2012 10th IEEE International Symposium on Parallel and Distributed Processing with Applications

Fairness resource sharing for dynamic workflow scheduling on Heterogeneous Systems

Hamid Arabnejad, Jorge Barbosa
LIACC, Departamento de Engenharia Informática
Faculdade de Engenharia, Universidade do Porto
Rua Dr. Roberto Frias, 4200-465 Porto, Portugal
hamid.arabnejad@fe.up.pt, jbarbosa@fe.up.pt

***Abstract*—For most Heterogeneous Computing Systems (HCS) the completion time of an application is the most important requirement. Many applications are represented by a workflow that is therefore schedule in a HCS system. Recently, researchers have proposed algorithms for concurrent workflow scheduling in order to improve the execution time of several applications in a HCS system. Although, most of these algorithms were designed for static scheduling, that is all application must be submitted at the same time, there are a few algorithms, such as OWM (online workflow Management) and RANK_HYBD, that were presented for dealing with dynamic application scheduling. In this paper, we present a new algorithm for dynamic application scheduling. The algorithm focus on the Quality of Service (QoS) experienced by each application (or user). It reduces the waiting and execution times of each individual workflow, unlike other algorithms that give privilege to average completion time of all workflows. The simulation results show that the proposed approach significantly outperforms the other algorithms in terms of individual response time.**

I. Introduction

Heterogeneous Computing Systems (HCS) are characterized by having a variety of different types of computational units and are widely used for executing parallel applications, especially scientific workflows. The workflow consist of many tasks with logical or data dependencies that can be dispatched to different computation nodes in the HCS. To achieve efficient execution of a workflow and minimize the turnaround time, we need an effective scheduling strategy that decides when and which resource must execute the tasks of the workflow. When scheduling multiple independent workflows that represent user jobs and, consequently, are submitted at different instants of time, the common definition of makespan needs to be extended in order to account the waiting time as well as the execution time of a given workflow, contrary to the makespan definition for single workflow scheduling [7]. The metric to evaluate a dynamic scheduler of independent workflows has to represent the individual makespan instead of a global measure for the set of workflows, in order to measure the Quality of Service experienced by the users which is related to the finish time of each user application.

A popular representation of a workflow application is the Directed Acyclic Graph (DAG) in which nodes represent individual application tasks and the directed edges represent inter-task data dependencies.

The DAG scheduling problem has been shown to be NP-complete [3]. DAG scheduling is mainly divided in two major categories, namely Static Scheduling and Dynamic Scheduling. Most of the scheduling algorithms in static category are restricted to single DAG scheduling. In [2] the authors compared 20 scheduling heuristics for single DAG scheduling. For static scheduling of multiple DAGs there were proposed some algorithms such as in [4] for homogeneous non-parallel task graphs with the aim of increasing system efficiency, [13] for heterogeneous clusters and non-parallel task graphs and [8] for multi-clusters and parallel task graphs. In [13] the authors proposed an algorithm for scheduling several DAGs at the same time where the aim was to achieve fairness in the resource sharing, defined by the slowdown each DAG experiences as a results of competing for resources. In [8] the authors proposed several strategies of sharing the resources based on the proportional share. They defined a proportional share based on critical path, width and work of each DAG. They also proposed a weighted proportional share that represent a better tradeoff between fairness resource sharing and makespan reduction of the DAGs. Both works differ from what is proposed here due to their static approach and due to the objective functions considered. Here we consider a dynamic scheduling and we focus on the response time the system gives to each application.

For dynamic scheduling multiple parallel task graphs on a heterogeneous system, it was proposed an algorithm in [1] that minimizes the overall makespan, that is the finish time of all DAGs. In [5] and [12] there were proposed two algorithms namely OWM and RANK_HYBD for dynamic scheduling of multiple workflows. Both algorithms were proposed for the same context of the algorithm proposed here.

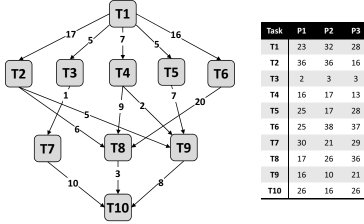
In this paper, we propose a new algorithm called the Fairness Dynamic Workflow Scheduling (FDWS) for scheduling dynamically workflow applications in a heterogeneous system. The remainder of this paper is organized as follows: section II describes the workflow representation; section III discuses related work; section IV presents the FDWS algorithm; section V presents the experimental results and discussion and section VI concludes the paper.

