Future Generation Computer Systems 27 (2011) 860–870

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Online scheduling of workflow applications in grid environments

Chih-Chiang Hsu [a], Kuo-Chan Huang [b,*], Feng-Jian Wang [a]

[a] Department of Computer Science, National Chiao-Tung University, No. 1001, Ta-Hsueh Road, Hsinchu, Taiwan

[b] Department of Computer and Information Science, National Taichung University, No. 140, Min-Shen Road, Taichung, Taiwan

ARTICLE INFO

Article history:
Received 31 May 2010
Received in revised form
10 October 2010
Accepted 25 October 2010
Available online 18 November 2010

Keywords:
Workflow
Grid
Mixed-parallel
Online scheduling

ABSTRACT

Scheduling workflow applications in grid environments is a great challenge, because it is an NP-complete problem. Many heuristic methods have been presented in the literature and most of them deal with a single workflow application at a time. In recent years, several heuristic methods have been proposed to deal with concurrent workflows or online workflows, but they do not work with workflows composed of data-parallel tasks. In this paper, we present an online scheduling approach for multiple mixed-parallel workflows in grid environments. The proposed approach was evaluated with a series of simulation experiments and the results show that the proposed approach delivers good performance and outperforms other methods under various workloads.

1. Introduction

Grid environments are an important platform for running high-performance and distributed applications. Many large-scale scientific applications are usually constructed as workflows [1–3] due to large amounts of interrelated computation and communication, e.g., Montage [4] and EMAN [5]. A grid environment is composed of widespread resources from different administrative domains. Miguel et al. [6] indicates that a grid environment usually has the characteristics: heterogeneity, large scale and geographical distribution. Task scheduling in a grid is a NP-complete problem [7,8], therefore many heuristic methods have been proposed. The workflow scheduling problem in grid environments is a great challenge. In the past years, there have been many static heuristic methods proposed [9–17]. They are designed to schedule only one single workflow at a time.

In this paper, we present a new approach called Online Workflow Management (***OWM***) for scheduling multiple online mixed-parallel workflows. There are four processes in ***OWM***: Critical Path Workflow Scheduling (CPWS), Task Scheduling, Multi-Processor Task Rearrangement and Adaptive Allocation (AA). CPWS process submits tasks into the waiting queue. Task scheduling and AA processes prioritize the tasks in the queue and assign the task with the highest priority to processors for respective execution. In data-parallel task scheduling, there may be some scheduling holes which are formed when the free processors are not enough for the first task in the queue. The multi-processor task rearrangement process works for dealing with scheduling holes to improve utilization. Many approaches can be adopted in this process, including first fit, easy backfilling [18], and conservative backfilling [18].

To evaluate the proposed OWM, we developed a simulator using discrete-event based techniques for experiments. A task-waiting queue and an event queue keep the tasks and events for processing. The grid environment is assumed to consist of several dispersed clusters, each containing a specific amount of processors. A workflow is represented by direct acyclic graph (DAG). A series of simulation experiments were conducted and the results show that ***OWM*** has better performance than ***RANK_HYBD*** [19] and ***Fairness_Dynamic*** based on the Fairness (F2) [20] in handling online workflows. For workflows composed of data-parallel tasks, the experimental results show that ***OWM(FCFS)*** performs almost equally to ***OWM(conservative)***, and outperforms ***OWM***(***easy***) and ***OWM***(***first fit***).

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 presents the ***OWM*** approach. Section 4 presents the experiments and discusses the results. Section 5 concludes the paper.

2. Related work

In the past years, most works dealing with workflow scheduling [9–17,21] were restricted to a single workflow application. Recently, some works [19,20,22–24] began to discuss the issue of multiple workflow scheduling. Zhao et al. [20] envisaged a scenario

* Corresponding author. Tel.: +886 4 22183813; fax: +886 4 22183580.
E-mail addresses: chanurnk@gmail.com (C.-C. Hsu), kchuang@mail.ntcu.edu.tw, kchuangvava@gmail.com (K.-C. Huang), fjwang@cs.nctu.edu.tw (F.-J. Wang).

0167-739X/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2010.10.015

that need to schedule multiple workflow applications at the same time. They proposed two approaches: composition approach and fairness approach.

(1) The composition approach merges multiple workflows into a single workflow first. Then, list scheduling heuristic methods, such as HEFT [13] and HHS [17], can be used to schedule the merged workflow.
(2) The main idea of the fairness approach is that when a task completes, it will re-calculate the slowdown value of each workflow against other workflows and make a decision as to which workflow should be considered next.

The composition and the fairness approaches are static algorithms and not feasible to deal with online workflow applications, i.e. multiple workflows come at different times. RANK_HYBD [19] is designed to deal with online workflow applications submitted by different users at different times. The task scheduling approach of RANK_HYBD sorts the tasks in *waiting queue* using the following rules repeatedly.

(1) If tasks in *waiting queue* come from multiple workflows, the tasks are sorted in ascending order of their rank value ($rank_u$) where $rank_u$ is described in HEFT [13];
(2) If all tasks belong to the same workflow, the tasks are sorted in descending order of their rank value ($rank_u$).

However, the number of processors to be used by each task is limited to a single processor. It is not feasible to deal with workflows composed of data-parallel tasks. T. N'takpe' et al. proposed a scheduling approach for mixed parallel applications on heterogeneous platforms [25]. Mixed parallelism is a combination of task parallelism and data parallelism where the former indicates that an application has more than one task that can execute concurrently and the latter means a task can run using more than one resource simultaneously.

The scheduling approach in [25] is only suitable for a single workflow. T. N'takpe' et al. further developed an approach to deal with concurrent mixed parallel applications [26]. Concurrent scheduling for mixed parallel applications contains two steps: constrained resource allocation and concurrent mapping. The former aims at finding an optimal allocation for each task. The number of processors is determined in this step. The latter prioritizes tasks of workflows. However, the approach in [26] is restricted to concurrent workflows submitted at the same time. It is infeasible to deal with online workflows submitted at different times. The OWM proposed in this paper is designed to deal with multiple online mixed-parallel workflows that previous methods cannot handle well.

3. Online workflow management in grid environments

This section presents the Online Workflow Management (OWM) approach proposed in this paper for multiple online mixed-parallel workflow applications. Fig. 1 shows the structure of ***OWM***. In ***OWM***, there are four processes: ***Critical Path Workflow Scheduling*** (***CPWS***), Task Scheduling, multi-processor task rearrangement and ***Adaptive Allocation*** (***AA***), and three data structures: online workflows, a grid environment and a waiting queue. The processes are represented by solid boxes, and the data structures are represented by dotted boxes.

When workflows come into the system or tasks complete successfully, ***CPWS***, takes the critical path in workflows into account, and submits the tasks of online workflows into the waiting queue. The task scheduling process in ***OWM*** adopts the RANK_HYBD method in [19]. In RANK_HYBD, the task execution order is sorted based on the length of tasks' critical path. If all tasks in the waiting queue belong to the same workflow, they are sorted in the descending order. Otherwise, the tasks in different workflows are

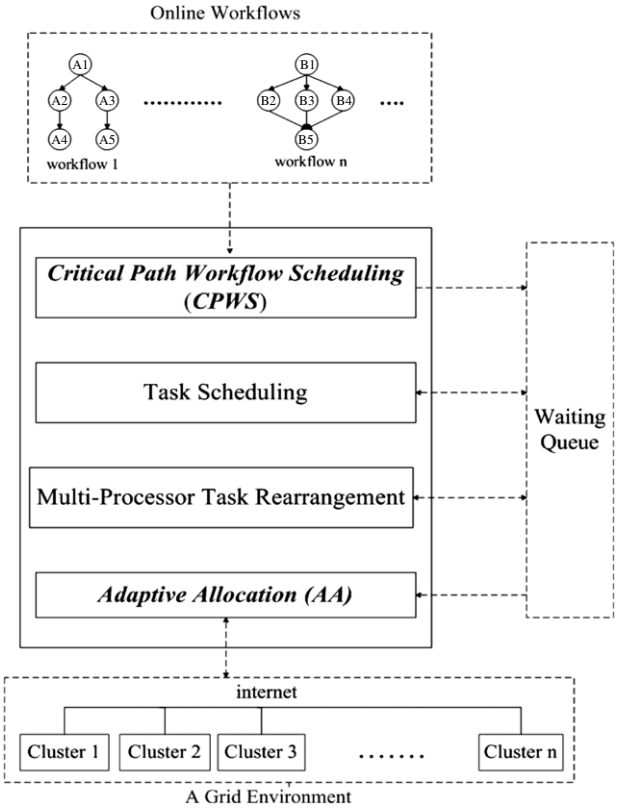


Fig. 1. Online workflow management (OWM).

sorted in the ascending order. In parallel task scheduling, there may be some scheduling holes which are formed when the free processors are not enough for the first task in the queue. The multi-processor task rearrangement process in ***OWM*** works for minimizing holes to improve utilization Several techniques might be used in the process including first fit, easy backfilling [18], and conservative backfilling [18] approaches. When there are free processors in the grid environment, ***AA*** takes the first task (the highest priority task) in the waiting queue, and selects the required processors to execute the task.

