PFAS: A Resource-Performance-Fluctuation-Aware Workflow Scheduling Algorithm for Grid Computing

Fangpeng Dong and Selim G. Akl
School of Computing, Queen's University
Kingston, ON Canada, K7L 3N6
{dong, akl}@cs.queensu.ca

Abstract

Resource performance in the Computational Grid is not only heterogeneous, but also changing dynamically. However scheduling algorithms designed for traditional parallel and distributed systems, such as clusters, only consider the heterogeneity of the resources. In this paper, a workflow scheduling algorithm, called PFAS, is proposed and tested in the Grid environment. PFAS considers dynamic resource performance fluctuation in the Grid, and conducts the scheduling according to its knowledge of the fluctuation. This new algorithm works in an offline way which allows it to be easily set up and run with less cost. Simulations show that our approach can achieve better schedules than the HEFT algorithm.

1. Introduction

The development of Grid infrastructures, e.g., Pegasus [1], Grid Flow [2] and ASKALON [3] now enables workflow submission and execution on remote computational resources. To exploit the non-trivial power of Grid resources, effective task scheduling approaches are necessary. In this paper, we consider the scheduling problem of workflows which can be represented by directed acyclic graphs (DAG) in the Grid. The ultimate goal guiding the mapping is to reduce the total completion time of all tasks (also known as *makespan*) in a workflow.

As most Grid resources are not dedicated to Grid users, Grid resource performance is not only heterogeneous, but also dynamically changing due to the competition among the uses. Therefore, some mechanisms are introduced to try to capture relevant information about resource performance fluctuation information (e.g., performance prediction [4]), or try to provide some guaranteed performance to users (e.g., resource reservation [5], [6]). These approaches make it possible for Grid schedulers to get relatively accurate resource information prior to producing a schedule, though resource performance fluctuation still makes task scheduling in the Grid more difficult compared with that in traditional parallel and distributed systems such as clusters, in which resource performance is usually heterogeneous, but static for a user. In this paper, we propose a workflow scheduling algorithm called PFAS for the targeted Grid environment. Basically, PFAS is a list heuristic. Although PFAS works in an offline manner, it can be aware of resource performance fluctuation in the Grid, and adopts a dynamic task ranking method in the scheduling procedures and a look-forward technique [7] to find proper task assignments. Experiments show that with the help of these two techniques, PFAS outperforms the well-known and frequently referenced HEFT scheduling algorithm [8].

The rest of this paper is organized as follows: in Section 2, related work is introduced; Section 3 presents the application and resources model used by the proposed algorithm; Section 4 describes the PFAS algorithm in detail; Section 5 presents simulation results and analysis; finally, conclusions are given in Section 6

2. Related Work

The DAG-based task graph scheduling problem in parallel and distributed computing systems is an interesting research area, and algorithms for this problem keep evolving with computational platforms, from the age of homogeneous systems, to heterogeneous systems and today's computational Grids [12]. Due to its NP-complete nature [13], most of algorithms are heuristic based and can be classified into three categories: list algorithms, clustering algorithms and task duplication based algorithms.

In list algorithms, tasks are assigned with priority values and scheduled in the order of decreasing priority values. The HEFT algorithm [8] and the Dynamic Critical

1-4244-0910-1/07/$20.00 ©2007 IEEE

Path algorithm (DCP) [7] are typical examples of list heuristics. Clustering is a way to reduce communication delay in DAGs by clustering tasks heavily communicating with each other to the same subgraph, and then assigning a subgraph to the same processor. Clustering algorithms have two phases: the task clustering phase that partitions the original task graph into subgraphs, and a post-clustering phase which can refine the clusters produced in the previous phase and get the final task-to-resource map. Examples of this kind of heuristics can be found in [13] and [14]. The main idea of duplication based scheduling is utilizing resource idle time to duplicate predecessor tasks. This may avoid the transfer of results from a predecessor to a successor, thus reducing the communication cost. In [15] and [16], two duplication-based scheduling algorithms are proposed for distributed-memory systems with homogeneous processors, and networks of heterogeneous processors, respectively. The problem of these algorithms is they take the resource performance as a constant during the execution of the job to be scheduled, which is usually not the case in the Grid.

The Extended Dynamic Critical Path algorithm (xDCP) [17] enhances the DCP algorithm to adapt to the dynamic and heterogeneous nature of Grid resources. But xDCP didn't use resource performance prediction explicitly to refine the schedule. In [18], a DAG scheduling algorithm considering background workload in multiclusters is proposed. In the targeted system, every resource has multiple processors and its own independent local scheduler, which is similar with the resource model in this paper. But it assumes that processors in the same resource cluster are homogeneous and share a local First-Come-First-Served queue, which is not an assumption of our approach. Another major difference is that the scheduler works in an online manner, that is, the Grid scheduler watches all queues of resource clusters and decides where the next schedulable task should go dynamically.