978-0-7695-4701-5/12 $26.00 © 2012 IEEE
DOI 10.1109/ISPA.2012.94

IEEE computer society

II. Workflow Representation

A workflow application can be represented by a Directed Acyclic Graph (DAG), $G(V, E, P, W)$ as shown in Figure 1, where V is the set of v tasks and E is the set of e communication edges between tasks. Each $e(i, j) \in E$ represents the task-dependency constraint such that task n_i should complete its execution before task n_j can be started. P is the set of p heterogeneous processors available in the system. W is a $v \times p$ computation cost matrix, where v is the number of tasks and p is the number of processors in the system. $w_{i,j}$ gives the estimate time to execute task v_i on machine p_j. The mean execution time of task n_i can be calculated by $\overline{w_i} = (\sum_{j \in P} w_{i,j})/p$. Each edge $e(i, j) \in E$ is associated with a non-negative weight $c_{i,j}$ representing the communication cost between the tasks n_i and n_j. Once this value could be computed only after defining where tasks i and j will be executed, it is common to compute the average communication costs to label the edges [11]. The average communication cost $\overline{c_{i,j}}$ of an edge $e(i, j)$ can be calculated by $\overline{c_{i,j}} = \overline{L} + \frac{data_{i,j}}{\overline{B}}$ where $\overline{L}$ is the average latency time of all processors and $\overline{B}$ is the average bandwidth of all links connecting the set of P processors. $data_{i,j}$ is the amount of data elements that task i needs to send to task j. Note that, when task i and j are assigned to the same processor, the real communication cost is considered to be zero because it is negligible compared to interprocessor communication costs.



Task	P1	P2	P3
T1	23	32	28
T2	36	36	16
T3	2	3	3
T4	16	17	13
T5	25	17	28
T6	25	38	37
T7	30	21	29
T8	17	26	36
T9	16	10	21
T10	26	16	26

Fig. 1: Application model and computation time matrix of the tasks in each processor

Next, we present some of the common attributes used in task scheduling, that we will refer in the following sections.

- $\mathbf{pred(n_i)}$: denotes the set of immediate predecessors of task n_i in a given DAG. A task with no predecessors is called an $entry$ task, n_{entry}. If a DAG has multiple entry tasks, a dummy entry task with zero weight and zero communication edges is added to the graph.
- $\mathbf{succ(n_i)}$: denotes the set of immediate successors of task n_i. A task with no successors is called an $exit$ task, n_{exit}. Like the entry task, if a DAG has multiple exit tasks, a dummy exit task with zero weight and zero communication edges from current multiple exit tasks to this dummy node is added.
- $\mathbf{makespan}$: it is the finish time of the exit task in the scheduled DAG, and is defined by $makespan = AFT(n_{exit})$ where $AFT(n_{exit})$ denotes the Actual Finish Time of the exit task.
- $\mathbf{Critical\ Path(CP)}$: the CP of a DAG is the longest path from n_{entry} to n_{exit} in the graph. The length of this path $|CP|$ is the sum of the computation costs of the tasks and intertask communication costs along the path. The $|CP|$ value of a DAG is the lower bound of the makespan. In this paper, the makespan includes de execution and waiting time spent in the system.
- $\mathbf{EST(n_i, p_j)}$: denotes the *Earliest Start Time* of a node n_i on a processor p_j.
- $\mathbf{EFT(n_i, p_j)}$: denotes the *Earliest Finish Time* of a node n_i on a processor p_j.

III. Related Work

In the past years, most of research on DAG scheduling were restricted to a single DAG. Only a few scheduling algorithms work on more than one DAG at a time.