A task in a workflow has four states: *finished*, *submitted*, *ready* and *unready*. A *finished* task means the task has completed its execution successfully. A *submitted* task means the task is in the waiting queue. A task is *ready* when all necessary predecessor(s) of the task have finished, otherwise, the task is *unready*. Workflow scheduling in RANK_HYBD [19] is straightforward. It simply submits the ready tasks into the waiting queue and we call it ***Simple Workflow Scheduling*** (***SWS***) hereafter in this paper. On the other hand, in our OWM, when a new workflow arrives, ***CPWS*** is adopted to calculate $rank_u$ of each task in the workflow and sort the tasks in descending order of $rank_u$ into a list. The list is named the critical path list. Here, $rank_u$ is the upward rank of a task [13] which measures the length of critical path from a task t_i to the exit task. The definition of $rank_u$ is as below

$$rank_u(t_i) = w_i + \max_{t_j \in succ(t_i)} (c_{i,j} + rank_u(t_j))$$

where $succ(t_i)$ is the set of immediate successors of task t_i, $c_{i,j}$ is the average communication cost of edge (i, j), and w_i is the average computation cost of task t_i. The computation of a rank starts from the exit task and traverses up along the task graph recursively. Thus, the rank is called upward rank, and the upward rank of the exit task t_{exit} is

$$rank_u(t_{exit}) = w_{exit}.$$

The system maintains an array List[] and List[$workflow_i$] points to the critical path list of $workflow_i$. According to the order in each critical path list, ***CPWS*** continuously submits the ready tasks in the list into the waiting queue until running into an unready task. The details of ***CPWS*** are described in Algorithm 1.

```
Algorithm 1: CPWS

D: a set of unfinished workflows
List[]:an array of critical path lists. List[workflow_i] keeps the
critical path list of workflow_i

CPWS(D,List[])
1  begin
2    while(D ≠ ϕ) do
3      for each workflow_i ∈D do
4        according to the order List[workflow_i], continuously
         submit the ready tasks into the waiting queue until
         running into an unready task;
5    end while
6  end
```

Figs. 2 and 3 show the difference between ***SWS*** and ***CPWS***. Fig. 2 shows an example of ***SWS***. Black nodes are finished tasks, i.e., A1, A2, B1 and B3. White nodes are ready tasks, i.e., A3, A4, B2 and B4. White nodes with dotted lines are unready tasks, i.e., A5 and B5. ***SWS*** submits all ready tasks into the waiting queue, i.e., A3, A4, B2 and B4. Fig. 3 shows an example of ***CPWS***. The critical path list of each workflow is sorted in descending order of $rank_u$. The critical path list for workflow A is A1 → A2 → A3 → A5 → A4 and the critical path list for workflow B is B1 → B3 → B4 → B5 → B2. A1, A2, B1 and B3 have been finished. A3, A4, B2 and B4 are ready. A5 and B5 are unready. According to the order in the critical path lists, ***CPWS*** submits tasks A3 and B4.

The following presents the Adaptive Allocation (AA) process. To better describe the process, we define the following quantities:

- The **Estimated Computation Time ECT**($\boldsymbol{t}$, $\boldsymbol{p}$) is defined as the estimated execution time of task t on processor group p.
- The **Estimated File Communication Time EFCT**($\boldsymbol{t}$, $\boldsymbol{p}$) is defined as the estimated communication time required by task t on processor group p to receive all necessary files before execution.
- The **Estimated Available Time EAT**($\boldsymbol{t}$, $\boldsymbol{p}$) is defined as the earliest time when processor group p has a large enough time slot to execute task t.
- The **Estimated Finish Time EFT**($\boldsymbol{t}$, $\boldsymbol{p}$) is defined as the estimated time when task t completes on processor group p:

$$EFT(t, p) = EAT(t, p) + ECT(t, p) + EFCT(t, p).$$

The main idea of ***AA*** is described below:

(1) When the number of clusters that can immediately execute the first task is 1, said C_i, AA first finds the cluster, said C_j, with the earliest estimated available time among other clusters. If the estimated finish time of the first task on C_j is earlier than that on C_i, the task will be kept in the waiting queue. Otherwise, AA allocates the task to C_i for immediate execution.

(2) When the number of clusters that can accommodate the highest priority task is larger than 1, AA allocates the highest priority task to the cluster that has the earliest estimated finish time.

```
Algorithm 2: AA

T: a set of tasks in the waiting queue
R: a set of free processors
C: a set of clusters

AA(T,R,C)
01  begin
02  while(T≠ϕ and R≠ϕ) do
03    select t_i∈T where t_i with the highest priority task;
04    If workflows are composed of data-parallel tasks
05      *Multi-Processor Task Rearrangement;

06    If allocateNumberOfClusters(R, t_i) = 0
07      task t_i keeps waiting in the waiting queue;
08    else if allocateNumberOfClusters(R, t_i) = 1
09      the free processor group P_x∈C_x and calculate EFT(t_i, P_x ) ;
10      find the processor group P_v∈C_v with the earliest estimated
        available time
        among other clusters, where C_x ≠ C_v;
11      If EFT(t_i, P_x ) ≤ EFT(t_i, P_v )
12        Assign task t_i to the processor(s) P_x;
13        T = T – {t_i};
14        R = R – {P_x};
15      else
16        task t_i keeps waiting in the waiting queue;
17    else   // allocateNumberOfClusters(R, t_i) > 1
18      for each processor group P_k ∈ R do
19        calculate EFT(t_i, P_k) ;  // EAT(t_i,p_k) = current time
20      Assign task t_i to the processor group P_k that has earliest
        estimated finish time, EFT(t_i, P_k);
21      T = T – { t_i};
22      R = R – {P_k};
23  end while
24  end

25  int allocateNumberOfClusters(R, t_i){
26    numberOfCluster=0;
27    for each cluster C_i do
28      If free processors in C_i ≥ processors that t_i requires
29        numberOfClusters++;
30    return numberOfClusters;
31  }
```

The details of ***AA*** are described in Algorithm 2. When there are free processors and the waiting queue contains at least one task, AA selects the first tasks and follows the above allocation rules. In parallel task scheduling, if the number of free processors is not enough for a task, the idle processors become a scheduling hole. To overcome this problem, we perform multi-processor task rearrangement to minimize the scheduling hole as shown in lines 4–5. The techniques which can be applied in multi-processor task rearrangement include first fit, easy backfilling [18] and conservative backfilling [18]. The first fit approach allocates the first waiting task that can fit into the scheduling hole. The conservative backfilling approach moves tasks forward only if they do not delay previous tasks in the queue. The easy backfilling

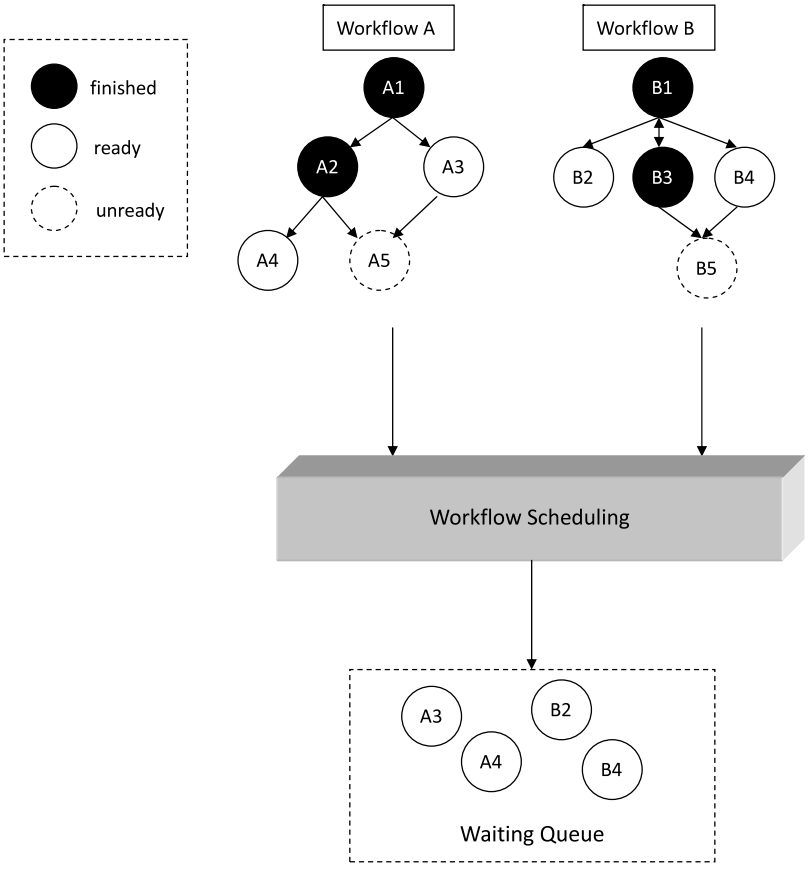


Fig. 2. An example of ***SWS***.