3. Models and Definitions

3.1 Resource Model

As mentioned previously, the dynamic performance fluctuation of processing nodes is considered. 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 indicates. Thus, by referring resource predictors or resource management components supporting reservation, the performance fluctuation can be caught. Theoretically, if the time axe can be divided into fine granular periods, the performance within a period can be approximated as a constant. Thus, to describe the fluctuation, a sequence of

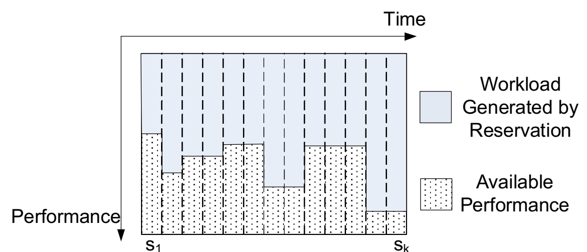

Fig. 1: Performance fluctuation resulted from advance reservation on a resource along time.

time slots $s_1,\ldots,s_k$ is introduced. The processing capability of p_i in time slot s_j is denoted as $c_{i,j}$, and we assume that in a time slot, $c_{i,j}$ is a constant. In terms of communication delay, the communication cost of a data unit along a connection $l_{i,j}$ is denoted as $w_{i,j}$ which is also a constant along time, and $w_{i,j} = 0$ if $i = j$.

3.2 Application Model

We assume that a workflow to be scheduled can be represented by a DAG G, as shown in the example of Fig. 2. A circular node t_i in G represents a task, where $1 \le i \le v$, and v is number of tasks in the workflow. q_i ($1 \le i \le v$) is the computational power consumed to finish t_i. For example, in Fig. 2, $q_1 = 5$. An edge $e(i, j)$ from t_i to t_j means that t_j need an intermediate result from t_i, so that $t_j \in succ(t_i)$, where $succ(t_i)$ is the set of all immediate successor tasks 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 (or communication for simplicity) transferred from t_i to task t_j. For example, the communication volume from t_1 to t_2 is 1 in Fig. 2.

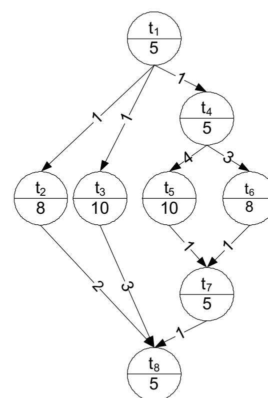

Fig. 2: A DAG depicting a workflow application, in which a node represents a task, and a labelled directed edge represents a precedence order with a certain size of intermediate result transfer.

When a feasible schedule is computed, the following additional restrictions apply: 1) Only one task can run on a processing node at the same time; 2) A task cannot begin until it gets all of its intermediate result. According to this restriction, we need to define the earliest start time (EST) of a task t_i on processing node p_j as:

$$EST(t_i, p_j) = \max_{t_x \in pred(t_i)} \{CT(t_x) + w_{PA(t_x),j} \times e(x,i)\}$$

Here $CT(t_x)$ is the completion time of t_x and $PA(t_x)$ is the processor to which t_x is assigned. All notations used in the algorithm description and their meanings can be found in Table 1.

4. PFAS Algorithm

The primary objective of PFAS is to assign tasks in a workflow to proper computational resources to minimize the makespan. To achieve this goal in dynamic heterogeneous environments, the proposed algorithm has the following features: (1) It updates the ranks of task nodes in real time in each scheduling step so that the critical path can be recognized dynamically. (2) To avoid a myopic optimization, it looks forward along the current recognized critical path when selecting a resource for the current task node. (3) To use idle time slots on a resource, it can insert an unscheduled task before a scheduled task on the same resource if the insertion doesn't violate precedence conditions.

4.1 Task Node Ranking

The critical path (CP) of a task graph is a set of nodes and edges, forming a path from an entry node to an exit node, and all of the nodes (called critical nodes) on this path have the same maximum rank value (See Equation (9)). As the scheduling proceeds, the CP of a task graph might be changed because of the following three reasons: 1) the communication cost between two conjunctive task nodes will be set to zero, if they are assigned to the same resource; 2) the execution time and completion time of a node can be estimated after it is scheduled; 3) the available time slots of a resource are to be changed once a task node is assigned to the resource. So, instead of using a static rank value computed at the beginning of the schedule, PFAS adopts a dynamic ranking strategy, that is, once a task node is scheduled, the ranks for all of the other nodes will be updated. At each scheduling step, the scheduler chooses the unscheduled task which has the highest dynamic rank. And staring from this node, the scheduler also constructs a *dynamic critical path* (*DCP*). To update the rank of a task node, average performance of processing nodes in *feasible time slots* is used. The feasible time *AVLT* of processing node p_i in the *m*th scheduling step is defined by the following Equation:

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

$AVLT_i(m)$ finds the earliest time when a processing node could run a task in the ready queue in 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 omit *m* from all expressions, without losing generality.