Zhao and Sakellariou [13] presented two approaches based on fairness strategy for multi DAGs scheduling. The fairness is defined on the basis of slowdown that each DAG would experience (the slowdown is the difference in the expected execution time for the same DAG when scheduled together with other workflows and when scheduled alone). They proposed two algorithms, one fairness policy based on finish time and another fairness policy based on current time. Both algorithms, at first, scheduled each DAG on all processors with the static scheduling (like HEFT [11] or Hybrid.BMCT [9]) as the pivot scheduling algorithm, save its schedule assignment and keep its makespan as the slowdown value of the DAG. Next, sort all DAGs in descending order of their slowdown in the list. Then until there are unfinished DAGs into the list, the algorithm selects the first DAG with highest slowdown and then selects the first ready task that has not been scheduled on the DAG. The key idea is to evaluate the slowdown value of each DAG after scheduling a task and make a decision on which DAG should be selected to schedule the next task. The difference between the two fairness based algorithms proposed is that the Fairness Policy based on Finish Time, calculates the slowdown value of *only* the selected DAG, whereas in the Fairness Policy based on Current Time, the slowdown value is recalculated for *every* DAG. But these two algorithms are designed to schedule multiple workflow applications that are known at the same time (off-line scheduling). Here we consider the scheduling of dynamic workflows, meaning that they arrive at different instants, where the age and remaining time of each concurrent DAG is considered by the scheduler (on-line scheduling).

Z. Yu and W. Shi in [12] proposed a planner-guided strategy (called RANK_HYBD algorithm) to deal with dynamic workflow scheduling of applications that are submitted by different users at different instants of time. The RANK_HYBD algorithm ranks all tasks using $rank_u$ priority measure [11]. In each step, the algorithm reads all ready tasks from all DAGs and selects the next task to schedule based on their rank. If the ready tasks belong to different DAGs, the algorithm selects the task with lowest rank and if they belong to the same DAG, the task with highest rank is selected. With this strategy, The RANK_HYBD algorithm allows the DAG with lowest rank

(lower makespan) to be scheduled first to reduce the waiting time of the DAG in the system. But this strategy does not achieve good fairness because it always like to finish first lower DAGs in the system and postpones the higher DAGs. For instance, if a longer DAG is being executed and several lower DAGs are submitted to the system, the scheduler postpones the execution of the longer DAG to give priority to the smaller ones.

Hsu, Huang and Wang in [5] proposed Online Workflow Management (OWM) for scheduling multiple online workflows. In OWM algorithm, unlike in the RANK_HYBD that puts all ready tasks from each DAG into the ready list, it selects only a single ready task from each DAG with highest rank into the ready list. Then until there are unfinished DAGs on the system, the OWM algorithm selects the task with highest priority from ready list. Then it calculates the earliest finish time (EFT) for the selected task on each processor and selects the processor with minimum earliest finish time. If the selected processor is free at that time, the OWM algorithm assigns the selected task to the selected processor otherwise keeps the selected task in the ready list in order to be scheduling later. In their results, The OWM algorithm has better performance than RANK_HYBD [12] and Fairness_Dynamic (modified version of fairness algorithm [13]) in handling online workflows. The results of different mean arrival intervals according to different performance metrics show that the OWM algorithm outperforms Fairness_Dynamic by 26% and 49%, and outperforms RANK_HYBD by 13% and 20% for average makespan and average SLR (defined by eq. 5), respectively. As the RANK_HYBD algorithm, OWM uses a fairness strategy but, instead of scheduling smaller DAGs first, it selects and schedules tasks from the longer DAGs first. OWM has better strategy by filling the ready list with one task from each DAG so that it gives to all DAGs the chance to be selected in current time for scheduling. In their simulation environment, the number of processors is always near to the number of workflows so that in the most cases the scheduler has suitable number of processors to schedule the ready tasks. This choice does not expose a fragility of the algorithm that occurs when the number of DAGs is significant in relation to the number of processors or, the same is to say, for heavier loaded systems. Another problem with the OWM algorithm is in the processor selection phase where if the processor with earliest finish time for the selected task is not free, the algorithm postpones that task and keeps it in the ready list. If the system is heavy loaded, it is possible that in the next task selection, it is postponed again because meanwhile it may arrive new tasks with highest rank.

Both algorithms, RANK_HYBD and OWM, present results in terms of average makespan. This metric combines in the same way long and short DAGs and do not allow to infer the average waiting time spent by the DAGs individually. In this paper we propose new strategies in both aspects of selecting tasks from ready list and in the processor assignment in order to reduce the individual completion time of the DAGs, that is the total time between the submission and the completion time which includes execution and waiting time. We use the metric Schedule Length Ratio (SLR) that is a normalized measure of the completion time and that is more appropriated to conclude about the individual performance experienced by each user.

IV. Fairness Dynamic Workflow Scheduling

This section presents the Fairness Dynamic Workflow Scheduling (FDWS) algorithm. Figure 2 shows the structure of the FDWS algorithm. It comprises four main components: (1) Submit application, (2) Workflow pool, (3) Selected Tasks and (4) Processor allocation.