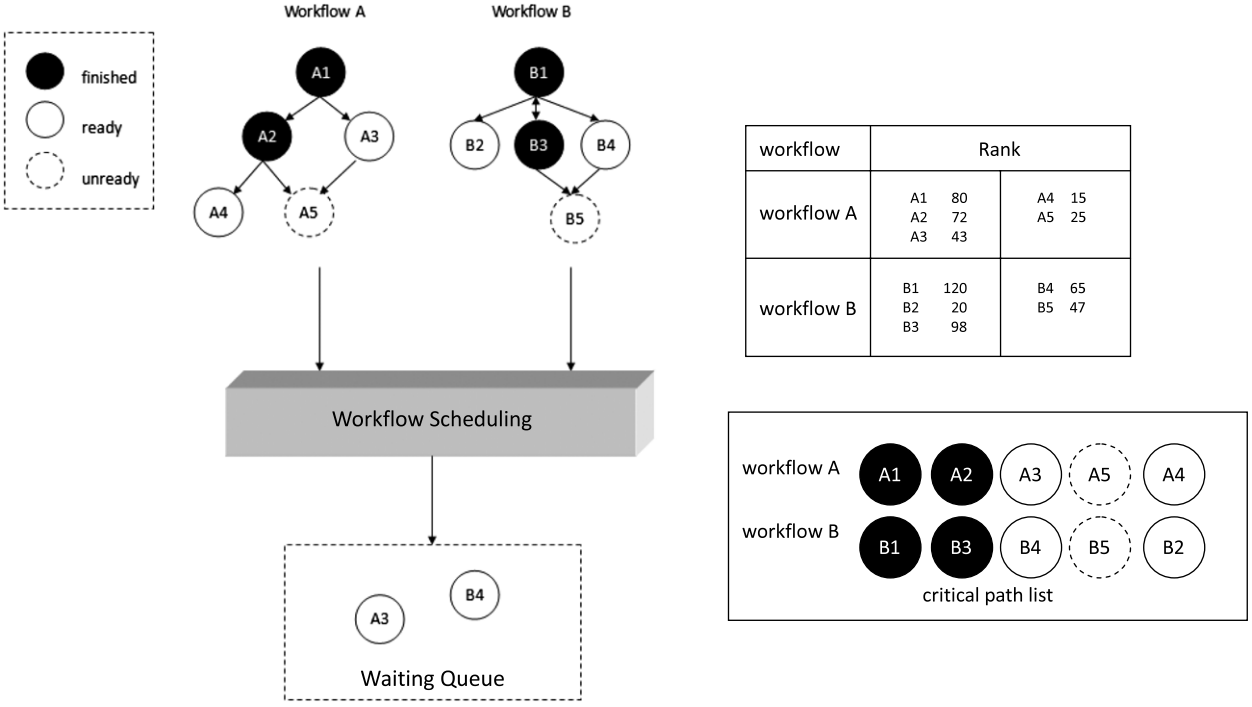


Fig. 3. An example of ***CPWS***.

approach is more aggressive and allows tasks to skip ahead provided they do not delay the job at the head of the queue [18].

Lines 25–31 show a function (*allocateNumberOfClusters*(R, t_i)). It returns the number of clusters that can accommodate the first task.

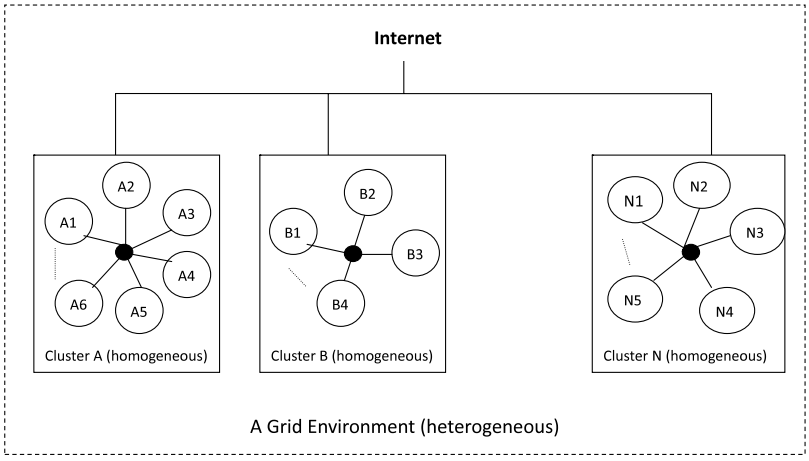


Fig. 4. A grid environment.

If the function returns 1, the steps in lines 8–16 work for rule 1 described previously. If the function returns a number larger than 1, the steps in lines 17–22 work for rule 2.

4. Simulation environments

We developed a software simulator to conduct a series of simulation experiments for evaluating the scheduling performance of OWM in a multi-cluster grid environment. In the simulator, a workflow application is represented by a direct acyclic graph (DAG). A DAG is defined as $G = (V, E)$, where V is a set of nodes, each representing a task, and E is the set of links, each representing the computation precedence order between two tasks. For example, a link $(x, y) \in E$ represents the precedence constraint that task t_x must completes before task t_y starts.

In the simulator, a grid is assumed to be composed of several clusters. A cluster contains an amount of processors. The grid is heterogeneous in that the processors at different clusters might run at different speed. On the other hand, each cluster is homogeneous, consisting of identical processors. The cost for a task includes computation and communication costs where the former means the execution time, and the latter means the data transfer time between processors. The computation cost of a task is the same for different processors in the same cluster, but may be different in different clusters. The communication cost between any two processors in the same cluster is set to be zero, but not in different clusters. Fig. 4 shows an example of a grid environment in our simulator. The processor speeds and network link speeds are homogeneous in the same cluster, but they are heterogeneous between different clusters.

In the simulator, we implemented a DAG_Generator module responsible for generating input workload consisting of a sequence of DAGs in their arrival order. The attributes and operations in DAG_Generator are described as follows.

Attributes:

1. Node: the number of tasks in a DAG.
2. Shape: the shape of a DAG.
3. OutDegree: the maximum of out degree of tasks in a DAG.
4. CCR: communication cost to computation cost ratio.
5. BRange (β): distribution range of computation cost of tasks on processors. It is the heterogeneous factor for processor speeds. A high range indicates significant differences in task's computation costs among the processors and a low range indicates that the expected execution time of a task is almost the same on each processor.
6. WDAG: the average computation cost of a DAG.
7. Cluster: the number of clusters in a grid environment.

