J Supercomput (2010) 51: 201–223
DOI 10.1007/s11227-009-0284-7

DAGMap: efficient and dependable scheduling of DAG workflow job in Grid

Haijun Cao · Hai Jin · Xiaoxin Wu · Song Wu · Xuanhua Shi

Published online: 6 May 2009

Abstract DAG has been extensively used in Grid workflow modeling. Since Grid resources tend to be heterogeneous and dynamic, efficient and dependable workflow job scheduling becomes essential. It poses great challenges to achieve minimum job accomplishing time and high resource utilization efficiency, while providing fault tolerance. Based on list scheduling and group scheduling, in this paper, we propose a novel scheduling heuristic called DAGMap. DAGMap consists of two phases, namely *Static Mapping* and *Dependable Execution*. Four salient features of DAGMap are: (1) Task grouping is based on dependency relationships and task upward priority; (2) Critical tasks are scheduled first; (3) Min-Min and Max-Min selective scheduling are used for independent tasks; and (4) Checkpoint server with cooperative checkpointing is designed for dependable execution. The experimental results show that DAGMap can achieve better performance than other previous algorithms in terms of speedup, efficiency, and dependability.

Keywords DAG Grid workflow · Critical task · Adaptive scheduling · Cooperative checkpointing

1 Introduction

Grid computing is considered as a cornerstone of next generation distributed computing that coordinates large-scale resource sharing and problem solving in dynamic,

H. Cao · H. Jin (✉) · S. Wu · X. Shi
Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China
e-mail: hjin@mail.hust.edu.cn

X. Wu
Communication Technology Lab, Intel China Research Center, Beijing, 100080, China

multi-institutional virtual organizations. Through Internet, Grid enables people to co-operate with each other and share all resources across corporate, institutional, and geographic boundaries without sacrificing local autonomy. Grid workflow is defined as the orchestration of a set of atomic tasks processed at distributed resources in a well-defined order to accomplish a large and sophisticated goal. Currently, *Directed Acyclic Graph* (DAG) has been extensively used in scientific computational workflow modeling, especially large-scale computing-intensive or data-intensive Grid applications [1] such as high-energy physics, geophysics, astronomy, medical image processing, and bioinformatics. Based on DAG, two well-known Grid workflow management systems, GridAnt [2] and DAGMan [3], have been implemented and applied in the Globus [4] and Condor [5] project, respectively.

DAG workflow job scheduling in a Grid environment determines how to map all atomic tasks to a bounded number of distributed computing resources [6]. The problem can be described as that N tasks are scheduled to M computing resources (hosts) subject to the conditions which are: (1) the execution precedence constraint exists between two dependent tasks; (2) only one task can be executed on a host at each time; and (3) task executions are nonpreemptive. Such a scheduling problem has been shown to be NP-complete [7]. Since Grid resources tend to be heterogeneous and dynamic, efficient and dependable workflow job scheduling becomes essential. It poses great challenges to achieve minimum job accomplishing time and high Grid resources utilization efficiency, while providing dependable execution.

In this paper, based on list scheduling and group scheduling, we propose a novel scheduling heuristic for DAG workflow job, called DAGMap. It consists of two phases, namely *static mapping* and *dependable execution*. The static mapping phase includes two steps: *task grouping* and *independent tasks scheduling*. At the task grouping step, firstly, the upward priorities, downward priority, and overall priority of each task are determined. Accordingly, the collection of critical tasks can be obtained. Then tasks are grouped by the upward priority and dependency relationship, while tasks in the same group are kept independent. At the independent tasks scheduling step, independent tasks are scheduled group by group in an ascending group order.

In summary, the salient features and contributions of our heuristic are as follows:

- **Task grouping policy** Compared with simple grouping policies merely based on the task dependency relationship, our grouping policy also takes the task upward priority into account.
- **Critical tasks scheduled first** Considering that critical tasks are the main factor that affects the overall job finish time, in each group, DAGMap schedules a critical task to a computing host with a minimum earliest completion time first.
- **Min-Min and Max-Min selective scheduling for independent tasks** In each group, except for critical tasks, independent tasks are scheduled by Min-Min or Max-Min selectively according to the deviation of average estimated task execution time.
- **Checkpoint server with cooperative checkpointing is designed for dependable execution** According to the static mapping results, tasks then are executed by dynamic Grid resources. In order to provide fault tolerance, checkpoint server with cooperative checkpointing is proposed.

The remainder of this paper is organized as follows. Section 2 gives an overview of related works. Section 3 presents the formal definitions and preliminaries of the job scheduling problem. In Sect. 4, the algorithms of DAGMap are detailed. Experiments and performance evaluation are conducted in Sect. 5. Finally, we conclude this paper and give some future works in Sect. 6.

2 Related works

DAG workflow job scheduling problem has been extensively studied and a number of scheduling heuristics were proposed. These heuristics can be classified into several categories [8, 16], which are list scheduling algorithms [9], group scheduling algorithms [10], and clustering algorithms.

List scheduling is one of most commonly used scheduling algorithms. In list scheduling, a weight is assigned to each task and edge, based on which ordered task list is constructed by assigning priority for each task. Then tasks are selected in the order of their priorities, and each selected task is scheduled to a computing host that can minimize a predefined cost function. As two typical list scheduling heuristics, HEFT (*Heterogeneous Earliest Finish Time*) and CPOP (*Critical Path on a Processor*) are studied in [9]. The upward rank and downward rank of each task are computed at the beginning. The HEFT algorithm always selects the task with the highest upward rank at each step. Then the selected task is assigned to a host that can minimize its earliest finish time. In contrast, the CPOP algorithm always selects the task with the highest total rank (upward rank + downward rank) value from ready tasks queue. In order to minimize the total execution time, CPOP schedules all critical tasks onto a single host with the best performance. During execution, if a selected task is noncritical, it will be mapped to a host which could minimize its earliest finish time, as in HEFT. Both HEFT and CPOP have low complexity, i.e., lower algorithm execution time. However, the study in [11] observed that the performances of these two algorithms are affected dramatically by how to assign weights to the nodes and edges. In some extreme cases, different weight assignment approaches can lead up to 47.2% of performance difference.

In another popular scheduling heuristic group scheduling, tasks are sorted into groups, under the constraint that tasks in the same group should be independent. Tasks then are scheduled group by group. The studies in [12] proposed a hybrid remapping heuristic. Tasks in a DAG are partitioned into levels so that there is no dependency among tasks at the same level. Then tasks are mapped to computing hosts with task/host pairs using a static algorithm (e.g., baseline). The merit of this hybrid heuristic is to revise the static mapping result during job execution by two runtime factors, the availability of computing hosts and the completion time of tasks in previous levels. However, the task partition merely considers the task dependency relationships. It does not take task priority into account, which may result in that some tasks with lower priority are sorted into improper groups (levels). We will compare this level based task grouping policy with ours in Sect. 4.4.

Clustering algorithms are proposed for the case of an unbounded number of computing resources, so they are not suitable for a Grid environment.

Generally, all above mentioned heuristics are static algorithms, that is, the schedule decisions are made at the static mapping phase, which is prior to the workflow job execution. They do not take runtime fault tolerance and failure recovery into consideration.

3 System model and preliminaries

To illustrate the job scheduling problem clearly, in this section, we present the formal definitions for DAG workflow job and computing resource, and introduce several scheduling factors considered in DAGMap.

3.1 DAG workflow job

A DAG Grid workflow job can be represented by a directed acyclic graph $G = (T, E)$, where $T = \{t_1, t_2, \dots, t_N\}$ is the collection of tasks (N is the total number of tasks), and E is the collection of edges indicating the dependency and precedence constraint between tasks.