To evaluate the performance of a resource as accurately as possible, the scheduler first needs to estimate the number of time slots which will be used to complete the job (the value of parameter k in $s_1, \ldots, s_k$). In PFAS, an optimistic estimation strategy is used: The scheduler estimates the serial processing time of the whole job on each processing node respectively, and chooses the smallest one. This strategy is based on the expectation that the parallel processing, even in the worst case, is not worse than the best sequential one.

To schedule a task graph efficiently, it is important to identify the critical tasks to be scheduled at each step. The delay of critical nodes may result in the extension of the schedule length. Usually, the priority of a task node can be obtained by finding the maximum "distance" from this node to the starting nodes and 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 very difficult to find how urgent a task node really is due to the variation of completion times of its successive tasks on different processing nodes and in different time slots. To estimate the completion times of nodes in such a scenario, several performance measurement can be used, such as using the median [9] or average value of resource performance. In the following discussion, we use the

Table 1: Symbols and Definitions.

Symbol	*Meaning*
$Rank_u(t_i)$	Upward rank of task t_i
$Rank_d(t_i)$	Downward rank of task t_i
$Rank(t_i)$	Total rank of task t_i
DCP_i	Dynamic critical path at scheduling step *i*.
$PA(t_i)$	The processor to which task t_i is assigned
$EST(t_i, p_j)$	The earliest start time of t_i on processor p_j
$ECT(t_i, p_j)$	The earliest complete time of t_i on processor p_j
$EPT(P, p_j, T)$	The estimated execution time of task nodes on path *P* on p_j starting from time *T* based on average performance of p_j.
$CT(t_i)$	Complete time of t_i after it is scheduled
$RT(t_i)$	Execution time of t_i on the processor where it is scheduled
RQ	The ready task queue
$AVLT_i(m)$	The earliest available time of processing node p_i at the *m*th scheduling step

average performance value to demonstrate our algorithm. The average performance of p_i in different slots, denoted by avg_c_i, is given by Equation (1), where k is the number of time slots used to conduct the scheduling.

$$avg_c_i = \frac{1}{k - AVLT_i} \sum_{AVLT_i \le j \le k} c_{i,j} \quad (1)$$

The average performance of all available computational resources, denoted by avg_c, is given by Equation (2), where n is the number of processing nodes.

$$avg_c = \frac{1}{n} \sum_{1 \le i \le n} avg_c_i \quad (2)$$

Similarly Equation (3) and (4) give the average communication cost of p_i to all other nodes avg_w_i, and the overall average communication cost avg_w, respectively.

$$avg_w_i = \frac{1}{n} \sum_{1 \le j \le n} w_{i,j} \quad (3)$$

$$avg_w = \frac{1}{n^2} \sum_{1 \le i,j \le n} w_{i,j} \quad (4)$$

It is assumed that the time required to complete a task on different processors is uniformly related to the performance of resources, that is to say, if the performance of a processing node p_j is a constant c_j, it will finish task t_i within time q_i / c_j. Following this assumption, Equation (5) relates computational cost, resource performance and time for the performance fluctuating scenario:

$$q_i = \frac{(s_s^{end} - s_{start})}{u} \times c_{j,s} + \sum_{e=s+1}^{c-1} c_{j,e} + \frac{(s_{complete} - s_c^{start})}{u} \times c_{j,c} \quad (5)$$

where s_i^{start} and s_i^{end} are the start and end of a time slot, u is the length of time slot, s_{start} and $s_{complete}$ are start time and completion time of t_i, and s and c are indexes of time slots in which t_i starts and completes, respectively. With $EST(t_i, p_j)$ and the 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 Equation (5) where $s_{start} = EST(t_i, p_j)$ and $s_{complete} = ECT(t_i, p_j)$.

Now we can define the priority of a task node in G, which is decided by its *upward* rank $rank_u$ and *downward* rank $rank_d$. To recognize the critical path dynamically, the rank of a node needs to be updated in different scheduling steps. $rank_u$ is computed recursively from the exit node upward to the entry node. If a node t_i is not scheduled yet, $rank_u(t_i)$ is defined as:

$$rank_u(t_i) = \frac{q_i}{avg_c} + \max_{t_j \in succ(t_i)} (e(i,j) \times avg_w + rank_u(t_j)) \quad (6)$$

In this case, since t_i is not scheduled, its execution time and the delay of sending intermediate results to its successors are given by estimate using average resource performance. If t_i has been scheduled, the real run time of t_i $RT(t_i)$ is known, but a successor of t_i might not scheduled yet. So in this case $rank_u(t_i)$ is defined as:

$$rank_u(t_i) = RT(t_i) + \max_{t_j \in succ(t_i)} (TransTime(i,j) + rank_u(t_j)) \quad (7)$$

$$TransTime(i,j) = \begin{cases} (e(i,j) \times w_{PA(t_i),PA(t_j)} & \text{if } t_j \text{ is scheduled} \\ e(i,j) \times avg_w_{PA(t_i)} & \text{otherwise} \end{cases}$$

If t_i is scheduled, the intermediate result transfer time from t_i to t_j is known, otherwise, the average communication cost of the processing node of t_i is used.