Operations:

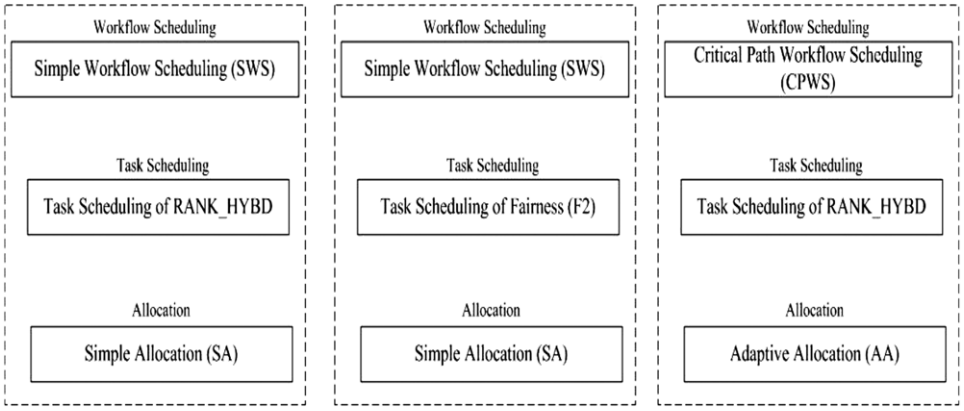
1. Generator(): randomly generates a DAG according to the 7 input parameters mentioned above. It invokes ShapeGenerator(), RelationGenerator(), CostGenerator() in turn.
2. ShapeGenerator(Node, Shape): generates the shape of a DAG using the Node and Shape parameters. The height (depth) of a DAG is randomly generated from a uniform distribution with the mean value equal to $\sqrt{Node}/Shape$. The width for each level is randomly generated from a uniform distribution with the mean value equal to $Shape \times \sqrt{Node}$. If $Shape >> 1$, it generates a shorter graph with a high parallelism degree. Otherwise, if $Shape << 1$, it generates a longer graph with a low parallelism degree.
3. RelationGenerator(Node, OutDegree): generates the connect relation of a DAG according to the input parameters Node and OutDegree defined above. Out degree of each task is randomly generated from a uniform distribution with the range [1, *OutDegree*].
4. CostGenerator(Node, BRange, WDAG, Cluster, CCR): generates the computation cost and the communication cost of a DAG. The average estimated computation cost of each task t_x, i.e., $\overline{w_x}$ is randomly generated from a distribution ranged $[1, 2 \times WDAG]$. The estimated computation cost of each task t_x on each cluster C_y, i.e., $w_{x,y}$ is randomly generated from a uniform distribution with the range:

$$\overline{w_x} \times (1 - BRange/2) \leq w_{x,y} \leq \overline{w_x} \times (1 + BRange/2).$$

5. Experimental results

This section presents the simulation experiments used to evaluate the proposed OWM approach and discuss the experimental results. The performance metrics used in our experiments are described below:

- **makespan**: the time between submission and completion of a workflow, including execution time and waiting time.
- **Schedule Length Ratio** (**SLR**): makespan usually varies widely among workflows with different sizes and other properties. To measure the scheduling efficiency objectively, we can use another performance metric derived from the makespan, which calculates the ratio of a workflow's makespan over the best possible schedule length in a given environment. The performance is called the Schedule Length Ratio (SLR) and defined by $SLR = \frac{makespan}{CPL}$ where CPL represents the Critical Path Length of a workflow. SLR is not sensitive to the size of a workflow.



(a) RANK_HYBD. (b) Fairness_Dynamic. (c) OWM.

Fig. 5. The difference between RANK_HYBD, Fairness_Dynamic and OWM.

- **win** (%): used for the comparison of different algorithms. For a workflow, one of the algorithms has the shortest makespan. The win value of an algorithm means the percentage of the workflows that have the shortest makespan when applying this algorithm. From users' perspective, a higher win value leads to a higher satisfaction.

In the following experiments, we compare OWM with two other approaches: ***RANK_HYBD*** and ***Fairness_Dynamic***. To better clarify the differences between these three approaches, we partition the complete scheduling process into three components, workflow scheduling, task scheduling and allocation approaches. Fig. 5 describes these three approaches according to the three components. ***RANK_HYBD*** [19] is shown in Fig. 5(a). The Fairness approach (F2) in [20] is a static algorithm and cannot deal with online workflows. In the following experiments, we modify the Fairness (F2) approach to handle online workflows by replacing the original workflow scheduling and allocation approaches in this approach with ***SWS*** and ***SA*** respectively. We call this new approach ***Fairness_Dynamic*** in Fig. 5(b). Here, ***SWS*** stands for ***Simple Workflow Scheduling***, which simply submits each ready task into the waiting queue, and ***SA*** represents ***Simple Allocation***, which selects the highest priority task and allocates it to the free processor group that has the earliest estimated finish time.

To experiment with different workload characteristics, we use the following parameters to generate different types of workflows. A workflow is represented as a Directed Acyclic Graph (DAG).

- Node = {20, 40, 60, 80, 100}
- Shape = {0.5, 1.0, 2.0}
- OutDegree = {1, 2, 3, 4, 5}
- CCR = {0.1, 0.5, 1.0, 1.5, 2.0}
- BRange = {0.1, 0.25, 0.5, 0.75, 1.0}
- WDAG = 100–1000.

The values of these parameters are randomly selected from the corresponding sets given above for each DAG. The arrival interval value between DAGs is set based on a Poisson distribution. Each experiment involves 20 runs, and each run has 100 unique DAGs in a grid environment that contains 3 clusters each containing 30 $\sim$ 50 processors respectively.

In the experiment, we also take other factors into account: the distribution of tasks' computation cost (*Wi_DisType*) and the computation intensity of a workflow represented by CCR (*computationIntensity*). The average computation cost of each task is randomly generated from a probability distribution within the range $[1, 2 \times WDAG]$. We experimented with both a uniform distribution and an exponential distribution for the tasks' computation cost. CCR is randomly selected from the set {0.1, 0.5, 1.0, 1.5, 2.0}. For computation-intensive workflows, CCR is randomly selected form the set {0.1, 0.5}, and for communication-intensive workflows, CCR is randomly selected from the set {1.5, 2.0}.

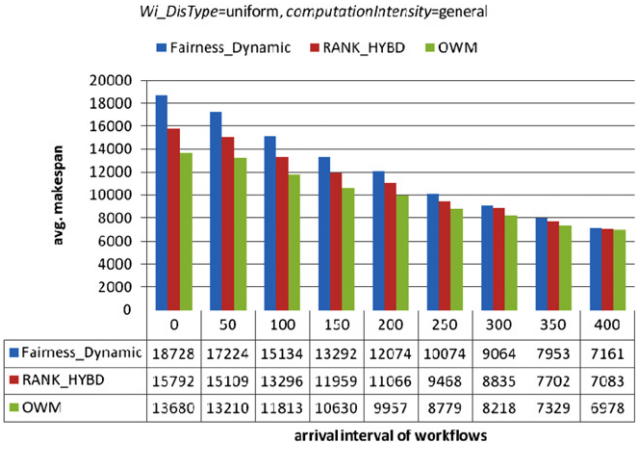


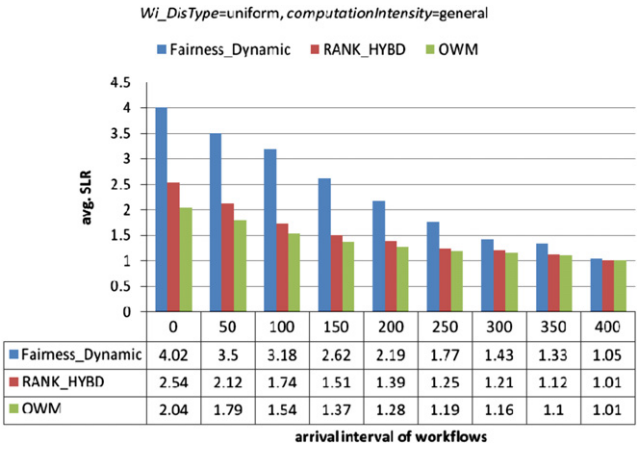
Fig. 6. Results of different mean arrival intervals for average makespan.

5.1. Impact of arrival interval of workflows

Figs. 6–8 show the results of different mean arrival intervals according to different performance metrics: average makespan, average SLR and win (%) respectively. It can be easily seen that when the system is more crowded, i.e., smaller arrival interval in the figures, ***OWM*** outperforms the other two algorithms significantly. When all DAGs are submitted at the same time, i.e., the zero arrival interval in the figures, ***OWM*** outperforms ***Fainess_Dynamic*** by 26% and 49%, and outperforms ***RANK_HYBD*** by 13% and 20% for average makespan and average SLR respectively, as shown in Figs. 6 and 7. ***Fairness_Dynamic*** has poor performance for average SLR, because it achieves fairness by the cost of enlarging the makespan of the workflows with shorter critical path length. ***OWM*** wins in terms of makespan in 94.55% of workflows as shown in Fig. 8. From a users' perspective, it means that 94.55% users may prefer ***OWM***. When workflows arrive at an interval about 400 time units, these three algorithms perform almost equivalently for average makespan, average SLR and win (%) because one workflow almost comes in after another one finishes. In real environments, most high-performance centers are overloaded, therefore ***OWM*** can outperform others in such environments.