In a given task graph, a task without any precedents is called an entry task, and a task without any successors is an exit task. If there is more than one entry/exit task, a zero-cost task can be added, and these entry/exit tasks can be connected to it with zero-cost edges. This can ensure that there are only one single-entry task (denoted as t_{entry}) and one single-exit task (denoted as t_{exit}) in a DAG workflow job.

$D[N][N]$ is a $N \times N$ matrix, in which $D[i][j]$ is denoted as the amount of data required to be transferred from Task t_i to Task t_j.

3.2 Grid computing resources

For Grid resources, $H = \{h_1, h_2, \dots, h_M\}$ is defined as the collection of computing hosts (M hosts in total).

For an arbitrary host h_i, $R(h_i)$ is the ready time, that is, how long Host h_i will finish the current task so that it can be available for a new task.

$B[M][M]$ is a $M \times M$ matrix, in which $B[i][j]$ is the bandwidth between two hosts h_i and h_j. Since there is no need for transferring data within the same host, $B[i][i] \to \infty$.

$L[M][M]$ is a $M \times M$ matrix, in which $L[i][j]$ is the network latency from Host h_i to Host h_j, including the communication setup cost and the propagation time. Specially, $L[i][i] = 0$.

3.3 Scheduling factors

(1) ***Transmission time***: Suppose that Task t_i is the direct precedent of Task t_j. If t_i is being executed on Host h_m, and t_j will be executed on Host h_n, then the time required for transferring the output of t_i from h_m to h_n is called the *transmission time*, denoted as $IO(t_{i(m)}, t_{j(n)})$. As shown in (1),

$$IO(t_{i(m)}, t_{j(n)}) = L[m][n] + \frac{D[i][j]}{B[m][n]}. \tag{1}$$

As mentioned above, $L[m][n]$ is the network latency from h_m to h_n, $D[i][j]$ is the amount of data required to be transferred from t_i to t_j, and $B[m][n]$ is the bandwidth between h_m and h_n. According to (1), if $m = n, IO(t_{i(m)}, t_{j(m)}) = 0$.

Taking no accounts of the specific hosts on which tasks are executed, the *average transmission time*, $\overline{IO(t_i, t_j)}$, is the average time for transferring the output of t_i to t_j. As shown in (2),

$$\overline{IO(t_i, t_j)} = \overline{L} + \frac{D[i][j]}{\overline{B}}. \tag{2}$$

Here, $\overline{L}$ is the average network latency, and $\overline{B}$ is the average bandwidth among hosts. (2) ***Estimated execution time***: The *estimated execution time*, $ET(t_i, h_n)$, is defined as the estimated time when Task t_i is executed on Host h_n. A number of researches have been done on the estimation of task/host execution time, which is beyond the scope of this paper. Based on this, the *average estimated execution time* of t_i, denoted as $\overline{ET(t_i)}$, can be calculated as follows:

$$\overline{ET(t_i)} = \sum_{n=1}^{M} ET(t_i, h_n)/M. \tag{3}$$

(3) ***Expected start time** and **expected finish time***: For a given pair of task/host t_i and h_n, $EST(t_i, h_n)$ is the expected start time. As shown in (4),

$$EST(t_i, h_n) = \max\Big(R(h_n), \max_{t_k \in Pre(t_i)} (AFT(t_k) + IO(t_k, t_i))\Big). \tag{4}$$

Here, $R(h_n)$ is the ready time of Host h_n. $Pre(t_i)$ is the direct precedent task collection of t_i. $AFT(t_k)$ is the *actual finish time* for Task t_k, which is determined by the host on which t_k is executed. Suppose that t_k is executed on h_n, then $AFT(t_k) = EFT(t_k, h_n)$. For the entry task, t_{entry}, $EST(t_{\text{entry}}, h_n) = 0$.

$EFT(t_i, h_n)$ is the *expected finish time*. As shown in (5),

$$EFT(t_i, h_n) = EST(t_i, h_n) + ET(t_i, h_n). \tag{5}$$

(4) ***Makespan***: For a given DAG workflow job, *Makespan* is defined as the *overall finish time* after static mapping, which is equal to the *actual finish time* of the exit task t_{exit}. As shown in (6),

$$Makespan = AFT(t_{\text{exit}}). \tag{6}$$

(5) ***Task priority***: The upward priority of Task t_i, denoted as $P_{\text{up}}(t_i)$, is defined as the longest distance form t_i to the exit task t_{exit}, including the average estimated execution time $\overline{ET(t_i)}$. Starting with t_{exit}, the upward priority of each task can be computed recursively by traversing the task graph upward. As shown in (7),

$$P_{\text{up}}(t_i) = \overline{ET(t_i)} + \max_{t_j \in Suc(t_i)} (\overline{IO(t_i, t_j)} + P_{\text{up}}(t_j)). \tag{7}$$

Here, $Suc(t_i)$ is the direct successor task collection of t_i.

In contrast, the *downward priority* of Task t_i, denoted as $P_{\text{down}}(t_i)$, is defined as the longest distance form the entry task t_{entry} to t_i, excluding the average estimated execution time $\overline{ET(t_i)}$. Accordingly, for the entry task, $P_{\text{down}}(t_{\text{entry}}) = 0$. Starting with t_{entry}, the downward priority of each task can be computed recursively by traversing the task graph downward. As shown in (8),

$$P_{\text{down}}(t_i) = \max_{t_j \in Pre(t_i)} (P_{\text{down}}(t_j) + \overline{ET(t_j)} + \overline{IO(t_j, t_i)}). \tag{8}$$

Here, $Pre(t_i)$ is the direct precedent task collection of t_i.

The *total priority* of Task t_i, denoted as $P_{\text{total}}(t_i)$, is defined as that passing through t_i, the longest distance from the entry task t_{entry} to the exit task t_{exit}. As shown in (9),

$$P_{\text{total}}(t_i) = P_{\text{up}}(t_i) + P_{\text{down}}(t_i). \tag{9}$$

(6) ***Critical task***: In a given DAG task graph, the path with the longest distance from the entry task t_{entry} to the exit task t_{exit} is called the critical path. Note that in general there may be more than one critical path in a DAG graph.

A task on the critical path is called a *critical task*. t_{entry} is a critical task. Thus, according to (9), the total priority of Task t_{entry}, $P_{\text{total}}(t_{\text{entry}})$, is equal to the length of the critical path. Therefore, for an arbitrary task t_i, if $P_{\text{total}}(t_i) = P_{\text{total}}(t_{\text{entry}})$, then t_i is a critical task.

(7) ***Task heterogeneity***: As proposed in [13], we use task Heterogeneity Factor (HF) to indicate the execution time deviation among independent tasks. For a collection of independent tasks, $T = \{t_1, t_2, \ldots, t_m\}$, HF is defined as the standard deviation of task average estimated execution time. As shown in (10),

$$HF = \sqrt{D(X)} = \sqrt{E(X - E(X))^2}$$
$$\begin{cases} x_1 = \overline{ET(t_1)} \\ x_2 = \overline{ET(t_2)} \\ \quad \vdots \\ x_m = \overline{ET(t_m)} \end{cases}. \tag{10}$$

Here, $D(X)$ is the mean square deviation of $(x_1, x_2, \ldots, x_m)$, and $E(X)$ is the mathematical expectation of $(x_1, x_2, \ldots, x_m)$.