Similarly, $rank_d(t_i)$ is computed recursively from the entry node downward to the exit node. If t_i has been scheduled, all of t_i's predecessors have been scheduled too. So, in this case, $rank_d(t_i)$ is defined as:

$$rank_d(t_i) = \max_{t_j \in pred(t_i)} \{rank_d(t_j) + RT(t_j) + e(j,i) \times w_{PA(t_j),PA(t_i)}\} \quad (8)$$

Specially, for the entry node t_1, $rank_d(t_1) = 0$. If t_i has not been scheduled yet, we need to consider whether its predecessors have been scheduled or not, so there are still two cases:

$$rank_d(t_i) = \max_{t_j \in pred(t_i)} \begin{pmatrix} rank_d(t_j) + RT(t_j) + e(j,i) \times avg_w_{PA(t_j)}, t_j \text{ scheduled} \\ rank_d(t_j) + \frac{q_j}{avg_c} + e(j,i) \times avg_w, \text{ otherwise} \end{pmatrix}$$

The rank of a task node is defined as the sum of its upward and downward ranks:

$$rank(t_i) = rank_u(t_i) + rank_d(t_i) \quad (9)$$

4.2 Processing Node Selection

After a critical task node is selected, the scheduler needs to find an appropriate time point on a computational resource to which the task node will be assigned. To utilize idle time slots as much as possible, the scheduler uses an insertion-based method, which can be formalized by the following rule.

Rule1: A task t_i can be inserted into processor p_j which contains a sequence of tasks $\{t_{j_1}, t_{j_2}, \ldots, t_{j_n}\}$at time s, if there is some m that for every task 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 task t_{j_y} in $\{t_{j_m}, t_{j_{m+1}}, \ldots, t_{j_n}\}$, $ECT(t_i, p_j) \le EST(t_{j_y}, p_j)$.

Rule1 states that a task can be inserted on a processor only if there is are time slots large enough to accommodate it without delaying tasks already scheduled or violating precedence orders among the tasks.

Intuitively, high priority tasks should be assigned to resources within their high performance time slots. The problem of selecting a processing node that only gives the earliest complete time for the current task is that this method is myopic and may fall into a local optimization. For example, it can happen that after the task t_a is assigned to a resource p_x, its successors on the critical path P starting from task t_b also has to be assigned to p_x because the intermediate result transfer between t_a and t_b might delay the earliest complete time of t_b otherwise. But actually, even t_b itself is delayed on another resource, say

p_y, it is possible that the execution of remaining tasks on P can make up this delay because p_y may have a faster computational speed. So instead of using the simple myopic earliest complete time strategy, PFAS adopts a look-forward approach to avoid a biased schedule that only considers the current task and resource status. To this end, we first define a function $EPT(P, p_i, T)$ which gives the estimate execution time of a partial path P on resource p_i, starting at time T.

$$EPT(P, p_i, T) = \frac{\sum_{t_j \in P} q_j}{\sum_{T \le j \le k} c_{i,j} / (k-T)}$$

Rule 2: If t_i is the current task 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 the processing node p_x which satisfies

$$\min_{1 \le x, y \le n} \{PEST(t_j) + EPT(P_{t_i}, p_y, PEST(t_j))\} \quad (10)$$

$$PEST(t_j) = ECT(t_i, p_x) + w_{x,y} \times e(i, j)$$

$PEST(t_j)$ is the earliest time when a partial critical path P_{t_i}, starting with t_j can possibly start, and function EPT computes the estimated execution time of tasks on P_{t_i} using the average performance of each processing node after this time point. So, *Rule 2* states that instead of only finding the processing node that can finish t_i the earliest, PFAS is trying to find a pair of processing nodes, p_x and p_y, so that execution time for the tasks on the longest path from t_i to an exit node will be minimized.

Now we can present the pseudo codes of the PFAS algorithm. The time complexity of PFAS will be calculated as what follows: The *while* loop on line 4 will run v times to schedule every task. In *Select_Processor*, to find a longest path for the current task node requires $O(v)$ times in the worst case, and the outer *for* loops on line 2 will run n times. To compute *ECT* of a task, costs $O(v*L)$, where L is the max node degree in task graph G. The inner *for* loop will also run n times, and in each iterate, it will cost at most $O(v)$ to compute the *EPT* function. To update the available time of each processor, requires $O(n*v)$ on line 8. So, the total cost of the *Select_Processor* procedure is $O(v+n*v*L+n^2*v+ n*v) = O(n^2*v)$. Back to Algorithm PFAS, to update priorities of unscheduled tasks, costs $O(n*k)+O(v)$, where $O(n*k)$ is the cost to update the available performance of each processor and $O(v)$ is the cost to update ranks of tasks. So the total complexity of PFAS is $O(n^2v^2+nkv)$.

