A Joint Data and Computation Scheduling Algorithm for the Grid

Fangpeng Dong[1] and Selim G. Akl[1]

[1]School of Computing, Queen's University
Kingston, ON, Canada K7L 3N6
{dong, akl}@cs.queensu.ca

Abstract: In this paper a new scheduling algorithm for the Grid is introduced. The new algorithm (JDCS) combines data transfer and computation scheduling by a back-trace technique to reduce remote data preloading delay, and is aware of resource performance fluctuation in the Grid. Experiments show the effectiveness and adaptability of this new approach in the Grid environment.

Keywords: Workflow, Scheduling, Algorithm, Grid Computing, Data

1 Introduction

The development of Grid infrastructures now allows workflow to be submitted and run on remote Grid resources. At the same time, fledging data Grid projects enable remote data access, replication, and transfer. These advances make it possible to run data-intensive workflows in the Grid. In this paper, we consider the problem of assigning metatasks in a workflow, which is assumed to be represented by an acyclic directed graph (DAG), to Grid resources. The objective is to minimize the total schedule length of the whole workflow (also known as the makespan). As the general form of this optimization problem is NP-Complete [6], heuristic approaches are usually adopted. In the Grid environment, the problem becomes even more challenging: first, the performance of a Grid resource is usually dynamically changing, which makes it harder to get an accurate estimate in advance of a task's execution time; second, input data of a task may cross a long distance from a data storage site to the computational resource. If computational resources and data storage sites are dynamically selected by a Grid scheduler, communication cost for input data transfer may vary according to different data storage and computational resource combinations. This situation is different from the traditional cases where data and computation are located on the same site, or data sites and computational sites are fixed prior to scheduling so that communication cost is a known constant.

Based on the above observations, we propose a workflow scheduling algorithm called JDCS (Joint Data and Computation Scheduling). It considers the possibility of overlapping the time of input data preloading and that of computation, thus reducing the waiting time of a task to be scheduled. This is achieved by using a back-trace technique. To overcome the difficulties brought about by performance fluctuation, JDCS takes advantage of mechanisms such as Grid performance prediction and resource reservation [1], which can capture the resource performance information and provide some kind of guaranteed performance to users. These approaches make it possible for Grid schedulers to get relatively accurate resource information prior to making a schedule, and according to the obtained information, JDCS updates metatask priorities dynamically when it makes a schedule.

The remainder of this paper is organized as what follows. Section 2 introduces models used by JDCS. Section 3 describes JDCS in detail, while Section 4 presents experimental results and a performance analysis. In Section 5, related research is reviewed. Conclusions and suggestions for future work are provided in Section 6.

2 Definitions and Models

A *Grid scheduler* running JDCS receives submitted workflows from Grid users, retrieves resource information from Grid Information Services, creates schedules and commits scheduled metatasks to Grid resources. The Grid has a set of computational resources $\{p_1,\ldots, p_n\}$ and a set of data storage sites $\{d_1,\ldots, d_m\}$. The performance of a computational resource is not only heterogeneous, but also dynamically fluctuating. In a resource management system supporting advance reservation, available resource performance at a specific time can be known by calculating the workload generated by jobs that have reserved resources at that time, as Fig. 1 (a) indicates. Thus, by referring resource predictors or resource management components supporting reservation, the performance fluctuation can be caught. Theoretically, if the time axis can be divided into fine granular periods, the performance within a period can be approximated as a constant. So, to describe the fluctuation, a sequence of time slots $s_1,\ldots, s_k$ is introduced. The computational power of p_i in s_j is denoted as a constant $c_{i,j}$, which can be known from Grid Information Services. To decide the number of time slots which will be used to complete the workflow, an optimistic estimation strategy is used: The scheduler estimates the serial executing time of the entire workflow on each resource, and chooses the smallest one. This strategy is based on the expectation that a parallel processing, even in the worst case, is not worse than the best sequential one.