In addition, we denote $HF_{\text{threshold}}$ as the threshold of task deviation. For a given collection of independent tasks, if $HF < HF_{\text{threshold}}$, it means the lengths of most tasks are within a small range. Otherwise, it means the lengths of tasks deviate from each other greatly.

(8) ***Computational consistency***: Besides heterogeneity, computational consistency of Grid resources is also ubiquitous. Suppose Task $t_i \in T$, two hosts $h_m, h_n \in H$, if the estimated execution times of t_i on these two hosts satisfy $ET(t_i, h_m) < ET(t_i, h_n)$, then when executing t_i, h_m is faster than h_n.

For an arbitrary $t_i \in T$, if $ET(t_i, h_m) < ET(t_i, h_n)$ is always satisfied, that is, any task can be executed faster on Host h_n than on Host h_m, there exists the resource consistency between h_m and h_n.

For two arbitrary hosts, $h_m, h_n \in H$, if there always exists resource consistency, then the computing resources in the host collection are computationally consistent. Otherwise, they are computationally inconsistent.

4 DAGMap scheduling heuristic

DAGMap scheduling heuristic is designed to take advantages of both list scheduling and group scheduling. It consists of two phases, namely *static mapping* and *dependable execution*. The static mapping phase includes two steps: *tasks grouping* and *independent tasks scheduling*.

4.1 Tasks grouping

As shown in Algorithm 1, firstly, according to (7) and (8), the upward priority and downward priority of each task are computed recursively (lines 1–2). Then for an arbitrary task t_i, the total priority $P_{\text{total}}(t_i)$ is obtained. For Task t_i, if its total priority $P_{\text{total}}(t_i)$ is equal to $P_{\text{total}}(t_{\text{entry}})$, it is added to the critical task collection CT (lines 4–10). Accordingly, the collection of critical tasks can be obtained.

Then tasks are grouped by the upward priority and dependency relationships. Tasks in the same group are independent. The procedures of task grouping are as follows:

(1) Tasks are sorted in descending order by the upward priority P_{up} (line 11).
(2) The entry task t_{entry} is added to $G_k (k = 1)$(line 12).
(3) For a successive task t_i, if it is independent from all tasks which have already been added into group G_k, t_i is added to G_k. Otherwise, a new group G_{k+1} is

Algorithm 1 DAGMap Heuristic

1. compute P_{up} for each task by traversing graph upward, starting with t_{exit}; // Eq. (7)
2. compute P_{down} for each task by traversing graph downward, starting with t_{entry}; // Eq. (8)
3. compute *HF* for all tasks; //according to Eq. (10)
4. $CT = \{\}$; //create the critical task collection
5. $P_{\text{total}}(t_{\text{entry}}) \leftarrow P_{\text{up}}(t_{\text{entry}}) + P_{\text{down}}(t_{\text{entry}})$;
6. **for** (each $t_i \in T$)
7.   compute $P_{\text{total}}(t_i)$; //according to Eq. (9)
8.   **if** $(P_{\text{total}}(t_i) == P_{\text{total}}(t_{\text{entry}}))$
9.   **then** add t_i to CT;
10. **end for**
11. sort tasks in a descending order of P_{up};
12. $k = 1$; $G_k = \{\}$; add t_{entry} to G_k;
13. **for** (each t_i in a descending order of P_{up})
14.   **if** $((\exists t_j \in G_k) \&\& (t_i$ depends on $t_j))$
15.   **then** k++; $G_k = \{\}$; //create a new group
16.   add t_i to G_k; //add task to the group
17. **end for**
18. **for** (each G_i in an ascending order)
19.   schedule independent tasks in group G_i;
20. **end for**

created, and t_i is added to G_{k+1} (lines 13–17). The task grouping operation is made repeatedly until all tasks are grouped.

After grouping, tasks are scheduled group by group in an ascending group order (lines 18–20), as detailed in Sect. 4.2.

4.2 Adaptive independent tasks scheduling

Min-Min and Max-Min heuristics are two typical independent task scheduling algorithms with the expectation that tasks are assigned to the machines which can compute them the earliest and fastest; in most cases, Min-Min shows an outstanding performance [14]. However, the study in [13] has shown that Max-Min can outperform Min-Min when the lengths of tasks deviate greatly. For instance, if there is only one long task and many short tasks, Min-Min executes all short tasks first, and then the long task would be executed while several machines sit idle. In contrast, Max-Min executes the long task first. At the mean time, it executes short tasks concurrently with the long task. This can result in a better makespan and even a better resource utilization rate and load balancing than Min-Min.

Considering the critical task and task deviation, we proposed adaptive independent tasks scheduling heuristic, which is shown in Algorithm 2.

Algorithm 2 Independent Tasks Scheduling

1. **while** $(G_i \neq \varnothing)$
2.   **for** (each task $t_i \in G_i$)
3.   **for** (each host $h_j \in H$)
4.   compute $EFT(t_i, h_j)$; //Expected Finish Time
5.   **end for**
6.   **end for**
7. **if** $((\exists t_j \in G_i)\&\&(t_j \in CT))$ **then** //Critical Task
8.   find Host h_m with the smallest m on which t_j can achieve the earliest EFT;
9.   assign t_j to h_m;
10. $G_i \leftarrow G_i - \{t_j\}$;
11.   update the host ready time $R(h_m)$ for h_m;
12.   update AFT for t_j; //Actual Finish Time
13. **else** // adaptive scheduling depends on HF
14. $TH = \{\}$; //create a temporary collection for task/host
15.   **for** (each task $t_k \in G_i$)
16.   find Host h_k with the smallest k on which t_k can achieve the earliest EFT;
17.   add the pair (t_k, h_k) to TH;
18.   **end for**
19.   **if** $(HF < HF_{\text{threshold}})$ **then** //adopt Min-Min
20.   select the pair $(t_s, h_s) \in TH$ with the minimum earliest EFT;
21.   **else** //otherwise, adopt Max-Min
22.   select the pair $(t_s, h_s) \in TH$ with the maximum earliest EFT;
23.   **end if**
24.   assign t_s to h_s;
25. $G_i \leftarrow G_i - \{t_s\}$;
26.   update the host ready time $R(h_s)$ for h_s;
27.   update AFT for t_s;
28. **end if**
29. **end while**

In this heuristic, as a first step, if there exists a critical task t_j in group G_i, t_j will be scheduled to Host h_m where t_j can achieve earliest *EFT*. Then t_j is removed from G_i, and the host ready time $R(h_m)$ and the actual finish time $AFT(t_j)$ are updated (lines 7–12). Since the critical task is the main factor that determines the overall finish time for the workflow job, we should schedule the critical task as early as possible.

Secondly, except for critical tasks, other remaining independent tasks are scheduled by Min-Min or Max-Min selectively according to the deviation of average estimated execution time. For an arbitrary Task t_k, Host h_k with the smallest k is found, through which t_k can achieve the earliest *EFT*. The pair (t_k, h_k) then is added to the temporary task/host collection *TH* (lines 14–18). Next, we compare the task length deviation *HF* with a given threshold $HF_{\text{threshold}}$. If $HF < HF_{\text{threshold}}$, Min-Min is adopted to select the successive tasks (lines 19–20). Otherwise, Max-Min is adopted (lines 21–22).

This process is repeated until all tasks in G_i are scheduled.