PFAS Algorithm
Input: A subgraph and a set of resources $r_1, \ldots, r_n$.
Output: task node to resource mapping
1. Compute $rank_u$ and $rand_d$ for each task using average resource performance;
2. Set the priority of each task as the sum of its $rank_u$ and $rand_d$;
3. Initial the ready queue *RQ* with the entry task;
4. While (there are unscheduled nodes){
5. Select the highest priority task t in *RQ*;
6. Call *Select_Processor*(t) to assign task t ;
7. Update priorities of all tasks;
8. }

□

Process *Select_Processor*(task t)
1. Find the longest path P from t to an exit node.
2. For all available processors p_i {
3. Compute $ECT(t, p_i)$;
4. For all processors p_j
5. Call $EPT(P, p_j, ECT(t,p_i)+w_{i,j}*e(i,j))$
6. }
7. Insert t to p_i that satisfies *Rule 2*;
8. For all available processors p_i, update available time of p_i $AVLT_i$ and feasible performance.

□

An example illustrating the PFAS algorithm is given below, which takes data in Table 1 (a) and (b) as resource performance and communication cost respectively, and schedules the task graph given by Fig. 2. It is also compared with other two scheduling methods: the HEFT algorithm and a performance fluctuation aware algorithm without the look-forward strategy (called NLF). In Table 2, ranks of tasks at each scheduling step are given, and in Table 3, the available average performance of each computational resource is given. In the first step, t_1 is selected as it is the highest rank ready task and the current dynamic critical path is $DCP_1 = \{t_1, t_4, t_5, t_7, t_8\}$. According to *Rule 2*, assigning t_1 to p_1 will give the minimum value

Table1: (a) A table showing the performance fluctuation of 3 computational resources in 12 time slots. (b) Communication cost of unit data transfer between resources given in (a).

	p_1	p_2	p_3
s_1	3	1	2
s_2	3	2	3
s_3	2	2	3
s_4	6	8	2
s_5	8	8	3
s_6	4	7	2
s_7	3	7	6
s_8	3	8	3
s_9	4	3	3
s_{10}	3	6	3
s_{11}	5	5	4
s_{12}	4	4	2
avg_c_i	4	5	3
avg_c	4		

(a)

	p_1	p_2	p_3
p_1	0	1	1.5
p_2	1	0	2
p_3	1.5	2	0
avg_w	1		

(b)

Table 2: Task Node ranks and the dynamic critical path of each scheduling step (in shading cells).

Steps	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
1	14.5	7.5	9	14.5	14.5	13	14.5	14.5
2	14.23	7.59	9.04	14.23	14.23	12.84	14.23	14.23
3	14.03	7.58	9.03	14.04	14.02	12.53	14.03	14.03
4	12.59	7.59	9.04	12.59	9.04	12.59	12.59	12.59
5	8.89	7.39	8.82	8.89	8.89	8.75	10.00	10.00
6	9.24	7.763	9.24	7.70	7.70	7.56	8.81	9.24
7	8.30	7.71	8.30	7.68	7.68	7.54	8.79	8.88
8	8.36	6.03	8.36	7.74	7.74	7.61	8.85	8,85

Table 3: Feasible average performance of computational resources in each scheduling step.

Steps	p_1	p_2	p_3	Avg.
Initial	4	5	3	4
1	129/31	85/14	87/28	4.4466
2	129/31	1240/203	87/28	4.4589
3	129/31	1024/169	87/28	4.4425
4	129/31	707/128	87/28	4.7629
5	129/31	602/113	87/28	4.1986
6	57/13	602/113	87/28	4.2731
7	100/27	602/113	87/28	4.0461

of Expression (10) (with p_x = p1 and $p_y = p_2$, the value is 6.7843, with p_x =1 p_y =1, the value is 7.6674, with p_x = 2, p_y = 2, the value is 7.0179). In the beginning of the second step, the ranks of tasks are updated as the finish time of t_1 is already known and available time slots on resources have changed. Now t_4 is in the ready queue with the highest priority, and $DCP_2 = \{t_4, t_5, t_7, t_8\}$. Again, *Rule 2* is called to find the best insertion which is processor 2 (with p_x =1 p_y =1, the value is 7.8947, with p_x = 2, p_y = 2, the value is 7.0660). Eventually, PFAS will give a schedule as the Gantt chart in Fig. 3 (a). Fig. 3 (b) and (c) give the results of the two compared methods. It is obvious that PFAS gives the best scheduled in term of makespan.