The network connection between p_i and p_j is denoted as $clink_{i,j}$ and one between p_i and d_j is denoted as $dlink_{i,j}$. The bandwidth of $clink_{i,j}$ or $dlink_{i,j}$ is denoted as $cw_{i,j}$ or $dw_{i,j}$, respectively. For $cw_{i,j}$, if $i = j$, $cw_{i,j} = +\infty$, which implies the communication delay within p_i is 0. Fig. 1 (b) gives an example of the system model. The input data for a metatask is located on a data storage d_j. Computational resources themselves can also provide limited storage capacity that allows input data to be precached prior to the start of the computation. The capacity of the input data cache on p_i is denoted as c_i. Once a metatask is assigned to a computational resource, its input data preloading can start, and once a metatask starts, the input data will be deleted and occupied space will be released. Due to the limitation of cache capacity, it may not be possible for all metatasks to pre-upload their data at the same time to the computational site. Available data cache space on computational resource p_j, at time T is denoted as $ACS(p_j, T)$, which initially equals c_i. To avoid potential conflicts resulting from simultaneous data preloading for different metatasks on the same resource, when a data transfer starts, required space on the destination will be reserved.

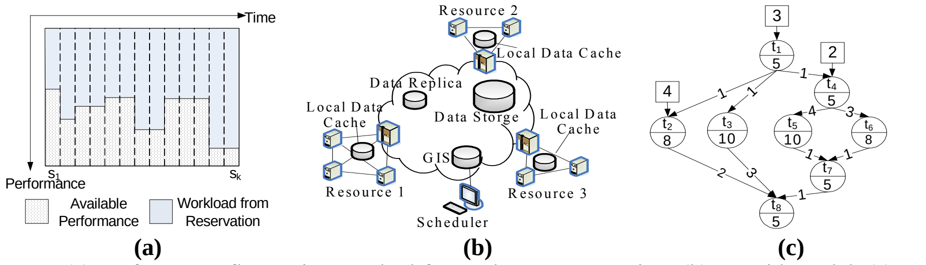

Fig. 1. (a) Performance fluctuation resulted from advance reservation; (b) A Grid Model. (c) A DAG depicting a metatask workflow with raw input data.

We assume that a workflow can be represented by a directed acyclic graph (DAG) G (Fig. 1 (c)). A circular node t_i in G represents a metatask, where $1 \le i \le v$, and v is number of metatasks in the workflow. The computational power consumed to finish t_i is denoted as q_i. An edge $e(i, j)$ from t_i to t_j means that t_j need an intermediate result from t_i, and $t_j \in succ(t_i)$, where $succ(t_i)$ is the set of all immediate successors of t_i. Similarly, we have $t_j \in pred(t_i)$, where $pred(t_i)$ is the set of immediate predecessors of t_i. The weight of $e(i, j)$ gives the size of intermediate results transferred from t_i to metatask t_j. A metatask may also take raw data as input, which originally resides on data storage sites. The raw input data size of metatask t_i is denoted as r_i. We have $D_{i,j} = 1$, if the raw input data of metatask t_i is on storage d_j, and $D_{i,j} = 0$, otherwise. In the example, a square node with an arrow to a metatask node represents the raw data input, and the number indicates its size.

3 JDCS Algorithm

To achieve a feasible schedule with a small makespan, the JDCS algorithm has the following features: (1) It considers the possibility of raw input data preloading that can be overlapping with computation to reduce the waiting time of a metatask. (2) It is aware of data cache capacity and schedules data preloading accordingly in order to avoid conflicts. (3) It updates the rank of a metatask in each scheduling step so that critical metatasks will be recognized dynamically. (4) To avoid myopic decisions, it looks ahead along the current longest unscheduled path to select a resource for the current metatask. (5) In order to use idle time slots on a resource, it can insert an unscheduled metatask before a scheduled metatask on that resource if the insertion does not violate precedence orders and cache space limitation.