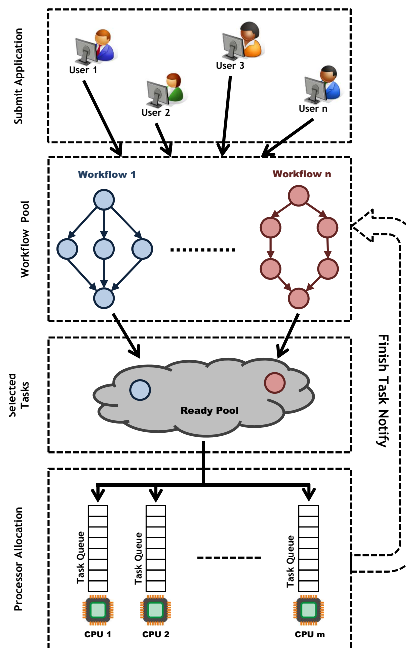


Fig. 2: Fairness Dynamic Workflow Scheduling (FDWS) System

Next we describe all these parts in detail:

- **Submit application:** users can submit their application at any time in the system.
- **Workflow pool:** after users submitted their applications they enter the workflow pool (each application can be represented by a DAG). At each scheduling time, this component finds all ready tasks of each DAG. The RANK_HYBD algorithm adds all ready tasks into the ready pool (or list) and the OWM algorithm adds only one task with highest priority from each DAG into the ready pool. Considering all ready tasks from each DAG leads to a unbiased preference for longer DAGs and the consequent postponing of smaller DAGs resulting higher SLR and unfair processor sharing. In FDWS algorithm, we add only a single ready task with highest priority from each DAG to ready tasks pool like as OWM. For assign the priority to tasks in the DAG, we used the upward rank

[11]. $rank_u$ represents the length of the longest path from task n_i to the exit node, including the computational cost of n_i and is given by equation 1.

$$rank_u(n_i) = \overline{w_i} + \max_{n_j \in succ(n_i)} \{\overline{c_{i,j}} + rank_u(n_j)\} \quad (1)$$

where $succ(n_i)$ is the set of immediate successors of task n_i, $\overline{c_{i,j}}$ is the average communication cost of $edge(i,j)$ and $\overline{w_i}$ is the average computation cost of task n_i. For the exit task $rank_u(n_{exit}) = 0$.

- **Selected tasks:** in this block we defined a different rank to select the task to be schedule from the ready tasks pool. To be selected to the pool we use $rank_u$ computed for each DAG individually. To select from the pool, we compute a new rank for task t_i belonging to DAG_j, defined by equation 2, and the task with highest $rank_r$ is selected.

$$rank_r(t_{i,j}) = \frac{1}{PRT\Big\{DAG_j\Big\}} \times \frac{1}{CPL\Big\{DAG_j\Big\}} \quad (2)$$

The metric $rank_r$ considers the Percentage of Remaining Task number (PRT) of the DAG and its Critical Path Length (CPL). The PRT value gives more priority to DAGs that are almost completed and only have few tasks to execute. This strategy is different from the Smallest Remaining Processing Time (SRPT) [6]. The SRPT algorithm, on each step, selects and schedules the application with the smallest remaining processing time. The remaining processing time is the time needed to execute all remaining tasks of the workflow. This is very different from our strategy. As an example, if we have two workflows with the same number of remaining tasks like it is shown in Figure3 (tasks with same name have the same computational time), using the SRPT strategy,

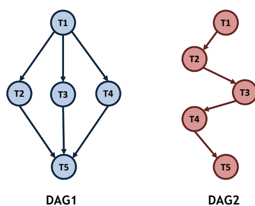


Fig. 3: Two sample workflows

there is no difference between these DAGs to select the next task, but attending to the response time, it is better to select a task from DAG1 because it has a lower expected finish time. With $rank_r$ we also consider the existing concurrency in each level by using the critical path (as it can be seen DAG1 has lower critical path length than DAG2 and therefore it would be schedule first). Also selecting only the smallest remaining time is not necessarily better; consider an application that is being executed and that have a few tasks remaining. With SRPT strategy, the application would be postponed if shorter DAGs arrive, which would increase the response time to the user. By using $rank_r$ we consider the percentage of remaining tasks to give the opportunity to the DAGs with fewer remaining tasks to be finished even if the work on those tasks is higher than the work on the new arrived DAGs.
Note that in RANK_HYDB and OWM, only the individual $rank_u$ is used for selecting tasks into the pool and to select one from the pool of ready tasks which leads to a scheduling decision that do not considers the DAG history in the workflow pool.