It is worth mentioning that for different workflow application job, $HF_{\text{threshold}}$ might be significantly diverse. Our method to determine the value of $HF_{\text{threshold}}$ follows such steps: (1) According to (10), *HF* can be obtained; (2) We set $HF_{\text{threshold\_1}} = \mathit{floor}(HF)$ and $HF_{\text{threshold\_2}} = \mathit{ceil}(HF)$, then $HF_{\text{threshold\_1}}$ and $HF_{\text{threshold\_2}}$ are used separately in Algorithm 2, consequently we can $Makespan_1$ and $Makespan_2$ for this workflow job, respectively. (3) For the above two makespans, the smaller, the better. Thus, we chose its corresponding threshold value of *HF* as $HF_{\text{threshold}}$.

Suppose there are X independent tasks in G_i and there are Y computing hosts, the time complexity for computing the *expected finish time* (*EFT*) for tasks (lines 2–6) is $O(XY)$. Because at each time only one task is scheduled, the overall time complexity for Algorithm 2 is $O(X^2Y)$.

4.3 Dependable execution

After static mapping, tasks are queued in the line of its corresponding host. However, as mentioned above, all static mapping heuristics are carried out prior to the workflow job execution, and are on the basis of two assumptions: (1) the estimated execution time of each pair of task/host are precisely made, and each task can be finished within the predefined time interval; (2) computing hosts and interconnection network are reliable. However, during real job execution, performance slowdown of resource might result in that job will be uncompleted in the time quota. Moreover, because of the dynamic feature of Grid, resources departure and failure are inevitable. Therefore, static mapping heuristics are not complete solutions, and the fault tolerance mechanism should be provided for dependable execution.

Until now, checkpointing is still the best approach for providing reliable completion of jobs on inherently unreliable hardware. Generally, checkpointing involves periodically saving a sufficient amount of the state of a running job to stable storage, allowing for that job to be started from the last successful checkpoint. A number of practical checkpointing packages have been developed for the Linux/UNIX family of operating systems. These packages may be divided into two classes, namely user space checkpointing and kernel based packages. Those which operate in user space are highly portable and can typically be compiled and run on any modern UNIX,

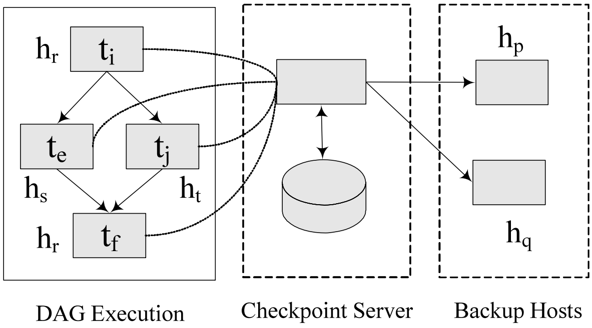


Fig. 1 Checkpoint server

such as Condor checkpoint [20] and Cocheck library [21]. In contrast, kernel based checkpointing packages such as Chpox [22] and Mosix checkpoint [23] for clusters and multiclusters can work as a Linux kernel module. Moreover, studies in [19] have shown that cooperative checkpointing provides greater performance and reliability than periodic checkpointing because it allows for irregular checkpoint intervals by giving the system an opportunity to skip requested checkpointing at runtime.

As shown in Fig. 1, by adopting cooperative checkpointing, we propose the checkpoint server with large capacity and high bandwidth storage, which is specifically designed to move checkpoint from local disk, across the high-speed interconnection network, and on to the stable storage. To insure that multiple checkpoints can take place simultaneously and frequently, if necessary, a scalable number of checkpoint servers can be placed to provide better performance.

The procedure of our checkpointing can be described as follows.

Firstly, checkpointing initiation. By using checkpointing API, the workflow programmer inserts checkpoint request in the atomic task code where the state is minimal, or where a checkpoint is efficient. Two groups of operation system processes are specified in the request: those that should be checkpointed; and those that should block while checkpointing. To save overhead, the local disk is specified to temporarily store the checkpoint file by default.

Secondly, checkpointing evaluation. After receiving a checkpoint request, with the cooperative checkpointing mechanism, the checkpoint server makes a decision either to grant or to deny the request based on failure prediction, which is through evaluating several system condition factors, such as network traffic, disk usage, job scheduling queue, and event prediction. Note that critical event based failure prediction on real system traces have seen accuracies up to 80% [17, 18].

Thirdly, checkpointing. If a checkpoint request is granted, the process of the original task is temporarily paused, and then the checkpointing process is taken to save the state of task on local disk. To reduce the checkpoint overhead, if necessary, the incremental checkpointing is adopted as an optimization technique.

Fourthly, restarting task and storing checkpoint image. After the checkpointing is complete, the restart function is invoked immediately to let the original task continue. Meanwhile, the checkpoint server receives the signal, and then transfers the checkpoint image with identifier meta-information from local disk to the stable storage.

In addition, in order to insure workflow job execution in case of any machine which is permanent failure, several backup hosts are connected to checkpoint server with high speed network. For instance, as shown in Fig. 1, suppose Task t_i is being executed on Host h_r, and its direct successor Task t_j will be executed on Host h_t. If h_r fails beyond a specific time interval, the scheduler will select a backup host, e.g., h_p, to take over the task and to restart it from the latest checkpoint. After completion, the output of t_i can be directly transferred from h_p to h_t. This is also suitable for the situation that a task can not be finished at the end of the time quota. In this case, for the host, because it is time to execute another new task, the current task must be forced to vacate the machine through checkpointing, and then it will migrate to a backup host to continue execution.

4.4 A static mapping example

In this section, in a consistent computational heterogeneous Grid environment, we choose a DAG job sample to comparatively illustrate the static mapping of DAG workflow job. As shown in Fig. 2(a), as a general representative of DAG workflow job, it is composed of 10 atomic tasks. The number next to each edge indicates the amount of data required to be transferred. Suppose there are three available computing hosts $H = \{h_1, h_2, h_3\}$. Figure 2(b) shows the *estimated execution time* (*ET*) for each task/host pair. Figure 2(c) shows the network property among these hosts. For simplicity, the bandwidth is fixed at 1 between two different hosts and at ∞ for within the same host. The network latency is fixed at 0.

According to Algorithm 1, for each task, we can compute the average estimated execution time $\overline{ET(t_i)}$, the upward priority P_{up}, the downward priority P_{down}, and the total priority P_{total}, as shown in Table 1(a). Accordingly, the collection of critical tasks and the grouping result can be obtained, as shown in Table 1(b) and (c), respectively.

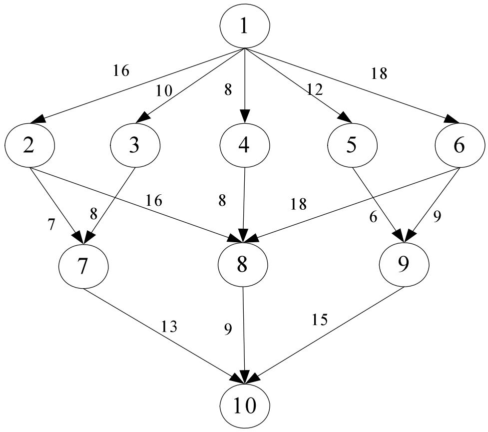
For comparison, we scheduled this DAG job using DAGMap, HEFT, CPOP, GS.Min-Min, and GS.Max-Min. Here, GS.Min-Min and GS.Max-Min are both based on grouping scheduling, and adopt Min-Min and Max-Min, respectively, to schedule the independent tasks in each group. From the scheduling results presented in Fig. 3, it is observed that for this workflow job, the makespan by DAGMap is 536, is less than by other four heuristics. As well as the speedup and resource efficiency by DAGMap are 1.80 and 0.741, respectively, which are also higher than by other related works.