5. Experiments

To evaluate the effectiveness of PFAS in the Grid circumstances, comparative experiments are done to simulate its performance. Three different scheduling algorithms are tested in the experiments: 1) PFAS, 2) PFAS without look-ahead along the dynamic critical path (NLF), and 3) HEFT algorithm. The performance metric we used for the comparison is the Scheduled Length Ratio (SLR), which is the ratio of real makespan to the theoretical lower bound of any possible scheduling, which equals the execution time the longest path measured in

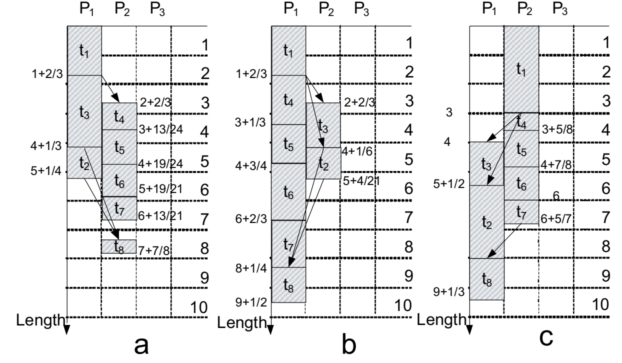

Fig. 3: Gantt charts of the different schedule approaches for the example: (a) PFAS; (b) PFAS without look-ahead (NLF); (c) HEFT.

computation cost on the fastest resources without any communication delays.

5.1 Experimental Settings

In the experiments, three resource clusters are used. Each cluster consists of 10 processing nodes connected by a LAN. The resource clusters are connected by a WAN. The topology and initial parameters such as processing capacity, communication cost, and load of each processor are generated using a toolkit named GridG1.0 [10].

In terms of input task graphs, a task graph generator called Task Graph For Free (TGFF) [11] is used to generate task graphs submitted to the Grid. TGFF has the ability of generating a variety of task graphs according to different configuration parameters, such as average number of task nodes of each graph, average outgoing and incoming degrees for each node in a graph, and computational and communication cost for each type of task nodes and edges.

To test the adaptive ability of our scheduling approach to different task graphs and resource settings, the following parameters are considered in the experiment:

- The average number task nodes in a graph *v*;
- The ratio of the average degree of a task node to the total number of tasks in a graph (Edge density in a graph);
- The computation-to-communication ratio (*CCR)* of a task graph. *CCR* is the average ratio of computation cost to communication cost. A high *CCR* value means a task graph is computation-intensive.
- Resource performance fluctuation factor which decides the percentage that the performance of a computational resource can increase or drop in different time slots.
- The communication heterogeneity factor which decides how different communication costs between different computational resources are.

5.2 Simulation Results

With respect to the number of nodes in a task graph, 5 different average values are applied: 20, 40, 60, 80 and 100. For each of these values, 25 graphs are generated. Fig. 4 (a) illustrates the average performance of the 3 scheduling algorithms. First, it can be observed that, as the number of task nodes increases, the performance of all of these three algorithms decreases. The explanation for the performance drop is that: the increasing of task nodes number will result in more accumulate error in task node ranking. Second, PFAS achieves the best performance among the three algorithms. NLF which only considers the performance fluctuation outperforms HEFT by a small margin. This implies that the benefit brought by only updating the task node ranks dynamically is limited.

The edge density is an important character of a graph, which decides the communication volume among tasks. To describe the edge density, the ratio of the average degree of each task node to the total number of nodes in a graph is used in our experiments. Five different settings are tested: 0.05, 0.1, 0.2, 0.3 and 0.4. For each setting, 25 different graphs are generated as well. As Fig. 4 (b) indicates, as the degree of task nodes increases, the SLR of PAFS firstly drops and then keeps steady, and it's the overall best. The SLR of NLF and HEFT firstly increases and then drops. Increasing the degree of tasks implies increasing of the total communication volumes, so the makespan is extended due to more communication delay. The interesting point is after the ratio is greater than 0.3, SLR of all of the three algorithms drops again. The explanation to this phenomenon is that, as the total number of task nodes is fixed, increasing the average degree of nodes has the effect of reducing the length of the critical path and increasing the breadth when a task graph is generated by TGFF. So, as the degree increases, the possibility of high parallelism also increases, which might shadow the increase in communication volume. This also explains why the performance PAFS is worse than HEFT at the beginning: when the critical path is longer, there are more errors in the look-head procedure which relies on the estimate to the finish time of the critical path.

The other parameter contributing to characteristics of a task graph is the CCR. In the experiment, the ratio increases from 0.5 to 10. As Fig. 4 (c) indicates, as the ratio increases, the SLR of PFAS and NLS slightly drops and then increase, and the one of PFAS is the lowest. The drop of SLR at the beginning is brought by the decreasing communication to computation cost ratio. But as computation cost of a task node increases, its execution time on different resources at different time becomes more different, which implies that the estimate to execution time departs from the real situation further.