- **Processor allocation:** the FDWS algorithm uses the following strategy for assigning a task to a processor. All processors are tested and it selects the processor with lowest finish time for the current task, but if in current time the processor is busy, it puts the task into the processor task queue, so that the task is schedule at the first attempt. The finish time of a task on a processor also considers the queue list. In RANK_HYBD algorithm only the free resources are considerer at any given scheduling instant. But the processor which is busy right now, may execute the task with lower finish time. On the other hand, the OWM algorithm tests all available processors (both free and busy processors), then if the finish time of the task in the busy processor is less than in a free processor, the OWM algorithm postpones the assignment of the task to the next steps. Since the system is dynamic, it is possible that at any time a new application may arrive and the postponed task may have lower priority than the new ones and therefore being postponed again. This may lead to an excessive completion time for smaller DAGs.

The proposed FDWS (Fairness Dynamic Workflow Scheduling) algorithm is formalized in Algorithm 1.

V. Experimental Results and Discussion

This section presents performance comparison of the FDWS algorithm with OWM and RANK_HYBD algorithms. For this purpose, we divide this section into 4 parts where we describe first the DAG structure; then we present the environment scheduling system and hardware parameters; in the third part we present the comparison metrics and in the last part we present results and discuss the results.

A. DAG structure

To evaluate the relative performance of the algorithm, we considered randomly generated workflow (DAG) application graphs. For this purpose, we used a synthetic DAG generation program available at [10]. There are four parameters that define a DAG shape:

- ***n***: number of computation nodes in the DAG (i.e., application tasks);
- ***fat***: gives the width of the DAG by $e^{fat*log(n)}$, that is the maximum number of tasks that can be executed concurrently. A small value will lead to a thin DAG, like a chain, with a low task parallelism, while a large value

Algorithm 1 The FDWS algorithm

1: **while** *Workflow Pool* is NOT Empty **do**
2: **if** new workflow has arrived **then**
3: calculate $rank_u$ for all tasks of the new Workflow
4: Insert the Workflow into *Workflow Pool*
5: **end if**
6: $Ready_Pool$ ← ready tasks (one task with highest $rank_u$ from each DAG)
7: calculate $rank_r(t_{i,j})$ for each task t_i belonging to DAG_j in $Ready_Pool$
8: **while** $Ready_Pool \neq \phi$ AND $CPUs_{free} \neq 0$ **do**
9: T_{sel} ← the task with highest $rank_r$ from $Ready_Pool$
10: $EFT(T_{sel}, P_j)$ ← Earliest Finish Time of task T_{sel} on Processor P_j with considering of $Task\ Queue$ and using insertion-based policy
11: P_{sel} ← the processor with lowest EFT for task T_{sel}
12: **if** P_{sel} is free **then**
13: Assign Task T_{sel} to processor P_{sel}
14: **else**
15: add Task T_{sel} into $Task_Queue$ of the processor P_{sel}
16: **end if**
17: remove Task T_{sel} from $Ready_Pool$
18: **end while**
19: **end while**

induces a fat DAG, like a fork-join, with a high degree of parallelism;

- ***density***: denotes the number of edges between two levels of the DAG, with a low value leading to few edges and a large value leading to many edges;
- ***regularity***: the regularity denotes the uniformity of the number of tasks in each level. A low value means that levels contain very dissimilar numbers of tasks, while a high value means that all levels contain similar numbers of tasks;
- ***jump***: indicates that an edge can go from level l to level $l+jump$. A jump of 1 is an ordinary connection between two consecutive levels.

In this paper, we used the synthetic DAG generator only for making the DAG structure which includes the specific number of nodes and their dependencies. In our experiment, for random DAG generation, we consider $n = [10, 20, 30, 40, 50, 60]$, $jump = [1, 2, 4]$, $regularity = [0.2, 0.8]$, $fat = [0.1, 0.4, 0.8]$ and $density = [0.2, 0.8]$. With these parameters we have 216 different structure DAGs for our experiment. The DAG structure is presented apart from other simulation parameters because the structure of the workflows is dependent from users requests and independent from the hardware environment.