As mentioned in the related works, the study in [12] adopted a task grouping policy based on level partition, which merely considers task dependency relationships. According to their policy, tasks in the sample workflow job are sorted into four groups: $\{t_1\}$, $\{t_2, t_3, t_4, t_5, t_6\}$, $\{t_7, t_8, t_9\}$, and $\{t_{10}\}$, as shown in Fig. 4.

To compare this level based task grouping policy with ours, according to the grouping result presented in Fig. 4, we schedule independent tasks at each level using Algorithm 2, Max-Min, and Min-Min, which are denoted as Level.Algorithm2, Level.Max-Min, and Level.Min-Min, respectively. The scheduling results are presented in Fig. 5.

Comparing the static mapping results in Fig. 3 and in Fig. 5, it is observed that for this workflow job, DAGMap performs better than Level.Algorithm2, GS.Max-Min is equal to Level.Max-Min, and GS.Min-Min is better than Level.Min-Min in terms of makespan, speedup, and resource efficiency. This is because our task grouping policy not only considers the dependency relationship between tasks, but also takes

Fig. 2 Example of DAG job with computing hosts



(a) Example of DAG job graph

Tasks	h_1	h_2	h_3
t_1	113	145	189
t_2	166	200	213
t_3	74	89	110
t_4	48	67	95
t_5	112	156	188
t_6	106	135	167
t_7	169	183	206
t_8	55	71	96
t_9	83	102	127
t_{10}	38	41	53

(b) Estimated execution time for task/host

Machines	Bandwidth	Latency
$h_1 - h_2$	1	0
$h_1 - h_3$	1	0
$h_2 - h_3$	1	0
$h_1 - h_1$	∞	0
$h_2 - h_2$	∞	0
$h_3 - h_3$	∞	0

(c) Network property

the task upward priority into account, which can prevent some tasks (e.g., t_4 and t_8) with lower priority from being sorted into improper groups.

5 Performance evaluation

In this section, we evaluate the proposed algorithms in two aspects: static mapping and dependable execution. The former is evaluated in terms of speedup, efficiency,

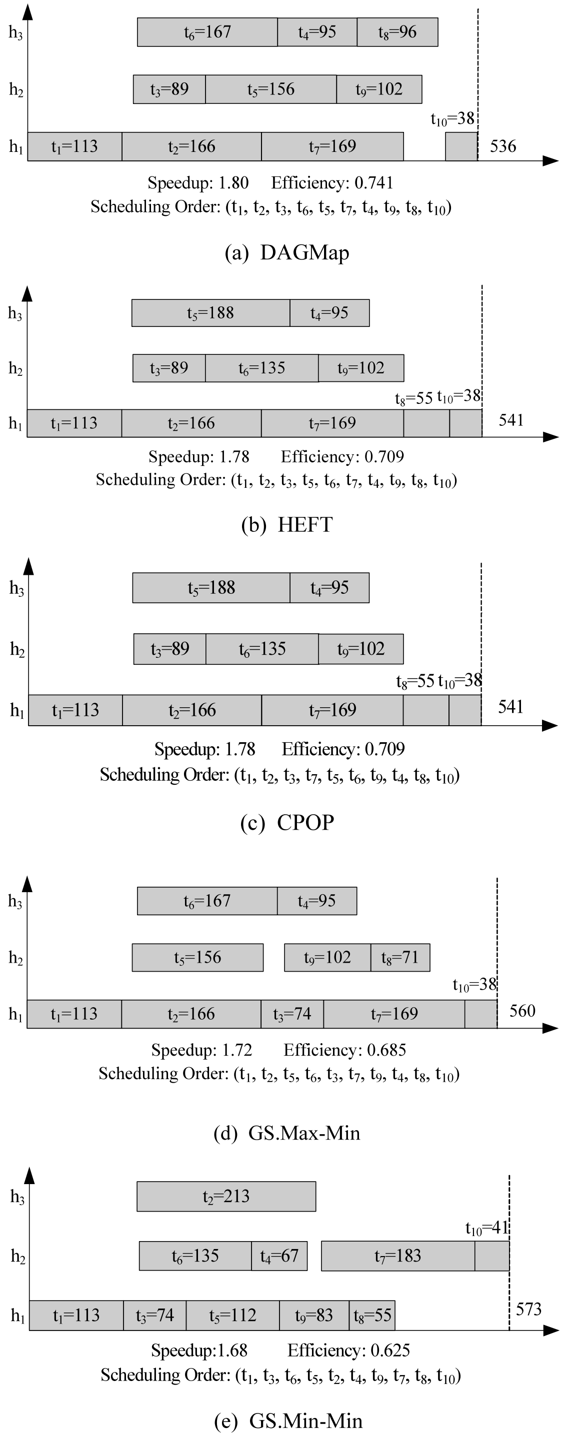


Fig. 3 Scheduling results

Table 1 Critical tasks and task groups

(a) Task priorities

Tasks	$\overline{ET}(t_i)$	P_{up}	P_{down}	P_{total}
t_1	149	608	0	608
t_2	193	443	165	608
t_3	91	342	159	501
t_4	70	205	157	362
t_5	152	321	161	482
t_6	136	308	167	475
t_7	186	243	365	608
t_8	74	127	374	501
t_9	104	163	319	482
t_{10}	44	44	564	608

(b) Critical tasks

CT	$\{t_1, t_2, t_7, t_{10}\}$

(c) Task groups

G_i	Tasks
G_1	$\{t_1\}$
G_2	$\{t_2, t_3, t_5, t_6\}$
G_3	$\{t_7, t_4, t_9\}$
G_4	$\{t_8\}$
G_5	$\{t_{10}\}$

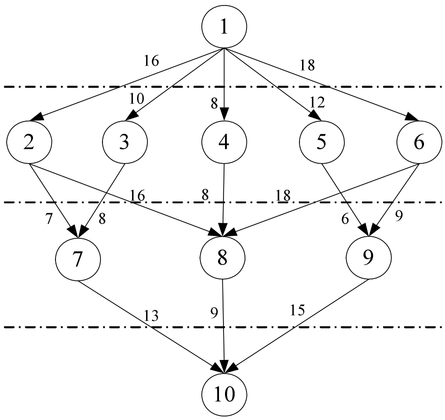
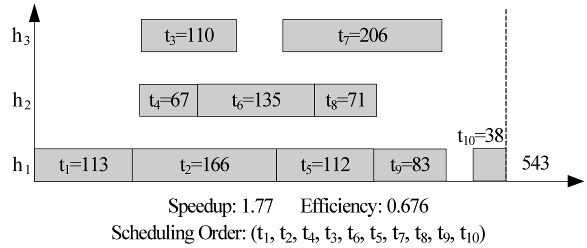


Fig. 4 Level based task grouping policy

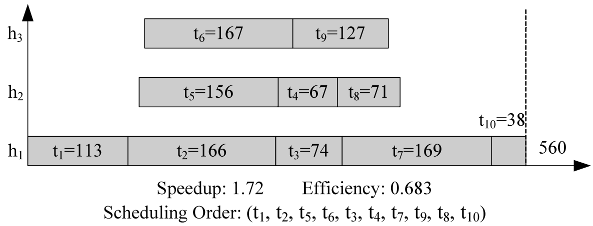
and algorithm running time, while the latter is evaluated in terms of success ratio and slowdown ratio. These metrics are defined as follows:

- *Speedup* For a given DAG workflow job, the speedup value is defined as the ratio between minimal sequential execution time and makespan, as shown in (11). The sequential execution time is the cumulative computation cost when mapping all the