To test the adaptiveness of the three scheduling

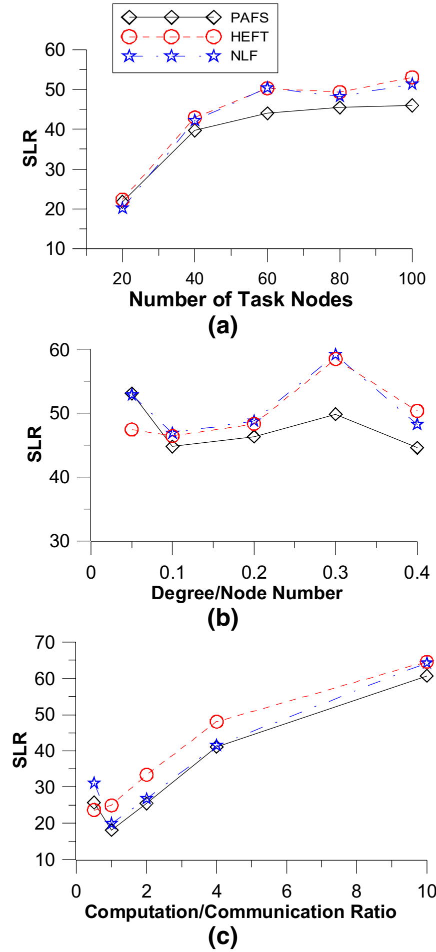

Fig. 4: Experiment results of different parameter settings. (a) Different number of tasks in a Grid Workflow. (b) Different average node degree in a task graph. (c) Different computation to communication ratio in a task graph.

methods to computational power fluctuation, five different values are assigned to the performance fluctuation factor: 20%, 40%, 50%, 60% and 80%, each denoting the maximum allowed percentage of full computation power drop in different time slots. As Fig. 5(a) shows, as resource performance becomes more fluctuating, the SLR of all methods increases which is brought by the more difficulty to get accurate estimate. PFAS, followed by NLF, is the best among the tested algorithm.

The other resource related parameter involved in the simulation is the communication cost heterogeneity ratio. In the experiment, the ratio is assigned 5 different values also: 0.2, 0.4, 0.6, 0.8 and 1.0, which gives the maximum percentage of the communication cost of a connection between two resources can different from the average cost

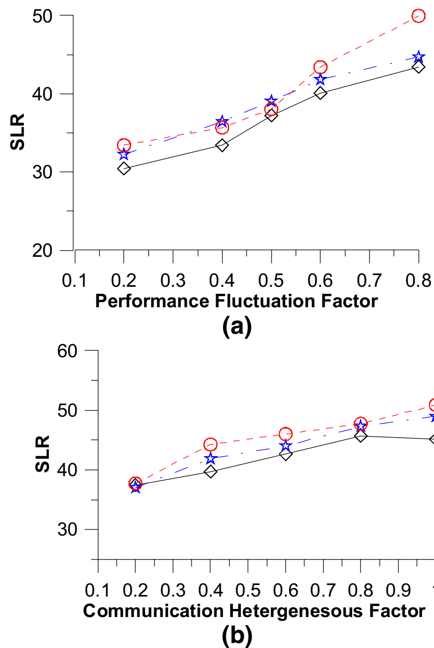

Fig. 5: (a) Different performance fluctuation factors. (b) Different communication cost factors.

value. As Fig. 5(b) indicates, the SLR of the three methods increases as the network connection becomes more heterogeneous which brings more errors to task node ranks. The SLR of PFAS is still the lowest, followed by NLF and HEFT, which means PFAS is more adaptive to the network heterogeneity than the other two methods.

6. Conclusions

In this paper we propose a resource performance fluctuation aware workflow scheduling algorithm PFAS for the Grid. Instead of using a static task ranking approach which is usually conducted once at the beginning of a DAG scheduling algorithm, PFAS updates task ranks and constructs the critical path dynamically in the scheduling procedure according to the change in performance of available resources. PFAS also adopts a look-ahead approach to assign a critical task. This allows it to overcome myopic decisions made by the earliest complete time criterion which is used by many other scheduling algorithms. Experiments show that the scheduling performance, measured in makespan, benefits from both techniques. Simulation results also show that PFAS is adaptive to different task graphs and resource topology settings. Its overall performance is much better than that of the HEFT algorithm, which is a powerful DAG scheduling algorithm designed for heterogeneous computational environments. The current implementation of PFAS does not consider the possibility of wrong performance prediction, which is likely in the real situations. This is the problem on which we are currently working. The simulations also show that estimating task ranks by average resource performance leads to an accumulation of estimate errors when the critical path is long or resources are more heterogeneous, so better and more complex ways might be introduced in the future for improvement. The algorithm is also going to be tested by realistic workflows in the Grid.

References

[1] E. Deelman, J. Blythe, et al. Pegasus: Mapping Scientific Workflows onto the Grid. In *the Proc. of Grid Computing: Second European AcrossGrids Conference (AxGrids 2004)*, pages:11- 26, January 2004.