B. Environment system parameters

From the DAG structure obtained as explained above, we obtain computation and communication costs by using the following parameters:

- ***CCR*** (Communication to Computation Ratio): ratio of the sum of the edge weights to the node weights in a DAG;
- ***beta*** (Range percentage of computation costs on processors): it is the *heterogeneity factor* for processors speed. A higher value for β implies higher heterogeneity and very different computation costs among processors and a low value implies that the computation costs for a given task is almost equal among processors. The average computation cost of a task n_i in a given graph $\overline{w_i}$ is selected randomly from a uniform distribution with range $[0, 2 \times \overline{w_{DAG}}]$, where $\overline{w_{DAG}}$ is the average computation cost of the given graph (in our experiment $\overline{w_{DAG}} = 100$). The computation cost of each task n_i on each processor p_j is randomly set from the range of equation 3.

$$\overline{w_i} \times \left(1 - \frac{\beta}{2}\right) \leq w_{i,j} \leq \overline{w_i} \times \left(1 + \frac{\beta}{2}\right) \qquad (3)$$

In our experiment, for random DAG generation, we consider $CCR = [0.1, 0.8, 2, 5]$, $\beta = [0, 0.1, 0.5, 1, 2]$ and $Processors = [8, 16, 32]$.

For creating the simulation scenarios, we consider two additional parameters: number of workflows in each scenario, that are 30 and 50 respectively; and arrival interval value between workflows, that are set based on the Poisson distribution with mean value of 0, 50, 100 and 200 time units respectively. With these parameters (number of concurrent DAGs, arrival interval time, CCR, beta and CPU number) and considering 10 different workflows per combination, each experiment involves a test case of 4800 workflows.

C. Performance metrics

For evaluate and compare our algorithm with other approaches, we used the following metrics:

- **Overall makespan:** is the finish time of the last task to be executed in a set of workflows submitted to the system. It is calculated by equation 4. This metric gives the time required to complete all workflows in the scenario.

$$Overall\ makespan = \max_{t_i \in DAGs} \{AFT(t_i)\} \qquad (4)$$

- **Schedule Length Ratio (SLR):** we have DAGs with very different structure and execution time. The SLR normalizes the makespan of a given DAG to the lower bound and is given in equation 5.

$$SLR_{DAG_i} = \frac{makespan(DAG_i)}{\sum_{n_i \in CP_DAG_i} \min_{p_j \in P} (w_{(i,j)})} \qquad (5)$$

The denominator in the SLR equation is the minimum computation cost of the critical path tasks. As the numerator represents the total time spent by a DAG in the system, waiting and execution time, a low value, such as 1, means that the response time was the lowest possible for that DAG in the target system. On the other hand, a higher value means that, due to the concurrent DAG

scheduling, the DAG required more time to complete. In this sense, the SLR is a metric of Quality of Service experienced by the users. We used the SLR value for each Scenario ($SLR_{Scenario}$) and it is equal to average SLR values of all DAGs in each Scenario.

- **Win(%):** this metric represents the percentage of the number of occurrences of better results that is the percentage of DAGs in each scenario that have the shortest makespan when applying the FDWS algorithm.

D. Results and discussion

In this section, we compare FDWS with RANK_HYBD and OWM algorithms in terms of SLR, Overall Makespan, Average Makespan and percentage of Wins. We present results for a set of 50 and 30 DAGs that arrive with a time interval mean value that ranges from zero (all DAGs available at time zero) to 200 time units. We consider 3 sets of processors with 8, 16 and 32, in order to analyse the behaviour of the algorithms concerning the system load. The maximum load configuration is observed for 8 processors, 50 DAGs and a mean arrival time interval of zero.

Figure 4 shows the $SLR_{scenario}$ obtained with the 3 algorithms. We can see that FDWS obtains significant performance improvement over RANK_HYBD and OWM for all arrival time intervals. It keeps a stable relative improvement above 35%, for all cases, and being of 40% for the most loaded scenario. The SLR, as mentioned before, is in fact a metric that reflects the Quality of Service (QoS) experienced by the users, and therefore, we can conclude that FDWS improves significantly the QoS of the system.

The results shown in [5] that compare OWM with RANK_HYBD show close results for both algorithms although OWM performs slightly better. In our experiment this difference is not obvious and in some cases RANK_HYBD performs better. This is mainly because in [5] the authors considered that they have 100 DAGs for 90 to 150 processors that is, the number of DAGs is in general less than the number of processors available. Consequently, the concurrency is lower than in our experiment.