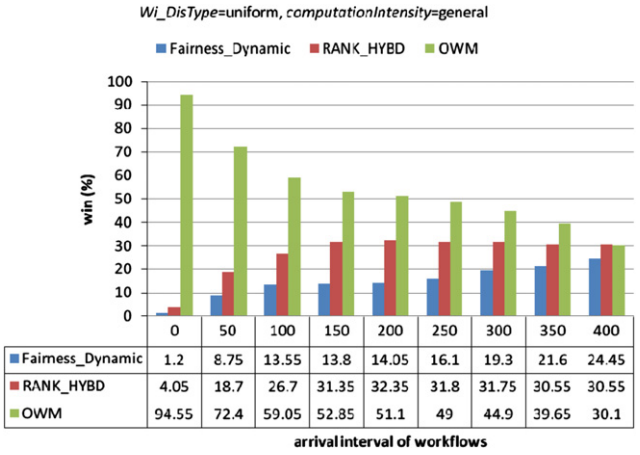
5.2. Impact of computation intensity with different distributions of tasks' computation cost

Figs. 9–11 show the results of different levels of computation intensity for average makespan, average SLR and win (%), respectively, with a uniform distribution of tasks' computation cost.



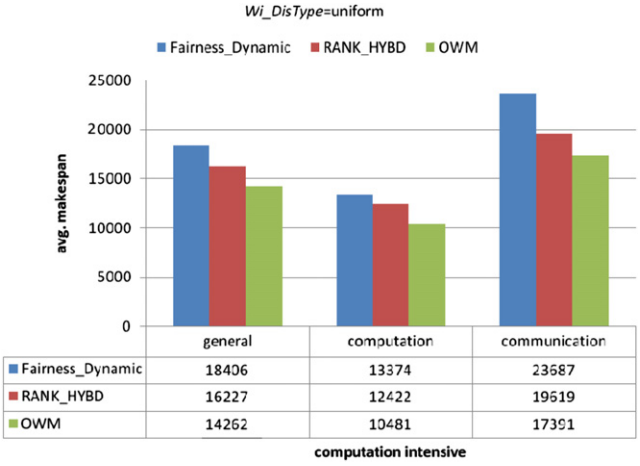
	0	50	100	150	200	250	300	350	400
Fairness_Dynamic	4.02	3.5	3.18	2.62	2.19	1.77	1.43	1.33	1.05
RANK_HYBD	2.54	2.12	1.74	1.51	1.39	1.25	1.21	1.12	1.01
OWM	2.04	1.79	1.54	1.37	1.28	1.19	1.16	1.1	1.01

Fig. 7. Results of different mean arrival intervals for average SLR.



	0	50	100	150	200	250	300	350	400
Fairness_Dynamic	1.2	8.75	13.55	13.8	14.05	16.1	19.3	21.6	24.45
RANK_HYBD	4.05	18.7	26.7	31.35	32.35	31.8	31.75	30.55	30.55
OWM	94.55	72.4	59.05	52.85	51.1	49	44.9	39.65	30.1

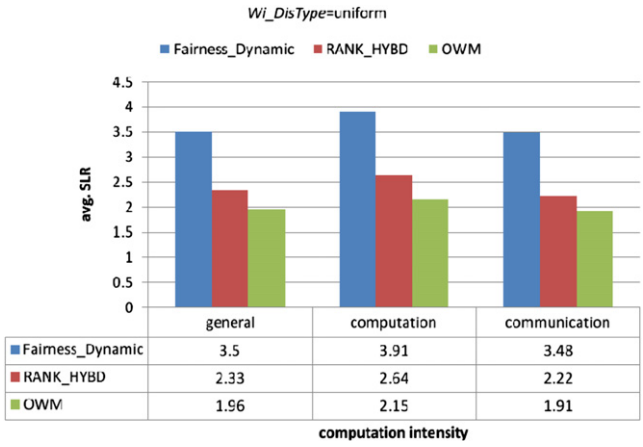
Fig. 8. Results of different mean arrival intervals for win (%).



	general	computation	communication
Fairness_Dynamic	18406	13374	23687
RANK_HYBD	16227	12422	19619
OWM	14262	10481	17391

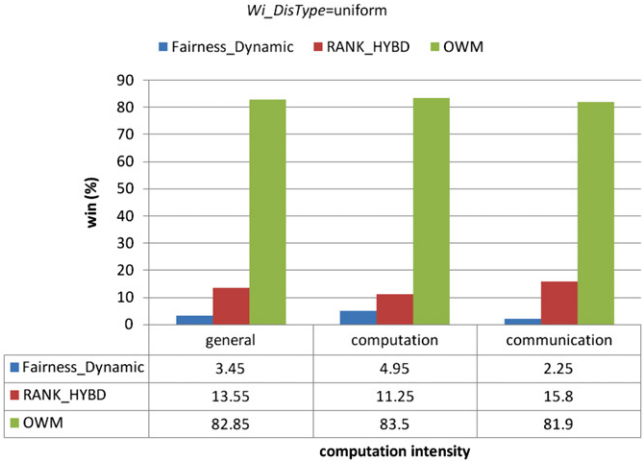
Fig. 9. Results of different computation intensity for average makespan with a uniform distribution of tasks' computation cost.

Figs. 12–14 show the results with an exponential distribution of tasks' computation cost. In these experiments, the arrival interval of workflows is set according to a Poisson distribution with the mean value of 20. It represents a situation that several workflows may be simultaneously running in the system. Obviously, ***OWM*** outperforms the other two algorithms. The superiority of ***OWM*** over the other two methods is that it consistently achieves the best performance for all types of workflows.



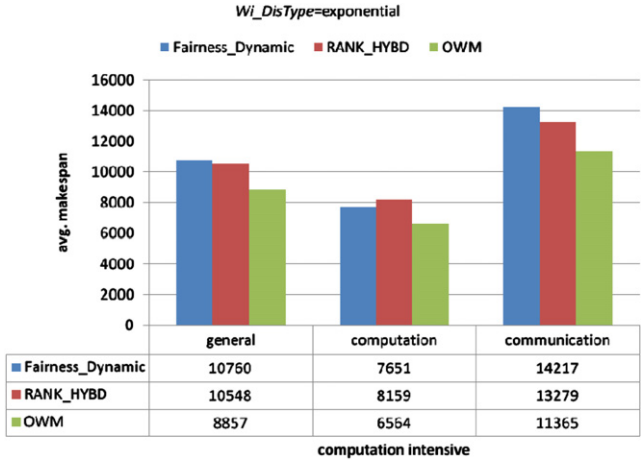
	general	computation	communication
Fairness_Dynamic	3.5	3.91	3.48
RANK_HYBD	2.33	2.64	2.22
OWM	1.96	2.15	1.91

Fig. 10. Results of different computation intensity for average SLR with a uniform distribution of tasks' computation cost.



	general	computation	communication
Fairness_Dynamic	3.45	4.95	2.25
RANK_HYBD	13.55	11.25	15.8
OWM	82.85	83.5	81.9

Fig. 11. Results of different computation intensity for win (%) with a uniform distribution of tasks' computation cost.



	general	computation	communication
Fairness_Dynamic	10760	7651	14217
RANK_HYBD	10548	8159	13279
OWM	8857	6564	11365

Fig. 12. Results of different computation intensity for average makespan with an exponential distribution of tasks' computation cost.

In Fig. 9, ***OWM*** outperforms ***Fairness_Dynamic*** by 23%, 22% and 27%, and outperforms ***RANK_HYBD*** by 12%, 17% and 11% in terms of average makespan for general, computation, and communication intensive workflows respectively. In Fig. 10, ***OWM*** outperforms ***Fairness_Dynamic*** by 44%, 45% and 45%, and outperforms ***RANK_HYBD*** by 16%, 19% and 14% in terms of SLR for general, com-

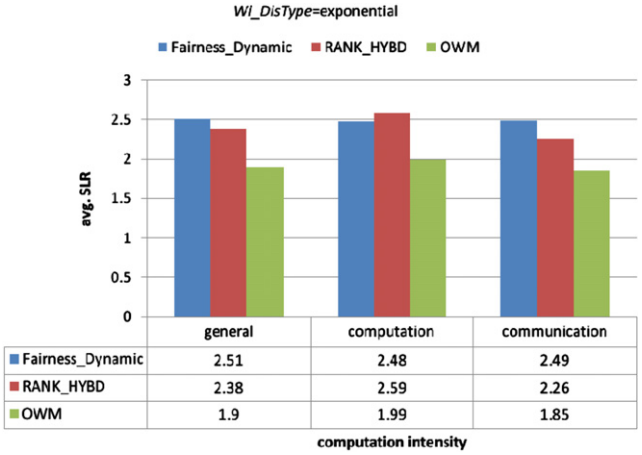


Fig. 13. Results of different computation intensity for average SLR with an exponential distribution of tasks' computation cost.