3. 1 Metatask Ranking

To schedule a workflow efficiently, it is important to identify the *critical tasks* in each scheduling step. A delay of critical tasks may result in the extension of the schedule length. Usually, the priority of a metatask node can be obtained by finding the maximum "distance" from this node to the starting or exiting node. Here, distance means the sum of computational and communication costs along a certain path. Unfortunately, due to the heterogeneity and fluctuation of resource performance, it is difficult to find how urgent a metatask node is. To estimate the completion time of a metatask in such a scenario, several performance measurements can be used, such as the median or average value of resource performance. In the following discussion, we use the average performance value to demonstrate our algorithm. To obtain the average performance as accurately as possible, only the performance of *feasible time slots* is used. The feasible time *AVLT* of p_i in the *m*th scheduling step is defined as:

$$AVLT_i(m) = \min_{t_j \in RQ}\{EST(t_j, p_i)\} \tag{1}$$

Here $EST(t_j, p_i)$ is earliest start time of t_j on p_i (See (3)). So, $AVLT_i(m)$ is the earliest time when a metatask in the ready queue (*RQ*) can start on p_i at the *m*th scheduling step. Time slots after this time will be considered feasible and the corresponding performance within these slots will be used to update priorities of task nodes. For simplicity, we remove *m* from all expressions, without losing generality. The average performance of p_i along time avg_c_i and the average performance of all available computational resources avg_c are given by the following two equations:

$$avg\_c_i = \frac{1}{k - AVLT_i} \sum_{AVLT_i \le j \le k} c_{i,j} \qquad avg\_c = \frac{1}{n} \sum_{1 \le i \le n} avg\_c_i$$

Similarly, the average unit communication cost between a computational resource p_i and others avg_cd_i, the overall average unit communication cost between any two resources avg_cd and the average unit communication cost between a data storage site d_j and any computational resources avg_dd_j are given by the following equations:

$$avg\_cd_i = \frac{1}{n} \sum_{1 \le j \le n} 1/cw_{i,j} \qquad avg\_cd = \frac{1}{n} \sum_{1 \le i \le n} avg\_cd_i \qquad avg\_dd_j = \frac{1}{n} \sum_{1 \le i \le n} 1/dw_{i,j}$$

We assumed that the time required to complete a metatask on different resources is uniformly related to the performance of resources. So, if the performance of a processing node p_j is a constant *K*, it will finish metatask t_i within time q_i/K. We have the same assumption for the communication cost. These assumptions lead to following equation which relates computational cost, resource performance and time for the performance fluctuating scenario:

$$q_i = (s_s^{end} - s_{start}) \times c_{j,s} / u + \sum_{e=s+1}^{c-1} c_{j,e} + (s_{complete} - s_c^{start}) \times c_{j,c} / u \tag{2}$$

where s_i^{start} and s_i^{end} are the start and end of a time slot, *u* is the length of a time slot, s_{start} and $s_{complete}$ are start time and completion time of t_i, and *s* and *c* are indices of time slots in which t_i starts and completes respectively. Now we can express the priority of a metatask $rank_u$ using a recursive definition:

$$rank_u(t_i) = \frac{q_i}{avg\_c} + \max_{t_j \in succ(t_i)} (TransTime(i, j) + rank_u(t_j))$$

$$TransTime(i, j) = \max(r_j \times \min_{1 \le x \le m \wedge D_{j,x} = 1} (avg\_dd_x), e(i, j) \times avg\_cd)$$

$TransTime(i, j)$ gives the estimated time needed to transfer intermediate results or raw input data to metatask t_j, which is an immediate successor of t_i. Since t_i is not scheduled yet, the time for intermediate results and that for raw input are both estimated using average values. The meaning of $rank_u$ and *TransTime* implies that a conservative policy is used in the ranking method. It assumes that raw input data preloading of t_i will not start from the data storage until t_i's predecessors finish. This conservative policy may assign a higher rank to metatasks with large raw input data, thus giving them higher priority and more chances to get cache space reservation.