Table I shows the improvement of FDWS over the two other concerning Overall Makespan. We can see that OWM achieves a better result showing that OWM obtains a total time to process all DAGs shorter than the others.

		Arrival Interval			
DAGs	Algorithm	0	50	100	200
50	OWM	-18.10%	-14.13%	-10.43%	-5.00%
	RANK_HYBD	20.55%	16.61%	13.75%	8.60%
30	OWM	-22.58%	-18.01%	-13.510%	-7.28%
	RANK_HYBD	15.77%	13.42%	10.81%	6.84%

TABLE I: Overall Makespan improvement of FDWS over the RANK_HYBD and OWM algorithm; a negative value means that FDWS as a worse result

Figure 5 shows boxplots for $SLR_{scenario}$ as a function of different hardware parameters such as CCR, heterogeneity

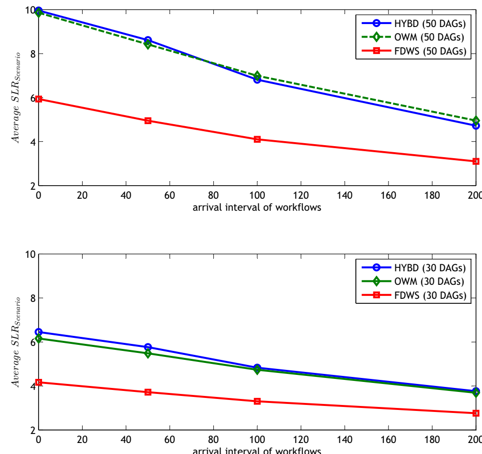


Fig. 4: Results of different mean arrival times for average SLR for 30 and 50 concurrent DAGs

factor and CPU number. The mean value is also shown by a individual stronger line. We can see that FDWS has statistically better behaviour for all CCRs, heterogeneity factor and CPU number. Lower mean and median values and also less dispersion.

Figure 6 shows results of Average Makespan, a non normalised measure, that is defined by the average Makespan of all DAGs in the scenario. These results show similar performance as for SLR that is the normalized makespan.

Figure 7 shows the percentage of wins and, as it can be seen, FDWS produces in most of the times better schedules for the DAGs. Only for a low loaded scenario with 30 DAGs and a mean arrival interval of 200, it is outperformed by OWM.

FDWS is always better than RANK_HYBD and in most of the cases it is better than OWM, obtaining similar results only for low loaded scenarios.

VI. Conclusions

Most workflow scheduling algorithms focused on single workflows and there are only a few works on multiple workflow scheduling. In this paper, we presented a new algorithm called FDWS (Fairness Dynamic Workflow Scheduling) and compared it with two recent algorithms, namely OWM [5] and RANK_HYBD [12] algorithms that deal with multiple workflow scheduling in dynamic situations. Based on our experiments, FDWS has better improvement in terms of SLR, Win(%) and Average Makespan, showing better Quality of Service characteristics. The drawback of this feature is to obtain a longer Overall execution time. As future work, we intend to find if the SLR improvement is the main reason to increase the Overall Makespan or if both characteristics can be reduced.

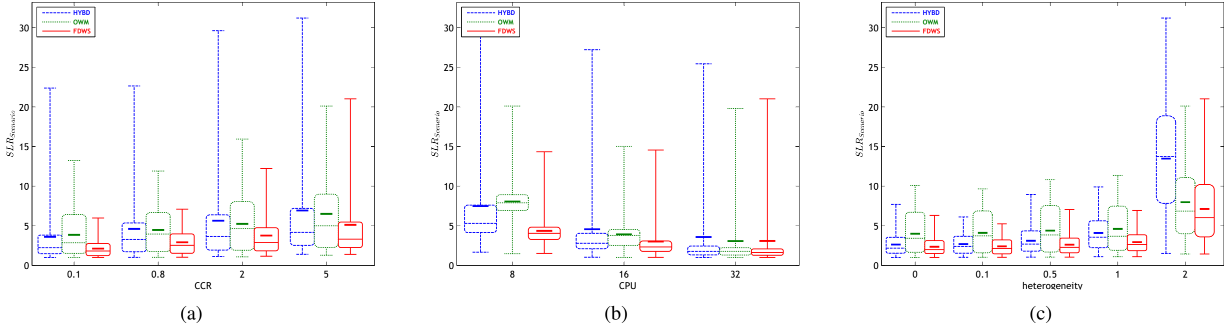


Fig. 5: Boxplots of SLR value for each scenario with respect to the (a) CCR, (b) CPU and (c) heterogeneity factor for random graphs