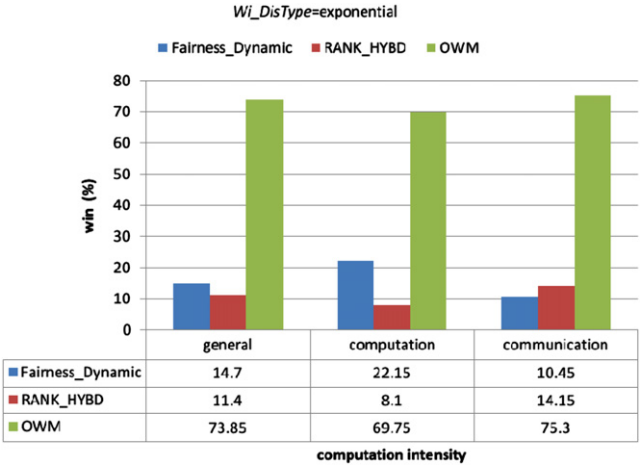


Fig. 14. Results of different computation intensity for win (%) with an exponential distribution of tasks' computation cost.

putation, and communication intensive workflows respectively. ***OWM*** wins in terms of makespan by about 82% for all the three types of workflows as show in Fig. 11.

In Fig. 12, OWM outperforms Fairness_Dynamic by 17%, 14% and 20%, and outperforms RANK_HYBD by 16%, 20% and 15% in terms of average makespan for general, computation, and communication intensive workflows respectively. In Fig. 13, ***OWM*** outperforms ***Fairness_Dynamic*** by 24%, 20% and 26%, and outperforms ***RANK_HYBD*** by 20%, 23% and 18% in terms of average SLR for general, computation, and communication intensive workflows respectively. ***OWM*** wins in terms of makespan by about 72% for all the three types of workflows as shown in Fig. 14.

5.3. Impact of the number of clusters

The following investigates the effects of the number of clusters in a grid environment. In the experiment, the grid environment is composed of 120 processors, and divided equally into a different number of clusters, 2, 4, 6, 8, 10 and 12, for each test case. We assume that the arrival interval of workflows conforms to the Poisson distribution with the mean value of 20. The results of the following figures indicate that OWM performs the best in all amounts of clusters. In average, ***OWM*** outperforms ***Fairness_Dynamic*** by 22%, and outperforms ***RANK_HYBD*** by 11% in terms of average makespan as shown in Fig. 15. ***OWM*** outperforms ***Fairness_Dynamic*** by 48%, and outperforms

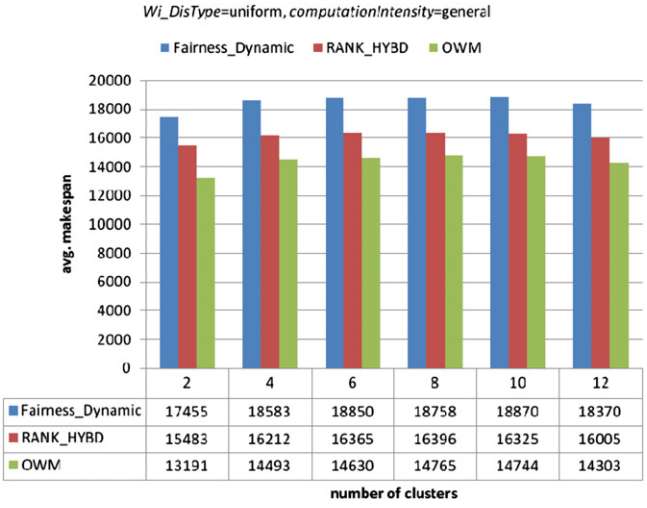


Fig. 15. Results of different numbers of clusters for average makespan.

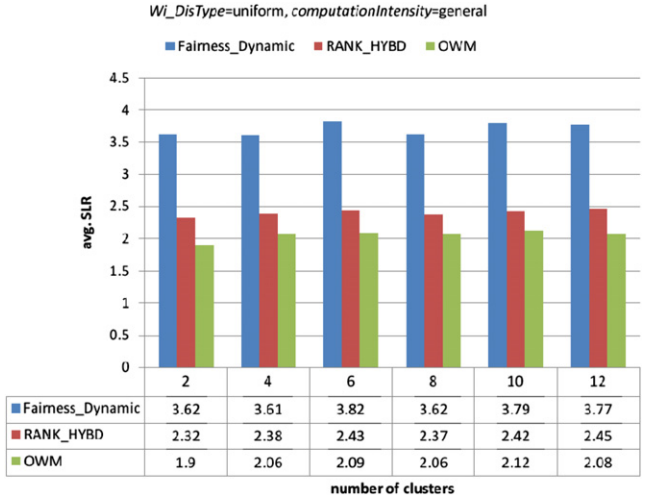


Fig. 16. Results of different numbers of clusters for average SLR.

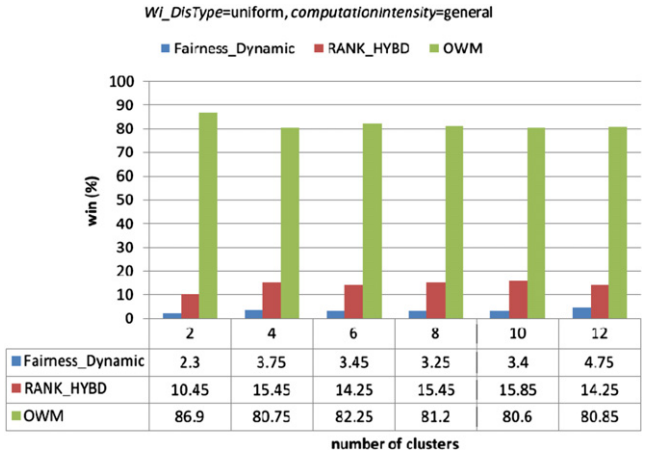
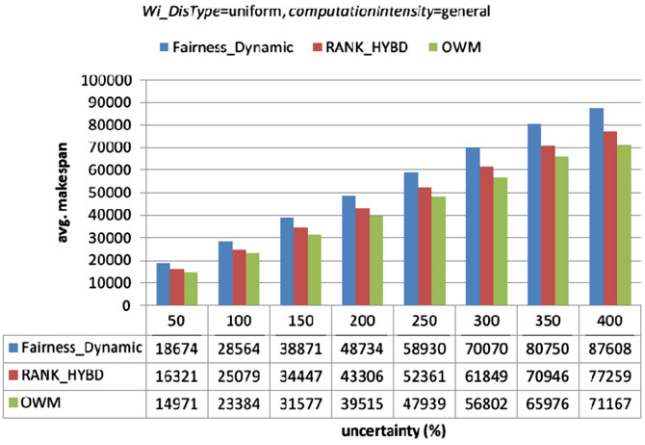


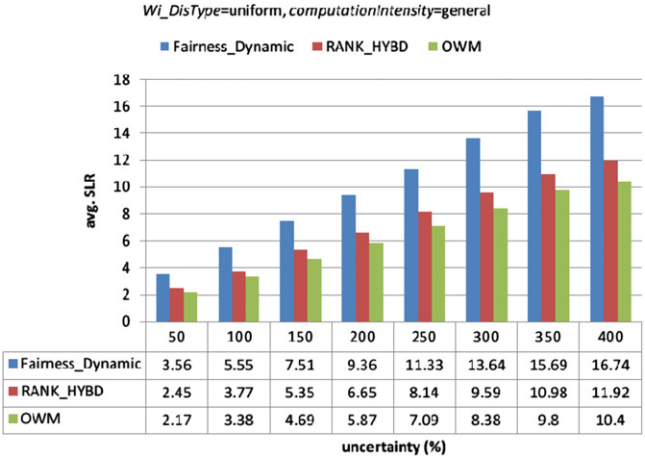
Fig. 17. Results of different numbers of clusters for win (%).

RANK_HYBD by 15% in terms of average SLR as shown in Fig. 16. ***OWM*** wins in terms of makespan by about 82% as shown in Fig. 17.



	50	100	150	200	250	300	350	400
Fairness_Dynamic	18674	28564	38871	48734	58930	70070	80750	87608
RANK_HYBD	16321	25079	34447	43306	52361	61849	70946	77259
OWM	14971	23384	31577	39515	47939	56802	65976	71167

Fig. 18. Results of inaccurate execution time estimates for average makespan.



