIEN2K DAGs.

Parameter	Value
v	200, 400, 600, 800, 1000
CCR	0.1, 0.5, 1.0, 5.0, 10.0
β	0.1, 0.25, 0.5, 0.75, 1.0
R	20, 40, 60, 80, 100
Δ	400, 800, 1200, 1600
δ	0.10, 0.15, 0.20, 0.25

Table 6. Average *makespan* and improvement rate by *AHEFT*.

Application	*HEFT*	*AHEFT*	Improvement rate
BLAST	4939.3	3933.1	20.4%
WIEN2K	3451.6	3233.8	6.3%



(a) Relationship of makespan and CCR

(b) Relationship of makespan and β

(c) Relationship of makespan and total number of jobs

(d) Relationship of makespan and initial resource pool

(e) Relationship of makespan and resource change interval

(f) Relationship of makespan and resource change percentage

Figure 8. Relationship of average *makespan* and different parameters. HEFT1: applying *HEFT* on BLAST DAG, AHEFT1: applying *AHEFT* on BLAST DAG, HEFT2: applying *HEFT* on WIEN2K DAG, IHEFT2: applying *AHEFT* on WIEN2K DAG.

the high parallelism degree in each of two sections (*LAPW1* and *LAPW2*) of WIEN2K DAG, the job *LAPW2_FERMI* is the single job on its level, which reduces the parallelism significantly because any job in the *LAPW2* section can not start until this job finishes. It is easy to conceive that extra resources can not help a single job if it can only utilize one resource at a time, which leaves other available resources idle. We further study the correlation between the performance improvement rate and DAG parameters and show them in Fig. 8 from six perspectives: (a) Relationship of *makespan* and CCR; (b) Relationship of *makespan* and β; (c) Relationship of *makespan* and total number of jobs; (d) Relationship of *makespan* and initial resource pool size; (e) Relationship of *makespan* and resource change frequency and (f) Relationship of *makespan* and resource change percentage. To better illustrate how *AHEFT* improves the schedule variously with different DAG parameters, Fig. 8 presents the results of *HEFT* and *AHEFT* for both BLAST and WIEN2K, where HEFT1 and AHEFT1 represent application of *HEFT* and *AHEFT* on BLAST respectively, similarly HEFT2 and AHEFT2 represent for application of *HEFT* and *AHEFT* on WIEN2K respectively. The improvement rate increases as the DAG gets more complex, i.e., the total number of jobs gets bigger, as Table 7 and Fig. 8(c) show. This holds true for both BLAST and WIEN2K applications, and the rate accelerates faster with WIEN2K than BLAST. It implies that adaptive rescheduling is more effective for more complex DAGs. When *CCR* goes up, the improvement rate increases slightly as well, as shown in Fig. 8(a). However the improvement rate increases with BLAST when *CCR* is bigger but is stable for WIEN2K as Table 8 shows. As one can tell by Fig. 8(d), the smaller the initial resource pool is the better *AHEFT* outperforms *HEFT*. But once the initial resource is big enough, the improvement rate becomes stable. Another observation is that, the more dynamic the grid environment is, i.e.,the more frequent the new resource is available, the more efficient *AHEFT* can be. Lastly, the improvement rate is not very sensitive to the parameter of β, i.e., the resource heterogeneous factor, and the percentage of resource change, as Fig. 8(b) and Fig. 8(f) illustrate respectively.

Table 7. Improvement rate with various total number of jobs.

Application	200	400	600	800	1000
BLAST	15.9%	18.3%	19.9%	21.9%	23.6%
WIEN2K	2.2%	4.3%	6.0%	7.8%	9.4%

Table 8. Improvement rate with various CCRs.

Application	0.1	0.5	1.0	5.0	10.0
BLAST	16.1%	15.5%	14.3%	19.1%	26.1%
WIEN2K	7.3%	7.3%	6.6%	5.3%	6.4%

Overall, the adaptive rescheduling algorithm *AHEFT* outperforms the traditional *HEFT* significantly, and it does even better for workflow applications of high complexity, data intensiveness and parallelism degree in the circumstances of high dynamics and low initial resources, which are exactly the essential characteristics of scientific workflow applications on grids.

5 Summary and Future Work

This paper analyzes both the benefits and issues of static scheduling strategy for grid workflow applications, and proposes a novel adaptive rescheduling strategy. The new approach not only addresses the issues with traditional static scheduling but also further exploits its inherent benefits. *AHEFT* is developed and tested for its stability and effectiveness with various DAGs, and the results are promising.

In the future, we will continue working on the design and implementation of the collaboration system model proposed in this paper in both Wayne State University Grid system and TeraGrid. We also intend to integrate the rescheduling with advance resource reservation and resource availability prediction model to implement the collaboration.

References

[1] F. Berman and *et al.*, New grid scheduling and rescheduling methods in the grads project. *International Journal of Parallel Programming*, 33(2):209–229, 2005.

[2] J. Blythe and *et al.*, Task scheduling strategies for workflow-based applications in grids. In *Proc. of CCGRid '05.*

[3] J. Cao, S. A. Jarvis, S. Saini, and G. R. Nudd. Gridflow: Workflow management for grid computing. In *Proceeding of CCGrid '03.*

[4] DAGMan. http://www.cs.wisc.edu/condor/dagman/.

[5] H. Dail and *et al.*, Scheduling in the grid application development software project. In *Grid Resource Management: State of the Art and Future Trends*. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[6] E. Deelman and *et al.*, Workflow management in griphyn. In *Grid Resource Management: State of the Art and Future Trends*, pages 99–116. Kluwer Academic Publishers, 2004.

[7] T. Fahringer and *et al.*, Askalon: A grid application development and computing environment. In *Proc. of 6th International Workshop on Grid Computing,* 2005.

[8] J. Frey and *et al.*, Condor-g: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[9] M. R. Garey and D. S. Johnson. *Computers and Intractibility: A guide to the Theory of NP-completeness.* W. H. Freeman, 1979.

[10] U. Hönig and W. Schiffmann. A comprehensive test bench for the evaluation of scheduling heuristics. In *Proc. of PDCS '04.*

[11] A. S. McGough and *et al.*, Making the grid predictable through reservations and performance modelling. *The Computer Journal*, 48(3):358–368, 2005.

[12] Montage. http://montage.ipac.caltech.edu.

[13] T. Oinn and *et al.*, Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinfomatics*, 20(17):3045–3054, 2004.

[14] R. Sakellariou and H. Zhao. A low-cost rescheduling policy for efficient mapping of workflows on grid systems. *Scientific Programming*, 12(4):253–262, 2004.

[15] SimJava, Division of Informatics, University of Edinburgh. http://www.dcs.ed.ac.uk/home/hase/simjava/.

[16] G. Singh and *et al.*, The pegasus portal: Web based grid computing. In *Proc. of ACM SAC '05.*

[17] D. Sulakhe and *et al.*, Gnare: an environment for grid-based high-throughput genome analysis. In *Proc. of CCGrid '05.*

[18] A. Sulistio, W. Schiffmann, and R. Buyya. Advanced reservation-based scheduling of task graphs on clusters. In *Proc. of HiPC '06.*

[19] H. Topcuouglu, S. Hariri, and M.-Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distribution Systems*, 13(3):260–274, 2002.

[20] M. Wieczorek, R. Prodan, and T. Fahringer. Scheduling of scientific workflows in the askalon grid environment. *SIGMOD Record*, 34(3):56–62, 2005.

[21] WIEN2K. http://www.wien2k.at/.

[22] H. Yu and *et al.*, Plan switching: an approach to plan execution in changing environments. In *Proc. of IPDPS '06.*