As scheduling proceeds, available performance of computational resources will change. Instead of using a static rank value computed at the beginning of the scheduling, JDCS applies a dynamic iterate ranking strategy, that is, once a metatask node is scheduled, the ranks for all of the metatask nodes yet to schedule will be updated using the resource performance in new feasible time slots.

A metatask node cannot start until it receives all of the intermediate results and raw input data. The value of $EST(t_i, p_j)$ is related to two factors: the latest intermediate result from t_i's predecessors, and the ready time of pre-loaded raw input data. According to the restrictions, the raw input data pre-loading of metatask t_i is not relevant to the finish time of t_i's predecessors directly. Ideally, t_i should start as soon as possible when its last predecessor finishes. We denote the time when t_i's last intermediate result can be ready on p_j as $LIRT(t_i, p_j)$, and we can obtain:

$$LIRT(t_i, p_j) = \max_{t_x \in pred(t_i)} (CT(t_x) + e(x, j) / cw_{PA(t_x), j})$$

$CT(t_x)$ is the real complete time of t_x. Raw input data preloading of t_i should finish before $LIRT(t_i, p_j)$, otherwise, t_i will suffer from additional delay. So, intuitively, raw data preloading should start as early as possible. Unfortunately, due to the limited data cache capacity, if raw input data of t_i is preloaded too early, it may occupy space that should be given to metatasks scheduled earlier than t_i. To find out the proper time to start preloading, JDCS uses a trace-back technique. Starting from time $LIRT(t_i, p_j)$, it traces back to check and reserve the earliest available cache space on p_j for the input data of t_i. If there is no space available backward, it will change the search direction to forward. Procedure *Find_Preloading_ StartTime*(t_i, p_j, *time*) in Fig. 2 describes how this works. It returns the earliest preloading start time of t_i on p_j $EPST(t_i, p_j)$. Thus, the earliest raw input data ready time $ERRT(t_i, p_j)$ and $EST(t_i, p_j)$ can be formulated as:

$$ERRT(t_i, p_j) = EPST(t_i, p_j) + \min_{1 \le x \le m \wedge D_{j,x}=1} (r_i / dw_{j,x})$$

$$EST(t_i, p_j) = \max(LIRT(t_i, p_j), ERRT(t_i, p_j)) \qquad (3)$$

For simplicity, the *while* loop iterates using integer time, but actually, the value of *ACS* only changes when a new data transfer is started or space is released. In the worst case, the *while* loop will run $O(v*L)$ times, where L is a constant indicating the maximum degree of a metatask graph. According to the restrictions and cache management policy, when a metatask node is scheduled, all of its predecessors have been already scheduled, so the scheduler can know precisely what the cache space status was at any time in the past. Therefore, tracing back and inserting input data is not going to conflict with any other scheduled metatasks, or result in deadlock.

With $EST(t_i, p_j)$ and performance of p_j in different times slots, it is straightforward to get the earliest complete time of t_i on p_j $ECT(t_i, p_j)$, according to (2) where s_{start} = $EST(t_i, p_j)$ and $s_{complete}$ = $ECT(t_i, p_j)$. At each step, the scheduler chooses the unscheduled metatask which has the highest dynamic rank.

3.2 Resource Assignment

In the second phase, the scheduler selects a resource for a metatask. Intuitively, a "good" assignment for a metatask should be the resource that can complete it the earliest. But the problem of this intuitive way is that it may lead to a local optimum. For example, we can consider the following case with a sequence of metatasks that forms a path P from metatask t_a to an exit node. It can happen that after t_a is assigned to a resource p_x that can finish t_a the earliest, the rest of metatask nodes on P also have to be assigned to p_x because the communication cost between any of them might delay their earliest complete time otherwise. But, if they are scheduled to another resource, say p_y, it is possible that the execution of the remaining metatasks on P can make up the communication delay because p_y has a faster computational speed. So instead of using the simple earliest complete time strategy, JDCS adopts a look-ahead approach, which is described by *Rule* 1 to avoid a biased schedule.