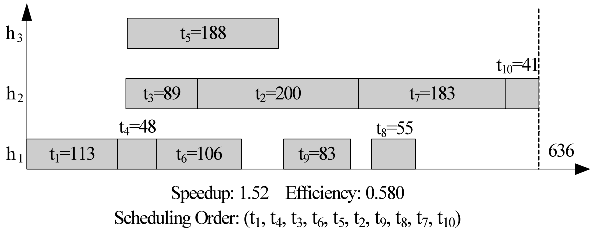
Fig. 5 Scheduling results with level based task grouping policy



(a) Level.Algorithm2



(b) Level.Max-Min



(c) Level.Min-Min

tasks sequentially to a single computing host.

$$Speedup = \frac{\min_{h_m \in H}\{\sum_{i=1}^{N} ET(t_i, h_m)\}}{Makespan}. \tag{11}$$

- *Efficiency* is defined as the utilization rate of all computing resources, as shown in (12). Here, $MET(t_i)$ is the *mapped execution time* of Task t_i.

$$Efficiency = \frac{\sum_{i=1}^{N} MET(t_i)}{Makespan * M}. \tag{12}$$

- *Algorithm Running Time* is the execution time of the static mapping algorithm itself, which indicates the algorithm complexity. Job execution time will not be counted.
- *Success Ratio* is the proportion of DAG jobs executed successfully to the total submitted workflow jobs during dependable execution.

- *Slowdown Ratio* is the exceeding proportion of workflow job execution time (JET_{ckpt}) versus the makespan, as shown in (13). Here, makespan is from static mapping without checkpoint, while JET_{ckpt} includes makespan, checkpointing overheads, and recomputation time from the latest checkpoint in case of failure.

$$Slowdown = \frac{JET_{\mathrm{ckpt}}}{Makespan} - 1. \tag{13}$$

5.1 Simulation environment

Based on GridSim and SimJava toolkit [15], we implement a simulator to establish the consistent and inconsistent heterogeneous Grid environments.

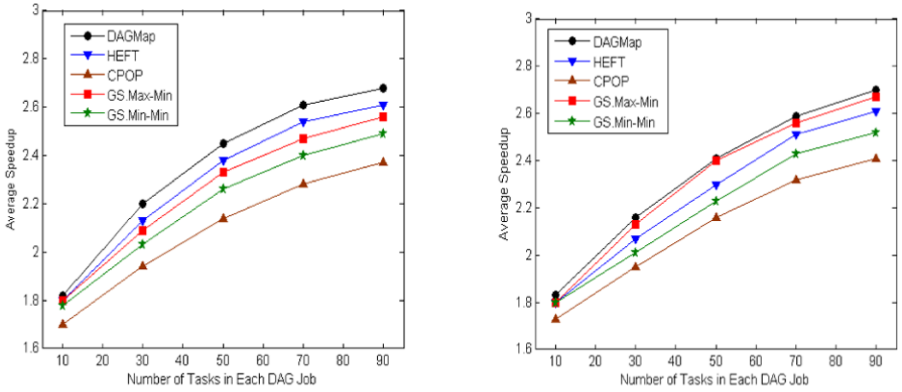
For static mapping, we conduct experiments to evaluate the performance of DAGMap with above mentioned heuristics comparatively. Inputs to the simulator include scheduling algorithm, type of DAG workflow job, tasks dependency relationships, number of hosts, estimated execution time for each task/host pair, and communication time between two dependent tasks. In our experiments, we use two common DAG jobs, i.e., Random Graphs and Laplace [11]. In each case, we randomly generate 10,000 DAG jobs, and each job consists of 10 to 100 tasks. The estimated execution time for a task/host pair is generated randomly following a uniform distribution over an interval [100, 500]. For any two dependent tasks, the transmission time is chosen randomly based on the communication-to-computation ratio (CCR) from the interval of [0.1, 0.2].

For dependable execution, in addition to above configurations in static mapping, other parameters for the simulator include MTTF (Mean Time To Failure), MTTR (Mean Time To Recovery), checkpointing overhead. In experiments, the time to failure (TF) and the time to recovery (TR) of computing hosts are exponentially distributed with mean of 400 (MTTF = 400) and 50 (MTTR = 50), respectively, namely TF $\sim$ E($1/400$) and TR $\sim$ E($1/50$). The checkpointing overhead (CO) of tasks follows a normal distribution with mean 40 and variance 16, i.e., CO $\sim$ N(40, 16).

5.2 Experimental results of static mapping

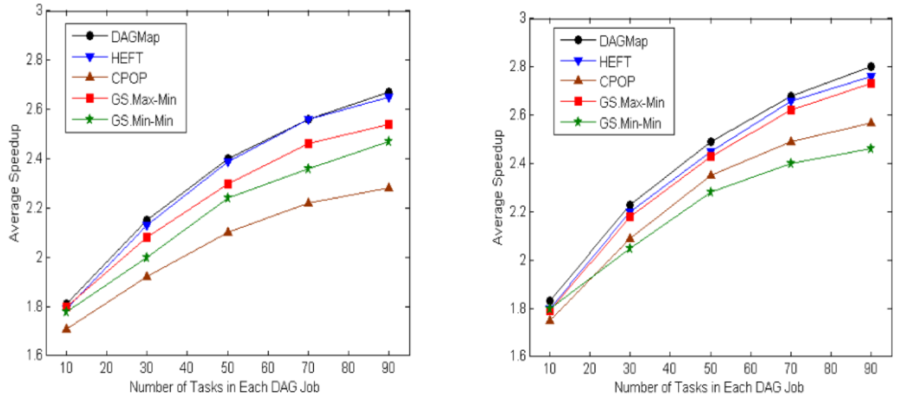
Figure 6 and Fig. 7 show the average speedup of Random graph and Laplace DAG jobs in consistent and inconsistent computing environments. Generally, DAGMap performs best, while HEFT and GS.Max-Min are better than CPOP and GS.Min-Min. It is noted that among all group-based scheduling algorithms, in consistent heterogeneous environments, DAGMap outperforms GS.Max-Min about 3% and outperforms GS.Min-Min about 13%. This is mainly because in each group, DAGMap schedules the critical tasks first. Moreover, the selective task chosen policy in DAGMap can best utilize Min-Min and Max-Min and avoid their shortcomings. It has also observed that for two list scheduling algorithms, HEFT performs better than CPOP. The reason is, as mentioned above, CPOP schedules all critical tasks to a single host with the best computing capacity. However, as the number of tasks in a workflow job increases, more and more tasks tend to be not critical.