	50	100	150	200	250	300	350	400
Fairness_Dynamic	3.56	5.55	7.51	9.36	11.33	13.64	15.69	16.74
RANK_HYBD	2.45	3.77	5.35	6.65	8.14	9.59	10.98	11.92
OWM	2.17	3.38	4.69	5.87	7.09	8.38	9.8	10.4

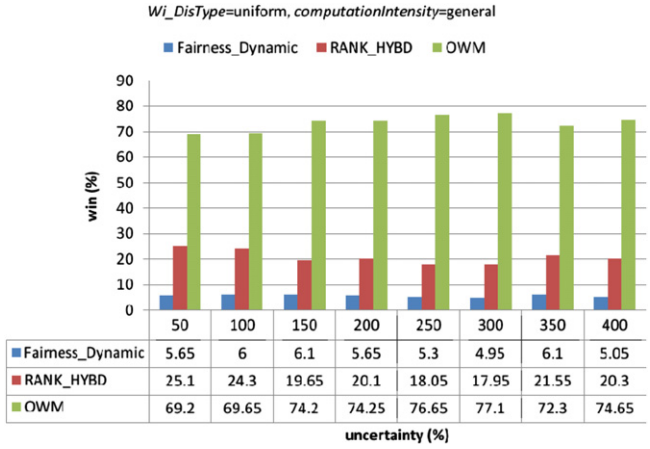
Fig. 19. Results of inaccurate execution time estimates for average SLR.

5.4. Impact of inaccurate execution time estimates

The execution time of each task on a specific processor is necessary information for workflow scheduling algorithms. In practice, the execution time of a task is usually difficult to know before the execution completes. Therefore, the estimated execution time used in scheduling algorithms is not precise. The following experiments investigate the effects of inaccurate estimation of task execution time. The simulator picks the actual execution time of a task randomly from the range:

$$\left[1, \left(et + 2 \times \frac{uncertainty}{100} \times et\right)\right]$$

where *et* is the estimated execution time of the task. For example, when the uncertainty is 400 and *et* of a task is 100, the actual execution time of the task is randomly picked from the range: [1, 900]. We also assume that the arrival interval of workflows conforms to the Poisson distribution with the mean value of 20. Figs. 18–20 show the results of inaccurate execution time estimates for average makespan, average SLR and win (%) respectively. It can be easily observed that ***OWM*** outperforms the other two algorithms for the uncertainty levels from 50% to 400%. In average, ***OWM*** outperforms ***Fairness_Dynamic*** by 19%, and outperforms ***RANK_HYBD*** by 8% for average makespan as shown in Fig. 18; ***OWM*** outperforms ***Fairness_Dynamic*** by 38%, and outperforms ***RANK_HYBD*** by 12% for average SLR as shown in Fig. 19. ***OWM*** wins in terms of makespan by about 74% as shown in Fig. 20. These results indicate that ***OWM*** is more resilient to inaccurate estimation of task execution time than the other methods.



	50	100	150	200	250	300	350	400
Fairness_Dynamic	5.65	6	6.1	5.65	5.3	4.95	6.1	5.05
RANK_HYBD	25.1	24.3	19.65	20.1	18.05	17.95	21.55	20.3
OWM	69.2	69.65	74.2	74.25	76.65	77.1	72.3	74.65

Fig. 20. Results of inaccurate execution time estimates for win (%).

5.5. Results for workflows composed of data-parallel tasks

In this section, we compare different multi-processor task rearrangement processes, including first fit, easy backfilling [18] and conservative backfilling [18], with the basic FCFS (First Come First Serve) approach. The FCFS approach does not try to choose among waiting tasks to fit the scheduling hole. In the following experiment, ***OWM(FCFS)***, ***OWM(conservative)***, ***OWM(easy)*** and ***OWM(first fit)*** stand for FCFS, conservative backfilling, easy backfilling and first fit approaches respectively.

The experimental setups for data-parallel tasks are the same as that for single-processor tasks, but each run has 50 unique workflows. To be more realistic, the maximal required processors of all tasks (*maxTaskNP*) and the distribution of the processors that tasks require (*NP_DisType*) are taken into account. In the experiment, *maxTaskNP* is defined with maximum, half and minimum.

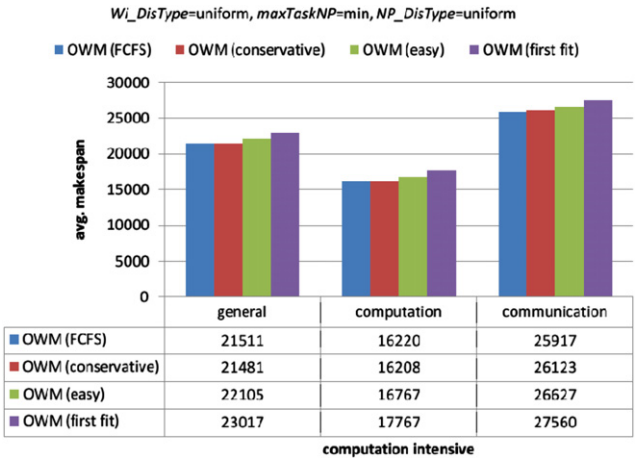
- *maxTaskNP* is maximum: *maxTaskNP* = the number of processors in the smallest cluster in the grid's environment.
- *maxTaskNP*is half: *maxTaskNP* = 1/2 × the number of processors in the smallest cluster in the grid's environment.
- *maxTaskNP* is minimum: *maxTaskNP* = 1/5 × the number of processors in the smallest cluster in the grid's environment.

The required processors of each task is randomly generated from a probability distribution within the range [1, *maxTaskNP*]. The experiment was conducted with uniform distribution, exponential distribution and normal distribution for *NP_DisType*.

Each experiment is configured by the following four parameters. Their values are assigned from the corresponding sets below:

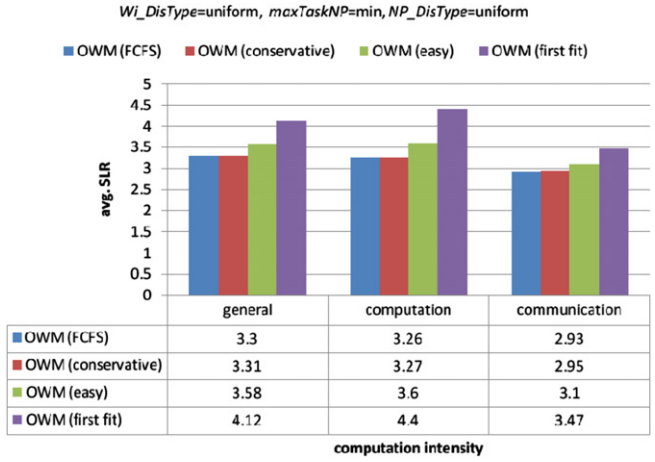
- *Wi_DisType* = {uniform, exponential}
- *maxTaskNP* = {maximum, half, minimum}
- *NP_DisType* = {uniform, exponential, normal}
- *computationIntensity* = {general, computation, communication}.

The combinations of the above parameter values give 54 different experiments. These 54 experiments lead to an interesting observation for workflow scheduling. In independent task scheduling, it is well known that the FCFS approach has worse performance than conservative backfilling [18], easy backfilling and first fit approaches. However, for workflow scheduling, the experiments show that ***OWM(FCFS)*** is almost equal to ***OWM(conservative)***, and outperforms ***OWM(easy)*** and ***OWM(first fit)*** for average makespan and average SLR. Moreover, it outperforms the other three approaches for win (%). The results indicate that that when the



	general	computation	communication
OWM (FCFS)	21511	16220	25917
OWM (conservative)	21481	16208	26123
OWM (easy)	22105	16767	26627
OWM (first fit)	23017	17767	27560

Fig. 21. Results of different computation intensity for average makespan with (uniform, min, uniform).

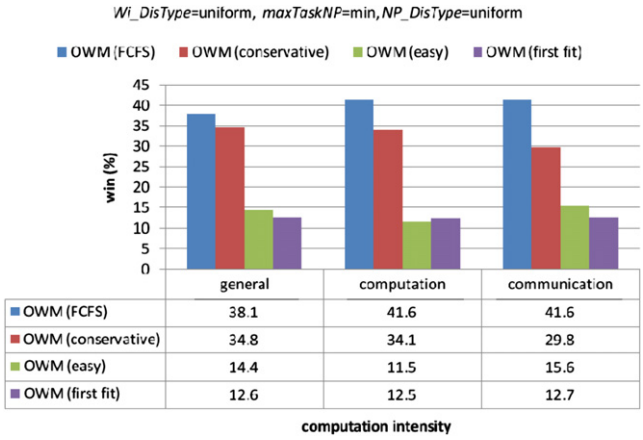


	general	computation	communication
OWM (FCFS)	3.3	3.26	2.93
OWM (conservative)	3.31	3.27	2.95
OWM (easy)	3.58	3.6	3.1
OWM (first fit)	4.12	4.4	3.47

Fig. 22. Results of different computation intensity for average SLR with (uniform, min, uniform).

waiting tasks are rearranged by the multi-processor task rearrangement process, it does not result in better performance as expected. The following example helps to explain the reason. The tasks in the waiting queue are sorted by their critical path in ascending order. Suppose that a workflow is near completion, and the last task in the workflow will get the highest priority in the waiting queue. If the task cannot be allocated due to the lack of free processors, the multi-processor task rearrangement process will find a waiting task to fit the scheduling hole. The above rearrangement technique may make the last task wait for a long time, and it in turn increases the makespan of the workflow. So, the performance may be reduced correspondingly. Therefore, frequent task rearrangement may contrarily lead to poor performance. For example, ***OWM(first fit)*** has the highest frequency of task rearrangement, and results in the worst performance in the experiments.