Rule 1*:* If t_i is the current metatask to be scheduled and t_j is the direct child of t_i on the longest path P_{t_i} measured by task rank from t_i to an exit node, then t_i should be scheduled to resource p_x which satisfies:

$$\min_{1 \le x,y \le n} \{PEST(t_j) + EPT(P_{t_i}, p_y, PEST(t_j))\}$$

where $PEST(t_j) = ECT(t_i, p_x) + \max(e(i,j)/cw_{x,y}$and $EPT(P, p_i, T) = (\sum_{t_i \in P} q_j) / \sum_{T \le j \le k} c_{i,j} /(k - T)$.

We cannot know the real earliest start time of t_j, since at least one predecessor of t_j has not been scheduled yet. Function *EPT* computes the estimated total execution time of metatasks on path P_{t_i} using the average performance of each resource after $PEST(t_j)$. Therefore, *Rule* 1 states that instead of finding the processing node that can finish t_i the earliest, we are trying to find a pair of resources, p_x and p_y, so that execution time for the metatasks on the longest path from t_i to an exit node will be minimized.

To utilize idle time slots on resources, JDCS allows task insertion, which is formalized by *Rule* 2. It states that a task can be assigned to a resource only if there are time slots large enough to accommodate it without delaying metatasks already scheduled or violating precedence orders among the metatasks.

Rule 2: A metatask t_i can be assigned to resource p_j which is already assigned with a sequence of metatasks $\{t_{j_1}, t_{j_2}, \ldots, t_{j_n}\}$ at time s, if there is some m such that for every metatask t_{j_x} in $\{t_{j_1}, t_{j_2}, \ldots, t_{j_m}\}$, $ECT(t_{j_x}, p_j) \le EST(t_i, p_j)$, and for every metatask t_{j_y} in $\{t_{j_m}, t_{j_{m+1}}, \ldots, t_{j_n}\}$, $ECT(t_i, p_j) \le EST(t_{j_y}, p_j)$.

The pseudo code of the JDCS algorithm is shown in Fig.2. The total complexity of JDCS is in the order of $O(n^2v^2+nkv)$.

4 Experiments

Simulation experiments are conducted to evaluate the performance of JDCS in the Grid. The performance metric we used for the comparison is the Scheduled Length Ratio (*SLR*), defined as the ratio of the real makespan to the lower bound of any possible scheduling, which is the minimum length of the critical path. In the experiments, the basic topology of Grid resources is generated using GridG1.0 [3]. This tool allows us to get realistic computational resources and networking settings for simulation, such as processing capability, data cache size, bandwidth and delay for LAN and WAN. Grid data storage sites are assumed to follow a uniform geographic distribution in the Grid. Performance of a computational resource is assumed to follow a Poisson distribution having the average value given by GridG as the average. TGFF [4] is used to generate basic DAGs. TGFF allows customized settings such as average number of metatask nodes in the graph, average out-degree and in-degree for a node, and range of computation and communication costs. However, TGFF cannot generate raw input data for a metatask. As a result, a graph generated by TGFF is reprocessed and raw data input is inserted randomly to metatasks in the graph. The size of raw input data is also uniformly distributed in the same range given to TGFF for cost generation.

```
Find_Preloading_StartTime($t_i$, $p_j$, time)
1. EPST = time;
2. if (ACS($p_j$, PST) > = $r_i$){
3.   While (ACS(p, EPST) > = $r_i$)
4.     EPST--;
5. }else
6.     while (ACS(p, EPST) < $r_i$)
7.       EPST++;
8. Return EPST
□
Algorithm JDCS(DAG G)
1. Compute initial task rank$_u$;
2. Initialize the ready queue RQ
   with the entry metatask;
3. While(there are unscheduled
   metatasks){
4.   Select the highest rank
     metatask t in RQ;
5.   Call Select_Resource(t);
6.   Update rank$_u$;
7. }
□
Select_Resource (metatask t)
1. Find the longest path P from t;
2. For available resources $p_i${
3.   Get EST(t,$p_j$) and ECT(t,$p_i$);
4.   For available resources $p_j$
5.     Call EAT(P, $p_j$, PEST(t));
6. }
7. Assign t to $p_i$ that satisfies
   Rule 1;
8. Set Data transfer time for t
   and write the cache log of $p_i$;
9. For all available resources $p_i$,
     update $AVLT_i$ of $p_i$.
□
```