Figure 8 and Fig. 9, respectively, show the heuristic efficiency for Random graph and Laplace DAG jobs in consistent and inconsistent computing environments. According to (12), the efficiency depends on three factors: makespan, number of hosts,



(a) Random Graph DAG Jobs (b) Laplace DAG Jobs

Fig. 6 Average speedup under consistent heterogeneity environment

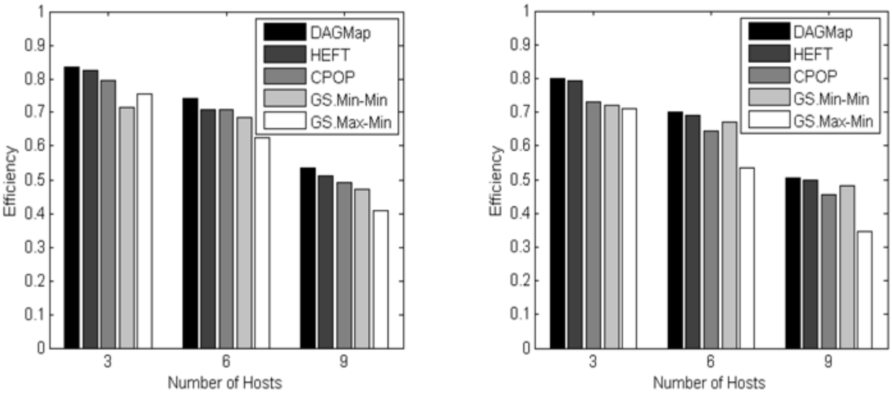


(a) Random Graph DAG Jobs (b) Laplace DAG Jobs

Fig. 7 Average speedup under inconsistent heterogeneity environment

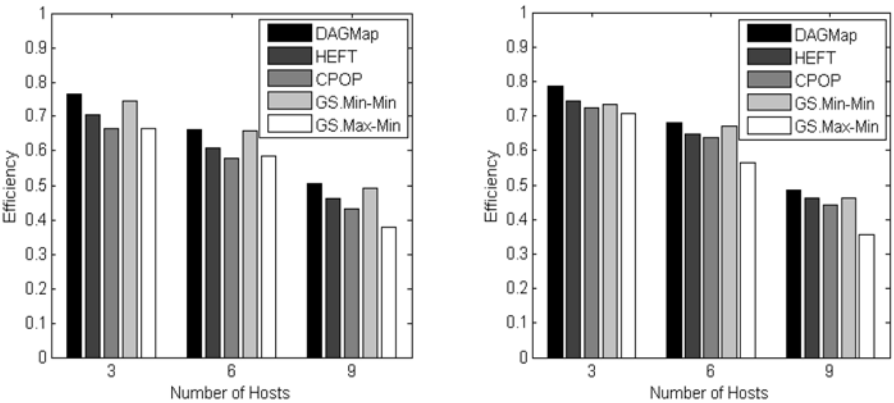
and actual execution time of each task. Because adopting DAGMap can obtain the least makespan for a workflow, DAGMap can achieve best resource utilization in each case. As the number of tasks increase, the depth of generated task graphs increase and the degree of parallelism becomes low. Hence, it is observed that except for the case that Random Graph jobs are executed on inconsistent heterogeneous hosts, HEFT can achieve a relative better performance than CPOP, GS.Min-Min, and GS.Max-Min, especially for Laplace workflow jobs.

In Fig. 10, we compare the average running time for static mapping of all algorithms when randomly generating DAG jobs. It has been observed that DAGMap has the longest execution time because of its complexity. This can be viewed as the algorithm tradeoff between the computing cost and the speedup/efficiency gain. However,



(a) Random Graph DAG Jobs (b) Laplace DAG Jobs

Fig. 8 Efficiency under consistent heterogeneity environment



(a) Random Graph DAG Jobs (b) Laplace DAG Jobs

Fig. 9 Efficiency under inconsistent heterogeneity environment

as all these heuristics are static scheduling algorithms, i.e., all tasks mapping and scheduling are made prior to the workflow job execution, and it is noted that the running time of static mapping remains at the millisecond level and cannot significantly influence the actual job execution time. DAGMap is still reasonable and acceptable.

5.3 Experimental results of dependable execution

For dependable execution, experiments are carried out when DAGMap adopts two checkpoint policies: periodic checkpointing and cooperative checkpointing with checkpoint server. The time interval of checkpoint request in periodic checkpointing is over a range of [100,400]. For cooperative checkpointing in real application, however, checkpoints requested by applications are irregularly, and checkpointing

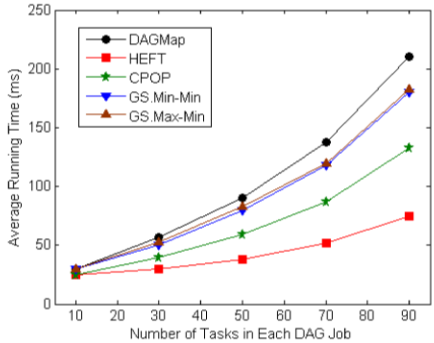


Fig. 10 Average running time

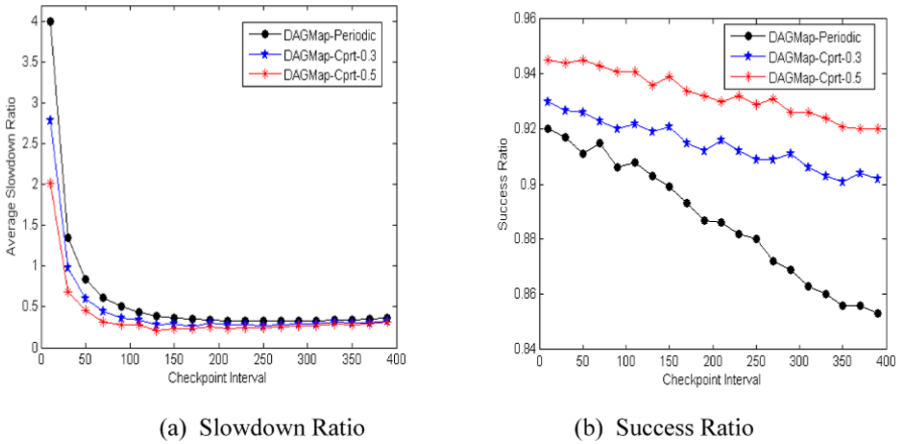


Fig. 11 Dependable execution with different checkpoint policies

will not be taken for each request. For simplicity, we make use of the same checkpoint request generator as in periodic checkpointing, and use a failure predictor with certain accuracies, i.e., 0.3 and 0.5, to respectively grant a checkpoint request.

Figure 11 shows the slowdown ratio and the success ratio of experiment results. It is observed that when the checkpoint interval becomes smaller, all checkpoint policies have high slowdown ratios. This is mainly because more frequent checkpoint results in the explosion of checkpoint overheads which dramatically increase the wasted time. Thus, to reduce slowdown, periodic cooperating may adopt an advice that checkpointing should be as infrequent as possible. However, Fig. 11(b) shows an infrequent checkpoint for periodic checkpointing will lead to much lower success ratio. Consequently, for periodic checkpointing, the results from this two metrics come to conflicting conclusions. In contrast, in any cases, cooperative checkpointing can keep lower slowdown ratio and much higher success ratio. For this, there are two reasons: first, cooperative checkpointing with failure prediction can keep from frequent checkpointing, thereby the overhead of checkpointing and recomputing will be reduced significantly. Second, in addition to restarting failed tasks from latest checkpoint, our checkpoint server with backup hosts can provide migration and continue execution in case that a task can not be finished during its time quota. Moreover, even if the prediction accuracy is low at 0.3, cooperative checkpointing can still gain good performance.

6 Conclusions

Since provided voluntarily, Grid resources tend to be heterogeneous and dynamic. Therefore, efficient and dependable workflow job scheduling in Grid becomes essential. Based on list scheduling and group scheduling, we propose a novel scheduling heuristic, called DAGMap. DAGMap consists of two phases: static mapping and dependable execution. The experiment results of static mapping show DAGMap can achieve better performance than other previous algorithms in terms of makespan, speedup, and efficiency. Because all static mapping heuristics are carried out prior to the workflow job execution, they cannot provide fault tolerance during runtime. To solve this problem, we design the checkpoint server with cooperative checkpointing mechanism. The experiment results of dependable execution show it can provide lower slowdown ratio and higher success ratio than periodic checkpointing.

