Online Scheduling of Workflow Applications in Grid Environment

Chih-Chiang Hsu¹, Kuo-Chan Huang², and Feng-Jian Wang¹

¹ Department of Computer Science
National Chiao Tung University
1001, University Road, Hsinchu, Taiwan
chanurnk@gmail.com, fjwang@cs.nctu.edu.tw
² Department of Computer and Information Science
National Taichung University
140, Min-Shen Road, Taichung, Taiwan
kchuang@mail.ntcu.edu.tw

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, there are several heuristic methods 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.

Keywords: workflow, grid, mixed-parallel, online scheduling.

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 due to large amounts of interrelated computation and communication, e.g., Montage [12] and EMAN [11]. A Grid environment is composed of widespread resources from different administrative domains. Miguel et al. [1] indicates that a Grid environment usually has the characteristics: heterogeneity, large scale and geographical distribution. Task scheduling in Grid is a NP-complete problem [2] [10], therefore many heuristic methods have been proposed. The workflow scheduling problem in Grid environments is a great challenge. In the past years, there are many static heuristic methods proposed [3] [4] [5] [6] [7] [8] [9] [14] [18]. 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 highest priority to processors for execution respectively. 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 [16], and conservative backfilling [16].

To evaluate the proposed OWM, we developed a simulator using discrete-event based techniques for experiments. Task-waiting queue and 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** [17] and **Fairness_Dynamic** based on the Fairness (F2) [19] 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 [3] [4] [5] [6] [7] [8] [9] [14] [18] were restricted to single workflow application. Zhao et al. [19] envisaged a scenario 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 [7] and HHS [18], can be used to schedule the merged workflow.
2. The main idea of fairness approach is that when a task completes, it will recalculate the slowdown value of each workflow against other workflows and make a decision on 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 [17] 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 \( \text{rank}_u \) where \( \text{rank}_u \) is described in HEFT [7];
2. If all tasks belong to the same workflow, the tasks are sorted in descending order of their rank value \( \text{rank}_u \).
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 [13]. 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 [13] is only suitable for a single workflow. T. N'takpe' et al. further developed an approach to deal with concurrent mixed parallel applications [15]. 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 [15] 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. Figure 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 [17]. 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 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 [16], and conservative backfilling [16] 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. 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 $\text{rank}_u$ into a list. The list is named the critical path list. Here, $\text{rank}_u$ is the upward rank of a task [7] which measures the length of critical path from a task $t_i$ to the exit task. The definition of $\text{rank}_u$ is as below:

$$\text{rank}_u(t_i) = \overline{w}_i + \max_{t_j \in \text{succ}(t_i)} (c_{i,j} + \text{rank}_u(t_j))$$  \hspace{1cm} (1)$$

where $\text{succ}(t_i)$ is the set of immediate successors of task $t_i$, $c_{i,j}$ is the average communication cost of edge $(i, j)$, and $\overline{w}_i$ 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 $t_{\text{exit}}$ is:

$$\text{rank}_u(t_{\text{exit}}) = \overline{w}_{\text{exit}}$$  \hspace{1cm} (2)$$

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

Figure 2 shows an example of $\text{CPWS}$. The critical path list of each workflow is sorted in descending order of $\text{rank}_u$. The critical path list for workflow A is $A1 \rightarrow A2 \rightarrow A3 \rightarrow A5 \rightarrow A4$ and the critical path list for workflow B is $B1 \rightarrow B3 \rightarrow B4 \rightarrow B5 \rightarrow B2$. A1, A2, B1 and B3 have finished. A3, A4, B2 and B4 are ready. A5 and B5 are unready. According to the order in the critical path lists, $\text{CPWS}$ submits tasks A3 and B4.
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.

\[ \text{CPWS}(D, \text{List[]}) \]
1 begin
2 while \((D \neq \emptyset)\) do
3 \hspace{1em} for each workflow_i \in D do
4 \hspace{2em} according to the order List[workflow_i], continuously submit the ready tasks into the waiting queue until running into an unready task;
5 \hspace{1em} end while
6 end

Fig. 2. An example of CPWS

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

- The Estimated Computation Time \( ECT(t, p) \) is defined as the estimated execution time of task \( t \) on processor group \( p \).
- The Estimated File Communication Time \( EFCT(t, p) \) 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 \( EAT(t, p) \) is defined as the earliest time when processor group \( p \) has a large enough time slot to execute task \( t \).
- The Estimated Finish Time \( EFT(t, p) \) is defined as the estimated time when task \( t \) completes on processor group \( p \):

\[
EFT(t, p) = EAT(t, p) + ECT(t, p) + EFCT(t, p)
\] (3)
The main idea of AA is described below:

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

<table>
<thead>
<tr>
<th>Algorithm 2: AA</th>
</tr>
</thead>
</table>

\[ T \]: 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 \((T \neq \emptyset \text{ and } R \neq \emptyset)\) do  
03 select \(t_i \in T\), where \(t_i\) with the highest priority task;  
04 if workflows are composed of data-parallel tasks  
05 *Multi-Processor Task Rearrangement;  
06 if \(\text{allocateNumberOfClusters}(R, t_i) = 0\)  
07 task \(t_i\) keeps waiting in the waiting queue;  
08 else if \(\text{allocateNumberOfClusters}(R, t_i) = 1\)  
09 find the processor group \(P_{x} \in C_{y}\) and calculate \(EFT(t_i, P_{x})\);  
10 if \(EFT(t_i, P_{x}) \leq EFT(t_i, P_{y})\)  
11 Assign task \(t_i \) to the processor(s) \(P_{x}\);  
12 \(T = T - \{t_i\}\);  
13 \(R = R - \{P_{x}\}\);  
14 else  
15 task \(t_i\) keeps waiting in the waiting queue;  
16 \(\text{else } // \text{allocateNumberOfClusters}(R, t_i) > 1\)  
17 for each processor group \(P_{k} \in R\) do  
18 calculate \(EFT(t_i, P_{k})\); // \(EAT(t_i, P_{k}) = \) current time  
19 Assign task \(t_i\) to the processor group \(P_{k}\) that has earliest estimated finish time, \(EFT(t_i, P_{k})\);  
20 \(T = T - \{t_i\}\);  
21 \(R = R - \{P_{k}\}\);  
22 end while  
23 end  
25 \(\text{int } \text{allocateNumberOfClusters}(R, t_i)\)  
26 numberOfCluster=0;  
27 for each cluster \(C_{i}\) do  
28 if free processors in \(C_{i} \geq\) processors that \(t_i\) requires  
29 numberOfClusters++;  
30 return numberOfClusters;  
31 }
(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.

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 to 5. The techniques which can be applied in multi-processor task rearrangement include first fit, easy backfilling [16] and conservative backfilling [16]. 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 approach is more aggressive and allows tasks to skip ahead provided they do not delay the job at the head of the queue [16]. Lines 25 to 31 show a function \( allocateNumberOfClusters(R, \hat{t}_i) \). It returns the number of clusters that can accommodate the first task. If the function returns 1, the steps in lines 8 to 16 work for rule 1 described previously. If the function returns a number larger than 1, the steps in lines 17 to 22 work for rule 2.

4 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 makespan, which calculates the ratio of a workflow’s makespan over the best possible schedule length in a given environment. The performance is called Schedule Length Ratio (SLR) and defined by \( SLR = \frac{\text{makespan}}{\text{CPL}} \) where CPL represents the Critical Path Length of a workflow. SLR is not sensitive to the size of a workflow.
- **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, the higher win value leads to the 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
Fig. 3. The difference between RANK_HYBD, Fairness_Dynamic and OWM

components, workflow scheduling, task scheduling and allocation approaches. Figure 3 describes these three approaches according to the three components. **RANK_HYBD** [17] is shown in figure 3(a). The Fairness approach (F2) in [19] is a static algorithm and can not 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 as **Fairness_Dynamic** in figure 3(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 Poisson distribution. Each experiment involves 20 runs, and each run has 100 unique DAGs in a grid environment that contains 3 clusters each containing 30~50 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 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 from the set \{0.1, 0.5\}, and for communication-intensive workflows, CCR is randomly selected from the set \{1.5, 2.0\}.
Figures 4, 5 and 6 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 Fairness_Dynamic by 26% and 49%, and outperforms RANK_HYBD by 13% and 20% for average makespan and average SLR respectively, as shown in figures 4 and 5. Fairness_Dynamic has pool 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 figure 6. From users’ perspective, it means 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 come in after another one finishes. In real environments, most high-performance centers are overloaded, therefore OWM can outperform others in such environments.

![Fig. 4. Results of different mean arrival intervals for average makespan](image)

![Fig. 5. Results of different mean arrival intervals for average SLR](image)
5 Conclusion

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.

References