All the 54 experiments give the same trend of performance results. Figs. 21–23 are an example among the 54 experiments and show the results of different computation intensity for average makespan, average SLR and win (%) respectively, with *Wi_DisType* = uniform, *maxTaskNP* = min, and *NP_DisType* = uniform.



	general	computation	communication
OWM (FCFS)	38.1	41.6	41.6
OWM (conservative)	34.8	34.1	29.8
OWM (easy)	14.4	11.5	15.6
OWM (first fit)	12.6	12.5	12.7

Fig. 23. Results of different computation intensity for win (%) with (uniform, min, uniform).

6. Conclusions

Most workflow scheduling algorithms are restricted to handle only one single workflow. There are few researches for scheduling online workflows. In the paper, we propose an online workflow management ***(OWM)*** approach for scheduling multiple online mixed-parallel workflows in a grid environment. Our experiments show that ***OWM*** outperforms ***RANK_HYBD*** and ***Fairness_Dynamic*** for average makespan, average SLR and win (%) under different experimental workloads.

Moreover, ***RANK_HYBD*** and ***Fairness_Dynamic*** do not work with mixed-parallel workflows composed of data-parallel tasks. There are few studies focused on mixed-parallel workflow scheduling. Our OWM takes this issue into account. OWM incorporates well-known approaches, e.g. first fit, easy backfilling and conservative backfilling, to deal with the allocation issue for workflows composed of data-parallel tasks.

Acknowledgement

The work of this paper is partially supported by the National Science Council in Taiwan under NSC 96-2221-E-432 -003 -MY3.

References

[1] E. Deelman, D. Gannon, M. Shields, I. Taylor, Workflows and e-science: an overview of workflow system features and capabilities, Future Gener. Comput. Syst. 25 (5) (2009) 528–540.
[2] M.J. Fairman, A.R. Price, G. Xue, M. Molinari, D.A. Nicole, T.M. Lenton, R. Marsh, K. Takeda, S.J. Cox, Earth system modelling with windows workflow foundation, Future Gener. Comput. Syst. 25 (5) (2009) 586–597.
[3] E. Elmroth, F. Hernández, J. Tordsson, Three fundamental dimensions of scientific workflow interoperability: model of computation, language, and execution, Future Gener. Comput. Syst. 26 (2) (2010) 245–256.
[4] G. Singh, E. Deelman, G.B. Berriman, et al., Montage: a grid enabled image mosaic service for the national virtual observatory, Astron. Data Anal. Softw. Syst. 13 (2003).
[5] S. Ludtke, P. Baldwin, W. Chiu, EMAN: semiautomated software for high resolution single-particle reconstructions, J. Struct. Biol. 128 (1999) 82–97.
[6] L.B. Miguel, A.D. Yannis, G.S. Eduardo, Grid characteristics and uses: a grid definition, in: Proceedings of the First European Across Grids Conference, ACG'03, Santiago de Compostela, Spain, Feb., 2004.
[7] J.D. Ullman, NP-complete scheduling problems, J. Comput. Syst. Sci. 10 (1975) 384–393.
[8] M.R. Gary, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman and Co., 1979.
[9] M. Wu, D. Gajski, Hypertool: a programming aid for message passing systems, IEEE Trans. Parallel Distrib. Syst. 1 (1990) 330–343.
[10] Y. Kwok, I. Ahmad, Dynamic critical-path scheduling: an effective technique for allocation task graphs to multi-processors, IEEE Trans. Parallel Distrib. Syst. 7 (5) (1996) 506–521.
[11] G.C. Sih, E.A. Lee, A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures, IEEE Trans. Parallel Distrib. Syst. 4 (2) (1993) 175–186.
[12] H. El-Rewini, T.G. Lewis, Scheduling parallel program tasks onto arbitrary target machines, J. Parallel Distrib. Comput. 9 (1990) 138–153.

[13] H. Topcuoglu, S. Hariri, M.Y. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Trans. Parallel Distrib. Syst. 2 (13) (2002) 260–274.
[14] T. Yang, A. Gerasoulis, DSC: scheduling parallel tasks on an unbounded number of processors, IEEE Trans. Parallel Distrib. Syst. 5 (9) (1994) 951–967.
[15] G. Park, B. Shirazi, J. Marquis, DFRN: a new approach for duplication based scheduling for distributed memory multi-processor systems, in: Proceedings of Int'l Conf. Parallel Processing, 1997.
[16] A. Mandal, K. Kennedy, C. Koelbel, G. Marin, J. Mellor-Crummey, B. Liu, L. Johnsson, Scheduling strategies for mapping application workflows onto the grid, in: Proceedings of the 14th IEEE Symposium on High Performance Distributed Computing, HPDC 14, 2005.
[17] R. Sakellariou, H. Zhao, A hybrid heuristic for DAG scheduling on heterogeneous systems, in: Proceedings of the 18th International Parallel and Distributed Processing Symposium, 2004.
[18] A.W. Mu'alem, D.G. Feitelson, Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling, IEEE Trans. Parallel Distrib. Syst. 12 (6) (2001).
[19] Z. Yu, W. Shi, A planner-guided scheduling strategy for multiple workflow applications, in: Proceedings of ICPP-W, Sept., 2008.
[20] H. Zhao, R. Sakellarious, Scheduling multiple DAGs onto heterogeneous systems, in: Proceedings of the 15th Heterogeneous Computing Workshop, Rhodes Island, Greece, April, 2006.
[21] M. Rahman, R. Ranjan, R. Buyya, Cooperative and decentralized workflow scheduling in global grids, Future Gener. Comput. Syst. 26 (5) (2010) 753–768.
[22] L. Bittencourt, E. Madeira, Towards the scheduling of multiple workflows on computational grids, J. Grid Comput. 22 (2009).
[23] M. Wieczorek, A. Hoheisel, R. Prodan, Taxonomies of the multi-criteria grid workflow scheduling problem, in: Proceedings of the CoreGRID Workshop on Grid Middleware, Dresden, Germany, June, 2007.
[24] M. Wieczorek, A. Hoheisel, R. Prodan, Towards a general model of the multi-criteria workflow scheduling on the grid, Future Gener. Comput. Syst. 25 (3) (2009) 237–256.
[25] T. N'takpe', F. Suter, A comparison of scheduling approaches for mixed-parallel applications on heterogeneous platforms, in: Proceedings of the 6th International Symposium on Parallel and Distributed Computing, IS-PDC, Hagenberg, Austria, July, 2007.
[26] T. N'takpe', F. Suter, Concurrent Scheduling of Parallel Task Graphs on Multi-Clusters Using Constrained Resource Allocations, Technical Report: Rapport de recherché no. 6774, December 2008.

Chih-Chiang Hsu received his M.S. degree in Computer Science and Information Engineering from National Chiao-Tung University, Taiwan ROC, in 2009. He is now serving his military service in Taiwan. His main research interests include distributed computing and workflow technology.

Kuo-Chan Huang received his B.S. and Ph.D. degrees in Computer Science and Information Engineering from National Chiao-Tung University, Taiwan, in 1993 and 1998, respectively. He is currently an Assistant Professor in Computer and Information Science Department at National Taichung University, Taiwan. He is a member of ACM and IEEE Computer Society. His research areas include parallel processing, cluster, grid, and cloud computing, workflow computing.

Feng-Jian Wang completed his Ph.D. program in Dept. of E.E.C.S., Northwestern University, 1988. Since then, he worked in National Chiao-Tung University, Taiwan. During his Ph.D. program, he worked on incremental analysis of data flow. Thereafter, he worked on the development, reuse, and data analysis of object-oriented programming language. Since 1995, he worked on the analysis and design of workflow programs and his laboratory constructed a workflow management system named Agentflow, where he studied a series of supporting analysis and tools associated with editing activities. Currently, he is focused on workflow, SOA and cloud computing techniques, and developing a rich interface pattern for virtual reality.