In the future, we will make efforts to study the dynamic workflow job scheduling at the running time with QoS constraints, such as tradeoff between time and cost, advanced resource reservation. Finally, we intend to use our scheduling heuristic in our real Grid platform for practical evaluations.

Acknowledgements This paper is supported by the ChinaGrid project, the National High-Tech Research and Development Plan of China under Grant 2006AA01A115, Program for New Century Excellent Talents in University under Grant NCET-07-0334, NSF of China under Grant 60673174, 60603058, 90412010, and SRF for ROCS, SEM.

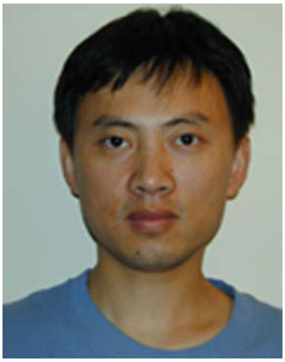
References

1. Ramakrishnan A, Singh G, Zhao H, Deelman E, Sakellariou R, Vahi K, Blackburn K, Meyers D, Samidi M (2007) Scheduling data intensive workflows onto storage-constrained distributed resources. In: Proceedings of the 7th IEEE symposium on cluster computing and the grid (CCGrid'07), 2007
2. Amin K, Hategan M, Laszewski GV, Zaluzec NJ, Hampton S, Rossi A (2004) GridAnt: a client-controllable grid workflow system. In: Proc 37th Hawai'i international conf on system science, 2004
3. Malewicz G, Foster I, Rosenberg AL, Wilde M (2007) A tool for prioritizing DAGMan jobs and its evaluation. J Grid Comput 5(2):197–212
4. Foster I (2005) Globus toolkit version 4: software for service-oriented systems. In: Lecture notes in computer science. vol 3779. Springer, Berlin
5. The Condor Project website (2007) Available: http://www.cs.wisc.edu/condor/
6. You SY, Kim HY, Hwang DH, Kim SC (2004) Task scheduling algorithm in GRID considering heterogeneous environment. In: Proc of the international conference on parallel and distributed processing techniques and applications (PDPTA '04), Nevada, USA, 2004, pp 240–245
7. Mandal A, Kennedy K, Koelbel C, Marin G, Mellor-Crummey J, Liu B, Johnsson L (2005) Scheduling strategies for mapping application workflows onto the grid. In: IEEE international symposium on high performance distributed computing (HPDC'05), 2005
8. Dong F, Akl SG (2006) Scheduling algorithms for grid computing: state of the art and open problems. Technical Report No. 2006-504, School of Computing, Queens University Kingston, Ontario

9. Topcuoglu H, Hariri S, Wu M (2002) Performance effective and low-complexity task scheduling for heterogeneous computing. IEEE Trans Parallel Distrib Syst 13(3):260–274
10. Muthuvelu N, Liu J, Soe NL, Venugopal SR, Sulistio A, Buyya R (2005) A dynamic job grouping-based scheduling for deploying applications with fine-grained tasks on global grids. In: Proc 3rd Australasian workshop on grid computing and e-research, Australia, 2005
11. Sakellariou R, Zhao H (2004) A hybrid heuristic for DAG scheduling on heterogeneous systems. In: Proc 13th heterogeneous computing workshop, USA, 2004
12. Maheswaran M, Siegel HJ (1998) A dynamic matching and scheduling algorithm for heterogeneous computing systems. In: Proc 7th heterogeneous computing workshop, 1998
13. Etminani K, Naghibzadeh PM (2007) A Min-Min Max-Min selective algorithm for grid task scheduling. In: Proc 3rd IEEE/IFIP international conference in Central Asia, 2007
14. Braun TD, Siegel HJ, Beck N, Boloni LL, Maheswaran M, Reuther AI, Robertson JP, Theys MD, Yao B (2001) A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. J Parallel Distrib Comput 61(6):810–837
15. Buyya R, Murshed M (2002) GridSim: a toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. J Concurr Comput Pract Exp (CCPE) 1175–1220
16. Hall R, Rosenberg AL, Venkataramani A (2007) A comparison of DAG-scheduling strategies for internet-based computing. In: Proc 22nd international parallel and distributed processing symposium (IPDPS), 2007
17. Sahoo RK, Oliner AJ, Rish I, Gupta M, Moreira JE, Ma S, Vilalta R, Sivasubramaniam A (2003) Critical event prediction for proactive management in large-scale computer clusters. In: Proc of the ACM SIGKDD, international conference on knowledge discovery and data mining, 2003, pp 426–435
18. Liang Y, Zhang Y, Jette M, Sivasubramaniam A, Sahoo RK (2006) Blue gene/l failure analysis and prediction models. In: Proc of the international conference on dependable systems and networks (DSN), 2006
19. Adam JO, Larry R, Ramendra KS (2006) Cooperative checkpointing: a robust approach to large-scale systems reliability. In: Proc of the 20th annual international conference on supercomputing, 2006
20. Michael L, Todd T, Jim B, Miron L (1997) Checkpoint and migration of UNIX processes in the condor distributed processing system. University of Wisconsin-Madison Computer Sciences Technical Report 1346
21. Stellner G (1996) Cocheck: checkpointing and process migration for MPI. In: Proc of the international parallel processing symposium, 1996
22. Sudakov OO, Meshcheriakov IS, Boyko YV (2007) CHPOX: transparent checkpointing system for Linux clusters. In: Intelligent data acquisition and advanced computing systems: technology and applications (IDAACS 2007), 2007, pp 159–164
23. Maoz T, Barak A, Amar L (2008) Combining virtual machine migration with process migration for HPC on multi-clusters and grids. In: IEEE Cluster, Tsukuba, 2008

Haijun Cao received his bachelor degree at the school of computer science and technology from Huazhong University of Science and Technology (HUST) in 2003. Currently, he is a Ph.D. candidate in the Cluster and Grid Computing Lab at HUST, China. His research interests include distributed computing, workflow, and service oriented computing.

Hai Jin is a professor of computer science and technology at the Huazhong University of Science and Technology (HUST) in China. He received his Ph.D. in computer engineering from HUST in 1994. In 1996, he was awarded the German Academic Exchange Service (DAAD) fellowship for visiting the Technical University of Chemnitz in Germany. He worked for the University of Hong Kong between 1998 and 2000 and participated in the HKU cluster project. He worked as a visiting scholar at the University of Southern California between 1999 and 2000. Now, he is the chief scientist of the ChinaGrid Project and Virtualization Technology for Computing System (973 Project). His research interests include cluster computing and Grid computing, P2P computing, and virtualization technology.



Xiaoxin Wu received his Ph.D. degree from the University of California, Davis, in 2001. Between 2002 and 2006, he had been working as a postdoctoral researcher in the Department of Computer Science, Purdue University, where he worked on wireless network privacy and security. Since 2006, he has been in the Intel Communication Technology Beijing Lab, working on mobile collaborative computing and broadband wireless networks. His research interests include designing and developing architecture, algorithm, and protocol for network performance improvement.

Song Wu is a professor of computer science and technology at the Huazhong University of Science and Technology (HUST) in China. He received his Ph.D. degree from HUST in 2003. In 2007, he was awarded the New Century Excellent Talents in University (NCET). His research interests include Grid computing and virtualization technology.

Xuanhua Shi is an associate professor of computer science and technology at the Huazhong University of Science and Technology (HUST) in China. He received his Ph.D. degree from HUST in 2005. Between 2006 and 2007, he had been working as a postdoctoral researcher in INRIA, France. His research interests include cluster and Grid computing, dependable computing, and virtualization technology.