[2] J. Cao, S. A. Jarvis, et al.. GridFlow: Workflow Management for Grid Computing. In *Proc. of the 3rd CCGrid*, pages:198-205, Tokyo, Japan, May 2003.

[3] M. Wieczorek, R. Prodan and T. Fahringer. Scheduling of Scientific Workflows in the ASKALON Grid Environment. In *ACM SIGMOD Record, Vol.34, No.3*, pages: 56-62, September 2005.

[4] L. Yang, J. M. Schopf and I. Foster. Conservative Scheduling: Using Predicted Variance to Improve Scheduling Decisions in Dynamic Environments. In *Proc. of the 2003 Supercomputing*, pages: 31-- 46, November 2003.

[5] K. Aggarwal and R. D. Kent. An Adaptive Generalized Scheduler for Grid Applications. In *Proc. of the 19th Annual International Symposium on High Performance Computing Systems and Applications (HPCS)*, pages: 15-18, May 2005.

[6] G. Mateescu**.** Quality of Service on the Grid via Metascheduling with Resource Co-Scheduling and Co-Reservation. In *International Journal of High Performance Computing Applications*, Vol. 17, No. 3, pages: 209-218, 2003.

[7] Y.K. Kwok and I. Ahmad. Dynamic Critical-Path Scheduling: an Effective Technique for Allocating Task Graphs to Multiprocessors. In *IEEE Trans. on Parallel and Distributed Systems*, Vol. 7, No. 5, pages: 506-521, May, 1996.

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

[9] H. Zhao and R. Sakellariou. An Experimental Investigation into the Rank Function of the Heterogeneous Earliest Finish Time Scheduling Algorithm. In *Proc. of Euro-Par 2003*, Springer-Verlag, LNCS 2790, pages: 189-194, Klagenfurt, Austria, August 2003.

[10] D. Lu and P. Dinda. Synthesizing Realistic Computational Grids. In *Proc. of ACM/IEEE Super-computing 2003*, Phoenix, AZ, USA, 2003.

[11] R.P. Dick, D.L. Rhodes and W. Wolf, TGFF Task Graphs for Free, *Proc. of the 6th. International Workshop on*

Hardware/Software Co-design, 1998.

[12] F. Dong and S. G. Akl. Grid Application Scheduling Algorithms: State of the Art and Open Problems. *Technical Report No. 2006-504, School of Computing, Queen's University,* Canada, Jan 2006.

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

[14] J. Liou and M. A. Palis. A Comparison of General Approaches to Multiprocessor Scheduling. In *Proc. of the 11th International Symposium on Parallel Processing*, pages:152-156, April 1997.

[15] T. Yang and A. Gerasoulis. DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors. In *IEEE Trans. on Parallel and Distributed Systems*, vol. 5, no. 9, pages: 951--967, 1994.

[16] S. Darbha and D.P. Agrawal. Optimal Scheduling Algorithm for Distributed Memory Machines. In *IEEE Trans. on Parallel and Distributed Systems*, vol. 9, no. 1, pages: 87-95, January 1998.

[17] R. Bajaj and D. P. Agrawal, Improving Scheduling of Tasks in A Heterogeneous Environment. In *IEEE Trans. on Parallel and Distributed Systems*, Vol.15, no. 2, pages: 107 – 118, February 2004.

[18] T. Ma and R. Buyya. Critical-Path and Priority based Algorithms for Scheduling Workflows with Parameter Sweep Tasks on Global Grids. In *Proc. of the 17th International Symposium on Computer Architecture and High Performance Computing*, pages: 251- 258, October 2005.

[19] L. He, S. A. Jarvis, D. P. Spooner, D. Bacigalupo, G. Tan, G. R. Nudd. Mapping DAG-based Applications to Multiclusters with Background Workload. In *Proc. of the 5th IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05)*, pages: 855-862, May 2005.

Biographies

Fangpeng Dong received his B.Sc from the Department of Computer Science and Technology, Peking University, Beijing, China in 2000 and M.E. from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China in 2003. He is now a Ph.D. student in the School of Computing, Queen's University at Kingston, Ontario, Canada. His major research interests include Grid computing and other parallel and distributed systems. He is also an IEEE student member.

Selim G. Akl received his Ph.D. degree from McGill University in Montreal in 1978. He is currently a professor of Computing at Queen's University, Kingston, Ontario, Canada. His research interests are in parallel computation. He is author of *Parallel Sorting Algorithms* (Academic Press, 1985), *The Design and Analysis of Parallel Algorithms* (Prentice Hall, 1989), and *Parallel Computation: Models and Methods* (Prentice Hall, 1997), and a co-author of *Parallel Computational Geometry* (Prentice Hall, 1992). Dr. Akl is editor in chief of *Parallel Processing Letters* and presently serves on the editorial boards of *Computational Geometry*, the *International Journal of Parallel, Emergent, and Distributed Systems*, and the *International Journal of High Performance Computing and Networking*.