Fig. 2. Pseudo code of JDCS.

To test JDCS in different resource settings and workflow patterns, six groups of experiments are conducted to test the influence of the following parameters to schedule results: 1) the number of input data replicas (*DR*) in the Grid; 2) the ratio of the average data input size to the average data cache size (*ICR*) on computational resources, 3) the ratio of the average degree of a node to the total number of metatasks in a graph (*DTR*); 4) the percentage of metatasks having raw input data (*RIP*) in a workflow, 5) the average ratio of computation cost to communication cost a metatask graph (*CCR*) (a high *CCR* value means a metatask graph is computation-intensive); and 6) the resource background workload factor (*BWF*) which decides the percentage that the performance of a computational resource can increase or drop in different time slots. For every group of experiments, we used five sets of workflows whose average metatask number varied from 20 to 100.

To test how the number of replicas of input data in the Grid will influence scheduling results, three different settings are compared, namely, (1) no replica, which means each metatask only has one copy of raw input data on data storage sites, (2) one replica, and (3) two replicas. As expected (Fig. 3 (a)), data replication benefits the schedule because the scheduler has more choices to reduce the preloading cost. But the small margin between the curves of one replica and two replicas implies that the gain from increasing the numbers of replicas is limited if they are evenly distributed in the network. We also test the outcome of the original HEFT [3] algorithm, which only considers intermediate results but not raw input data preloading when it makes a schedule, and a revised HEFT algorithm using the trace-back method introduced in this paper. The results show that the original approach will bring significant lag to the performance, which supports the basic motivation of our work in this paper.

Another element impacting scheduling results is how large the data cache of computational resources is. To make results comparable, we use a normalized ratio of input data size to cache size (Fig. 3(b)). It can be observed that as the ratio increases, *SLR* is increased dramatically and slopes of curves turn from sub-linear to super-linear. The explanation is that as the ratio is higher, it becomes more difficult for a metatask to get a free data cache space to preload its input data. In particular, when the ratio is higher than 0.5, in most cases, simultaneous preloading is impossible because a data cache can only hold input data for one metatask.

Intermediate results in a workflow are also competing for data cache space on computational resources. So the ratio of intermediate data size to the data cache size also influences a schedule. Given a uniform distribution of intermediate result size when a metatask graph is generated, the total size of intermediate results is proportional to the number of edges in the graph, or the degrees of metatask nodes. To describe the edge density in a graph, the ratio of the average degree of each node to the total number of nodes is used in our experiments (Fig. 3 (c)). Results show that the performance of JDCS is stable in different ratios. Intuitively, *SLR* will be higher as the size of intermediate results increases. But increasing the degree of metatasks implies the task graph is more connected and the length of the path to an exit node might become shorter. As JDCS uses a look-forward strategy in the resource selecting phase, a shorter path means a more accurate estimation of the execution time, which will benefit the task assignment. Another reason is given in the following analysis.