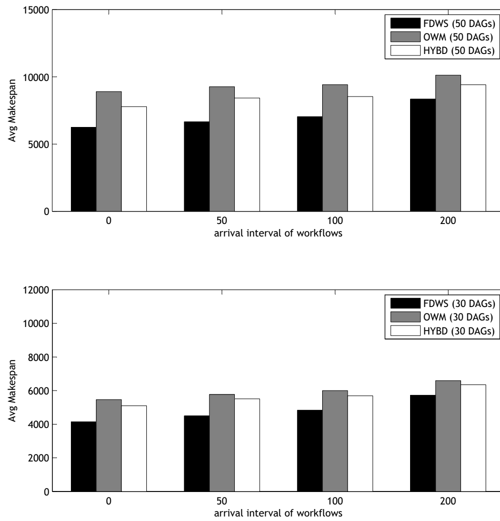


Fig. 6: Results of different mean arrival times for Average Makespan

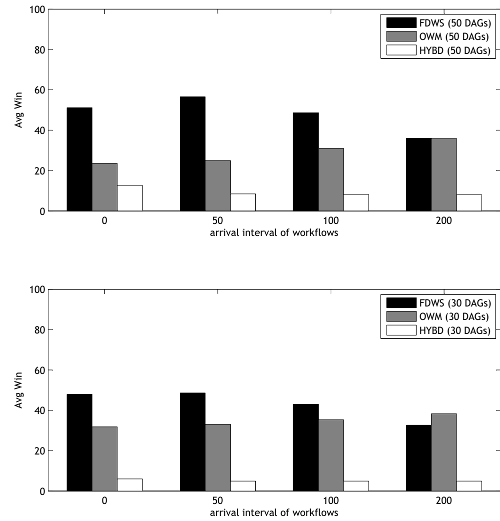


Fig. 7: Results of different mean arrival times for Win(%)

ACKNOWLEDGEMENTS

We would like to thank the support given by Cost Action IC0805 Open Network for High-Performance Computing on Complex Environments, Working Group 3: Algorithms and tools for mapping and executing applications onto distributed and heterogeneous systems.

REFERENCES

[1] J.G. Barbosa and B. Moreira. Dynamic scheduling of a batch of parallel task jobs on heterogeneous clusters. *Parallel Computing*, 37(8):428–438, 2011.

[2] L.C. Canon, E. Jeannot, R. Sakellariou, and W. Zheng. Comparative evaluation of the robustness of dag scheduling heuristics. In *Grid Computing*, pages 73–84. Springer, 2008.

[3] E.G. Coffman and J.L. Bruno. *Computer and job-shop scheduling theory*. John Wiley & Sons, 1976.

[4] U. Hönig and W. Schiffmann. A meta-algorithm for scheduling multiple dags in homogeneous system environments. In *Parallel and Distributed Computing and Systems*. ACTA Press, 2006.

[5] C.C. Hsu, K.C. Huang, and F.J. Wang. Online scheduling of workflow applications in grid environments. *Future Generation Computer Systems*, 27(6):860–870, 2011.

[6] D. Karger, C. Stein, and J. Wein. Scheduling algorithms. *CRC Handbook of Computer Science*, 1997.

[7] Y.K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, 1999.

[8] T. N'takpé and F. Suter. Concurrent scheduling of parallel task graphs on multi-clusters using constrained resource allocations. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[9] R. Sakellariou and H. Zhao. A hybrid heuristic for dag scheduling on heterogeneous systems. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 111. IEEE, 2004.

[10] F. Suter. Synthetic dag generation program. http://www.loria.fr/~suter/dags.html, 2010.

[11] H. Topcuoglu, S. Hariri, and M. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260–274, 2002.

[12] Z. Yu and W. Shi. A planner-guided scheduling strategy for multiple workflow applications. In *Parallel Processing-Workshops, 2008. ICPP-W'08. International Conference on*, pages 1–8. IEEE, 2008.

[13] H. Zhao and R. Sakellariou. Scheduling multiple DAGs onto heterogeneous systems. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 14–pp. IEEE, 2006.