The third element relating to data cache competition is the number of metatasks having raw input data. Different percentages of the metatasks having raw input data in a workflow are tested (Fig. 3 (d)). It can be observed that, as the percentage increases, gaps between different curves increase, which implies that JDCS is more sensitive to changes in the number of raw input data than that of intermediate results. The reason behind this is that the timing of intermediate results transfer is more restricted by both precedence orders among metatasks and cache size limitation, while the raw input data preloading solely depends on cache availability.

In our experiments, we assume *CCR* only influences the ratio of computation to intermediate data, but not the raw input data. Fig. 3 (e) shows that, as *CCR* increases, the *SLR* drops significantly. The reason behind is that the more communication in the workflow, the higher *SLR* is going to be. But as *CCR* reaches a point in our model, the main contribution of the communication cost will come from raw input data preloading, which will then limit the drop of *SLR*.

The last group of experiments is to discover how JDCS adapts to performance fluctuation of computational resources. In the experiments, we allow the background workload of a computational resource to increase up to a certain percentage of its full performance in different time slots (Fig. 3 (f)). It can be observed that, as the fluctuation in performance grows, *SLR* is going to be extended. However in every workflow set, the increase remains stable and moderate. This implies that JDCS can work well even with resource performance changes in a wide range of up to 80%.

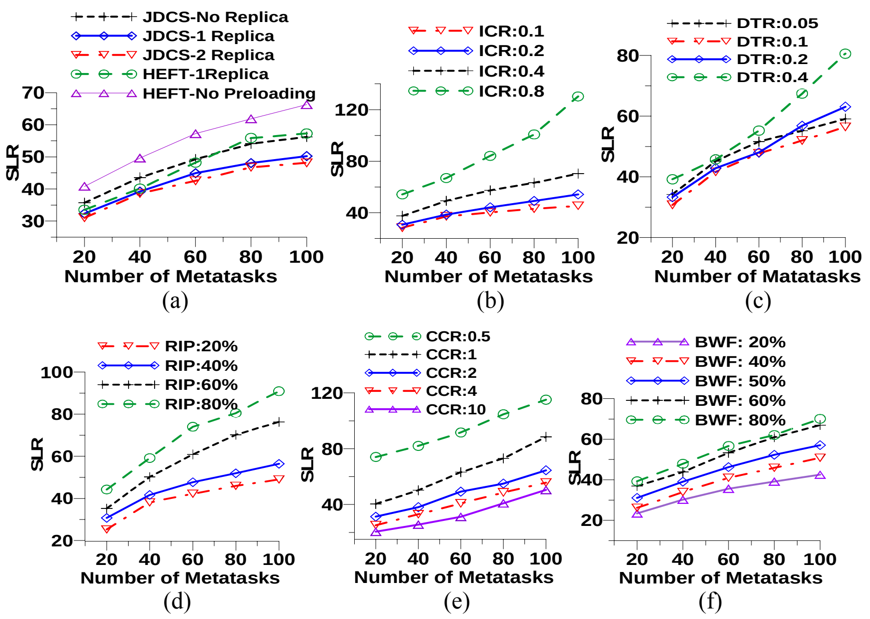

Fig. 3. Performance on different settings of: (a) DR (ICR=0.2, DTR =0.2, RIP=40%, CCR = 2, BWF=50%). (b) ICR (DR=1, DTR =0.2, RIP=40%, CCR = 2, BWF=50%). (c) DTR (DR=1, ICR=0.2, RIP=40%, CCR = 2, BWF=50%); (d) RIP (DR=1, DTR =0.2, ICR=0.2, CCR = 2, BWF=50%), (e) CCR (DR=1, DTR =0.2, ICR=0.2, RIP=40%, BWF=50%) and (f) BWF (DR=1, DTR =0.2, ICR=0.2, RIP=40%, CCR =2).

5 Related Works

In [7], Park classifies different scenarios about locations of data and computation in the Grid into five different categories. JDCS belongs to the category of Remote Data and Different Remote Execution, because both computation and data sites are selected dynamically and not necessarily close to each other. We believe this is a more general case compared with the other four patterns in the Grid. There are two different points of view on the relationship between data scheduling and computation scheduling, namely, decoupling the two kinds of scheduling or combining them. In a decoupled method given in [8], data replication strategies are independent from task scheduling strategies, and a complete scheduling algorithm is an arbitrary combination of the two. The primary goal of that approach is to maximize the system throughput. By contrast, the goal of JDCS is to optimize the finish time of each Grid workflow, and the algorithm does not generate new data replication in scheduling procedures. Scheduling of computation is combined with data replica selection in [9] to reduce the execution time of a collection of applications which can be unified to compose a DAG. The goal of the approach is the same as JDCS, but main differences include: 1) JDCS focuses on finding the right time to start a data input preloading other than selecting a replica; and 2) JDCS does not partition a DAG graph to make a schedule. The algorithm in [10] considers the constraint of storage capacity of each computation site when it tries to optimize the throughput of a Grid system for independent jobs each of which requests to refer to a certain data set. The differences from JDCS includes taking independent jobs as input and system throughput as objective function. Finally, the algorithm in [11] introduces economic cost as a part of the objective function for data and computation scheduling. It does not use data replication, and is designed for independent Grid jobs as well.

6 Conclusions

A joint data and computation scheduling algorithm for Grid workflow is proposed in this paper. JDCS considers realistic situations of workflow scheduling in the Grid, such as limited data cache space on computational resources and resource performance fluctuation. Experiments and analysis verify the effectiveness of JDCS under different system and workflow settings and support our basic motivation for this research. The current implementation of JDCS does not consider the possibility of wrong performance prediction, which is likely in real situations. This is the focus of our current research. Future work also includes improving the resource selection algorithm in the second phase, in order to make it more adaptive to performance fluctuations as we did in [12].

References

[1] L. Yang, J. M. Schopf and I. Foster, Conservative Scheduling: Using Predicted Variance to Improve Scheduling Decisions in Dynamic Environments, Super-Computing, 2003.

[2] K. Aggarwal and R. D. Kent, An Adaptive Generalized Scheduler for Grid Applications, the 19th HPCS, 2005.

[3] H. Topcuoglu, S. Hariri and M.Y. Wu, Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing, IEEE Trans. on Parallel and Distributed Systems, Vol. 13, No. 3, 260 - 274, 2002.

[4] D. Lu and P. Dinda, Synthesizing Realistic Computational Grids, Super-Computing 2003.

[5] R.P. Dick, D.L. Rhodes and W. Wolf, TGFF Task Graphs for Free, the 6th. International Workshop on Hardware/Software Co-design, 1998.

[6] H. El-Rewini, T. Lewis, and H..Ali, Task Scheduling in Parallel and Distributed Systems, ISBN: 0130992356, PTR Prentice Hall, 1994.

[7] S. Park and J. Kim, Chameleon: a Resource Scheduler in a Data Grid Environment, the 3rd CCGrid, 2003.

[8] K. Ranganathan and I. Foster, Decoupling Computation and Data Scheduling in Distributed Data-Intensive Applications, the 11th HPDC, 2002.

[9] A. H. Alhusaini, V. K. Prasanna, and C. S. Raghavendra, A Unified Resource Scheduling Framework for Heterogeneous Computing Environments, the 8th HCW Workshop, 1999.

[10] F. Desprez and A. Vernois, Simultaneous Scheduling of Replication and Computation for Bioinformatic Applications on the Grid, in Proc. of CLADE 2005.

[11] S. Venugopal and R. Buyya, A Deadline and Budget Constrained Scheduling Algorithm for eScience Applications on Data Grids, 6th ICA3PP, 2005.

[12] F. Dong and S. Akl, PFAS: A Resource Performance Fluctuation Aware Workflow Scheduling Algorithm for Grid Computing, the 16th HCW Workshop, 2007.