CHAPTER 1

FAIR RESOURCE SHARING FOR DYNAMIC SCHEDULING OF WORKFLOWS ON HETEROGENEOUS SYSTEMS

HAMID ARABNEJAD[1], JORGE G. BARBOSA[1], FRÉDÉRIC SUTER[2]

[1]LIACC, Departamento de Engenharia Informática, Faculdade de Engenharia, Universidade do Porto, Portugal

[2] IN2P3 Computing Center, CNRS, IN2P3, Lyon-Villeurbanne, France

1.1 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. A workflow consists in many tasks with logical or data dependencies that can be dispatched to different compute nodes in the HCS. To achieve an 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 to account for the waiting time as well as the execution time of a given workflow. The metric to evaluate a dynamic scheduler of independent workflows has to represent the individual execution time instead of a global measure for the set of workflows, in order to reflect the Quality of Service experienced by the users which is related to the response time of each user application.

The efficient usage of any computing system depends on how well the workload is mapped to the processing units. The workload that is considered here consists on workflow applications that are composed by a collection of several interacting components, or tasks, that need to be executed in a certain order for successful execution of the application as a whole. The operation of scheduling, that consist on defining a mapping and a order of task execution, have been addressed mainly for single workflow scheduling, that is a schedule is generated for a workflow and a specific number of processors that are used exclusively throughout the workflow execution. When execution more than one workflow, they are considered as independent applications that execute on independent subsets of processors. However, due to tasks precedences, not all processors are fully used when executing a workflow leading to lower rates of efficiency. A way to improve the system efficiency is to consider concurrent workflows, meaning that processors are shared among workflows. In this context there is no exclusive use of processors by an workflow and throughout its execution, it can use any processor available in the system. The processors are not used exclusively by a workflow but only one task runs in a processor at each time.

Here, we present a review on concurrent workflow scheduling and an extended comparison of dynamic workflow scheduling algorithms for randomly generated graphs. Next, we introduce the concept of an application, the heterogeneous system, the performance metrics that are commonly used in workflow scheduling and a metric for accounting for the total execution time is introduced.

1.1.1 Application model

A typical scientific workflow application can be represented as a Directed Acyclic Graph (DAG). In a DAG, nodes represent tasks and the directed edges represent execution dependencies as well as the amount of communication between nodes.

Specifically, a workflow is modeled by a DAG $G = (V, E)$, where $V = \{n_j, j = 1..v\}$ represents the set of v tasks (or jobs) to be executed, E is a set of e weighted directed edges and represents communication requirements between tasks. Each $edge(i, j) \in E$ represents the precedence constraint that task n_j can not start without successful completion of task n_i. $Data$ is a $v \times v$ matrix of communication data, where $data_{i,j}$ is the amount of data required to be transferred from task n_i to task n_j.

The target computing environment consists of a set P of p heterogeneous processors organized in a fully connected topology in which all inter-processor communications are assumed to be performed without contention, as explained in Section1.1.2.

The data transfer rates between the processors, i.e., bandwidth, are stored in a matrix B of size $p \times p$. The communication startup costs of processors, i.e., latency, are given in a p-dimensional vector L. The communication cost of the $edge(i, j)$, which is for transferring data from task n_i (executed on processor p_m) to task n_j

(executed on processor p_n) is defined as:

$$c_{i,j} = L_m + \frac{data_{i,j}}{B_{m,n}} \tag{1.1}$$

When both tasks n_i and n_j are scheduled on the same processor, $c_{i,j} = 0$. Typically, the communication cost simplifies by introducing an average communication cost of an $edge(i,j)$ as defined by:

$$\overline{c_{i,j}} = \overline{L} + \frac{data_{i,j}}{\overline{B}} \tag{1.2}$$

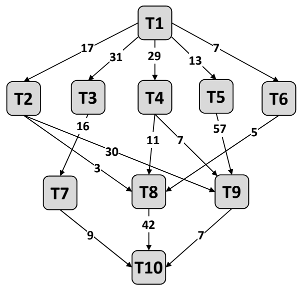
where $\overline{B}$ is the average bandwidth among all the processor pairs and $\overline{L}$ is the average latency. This simplification is commonly considered to label the edges of the graph to allow the computation of tasks priority rank before assigning tasks to processors [17].

Also, each task may have a different execution time on each processor, W is a $v \times p$ matrix of computation costs in which each $w_{i,j}$ represents the execution time to complete task n_i on processor p_j. The average execution cost of task n_i is defined as:

$$\overline{w_i} = \sum_{j=1}^{p} \frac{w_{i,j}}{p} \tag{1.3}$$

As for the communication costs, the average execution time is commonly used to compute a priority rank for the tasks.

A DAG example is shown in Figure 1.1 which models a DAG and a target system with three processors and corresponding communication and computation costs. In the figure 1.1, the weight of each edge represents its average communication cost and the figures in the table represent the computation time of each task in each of the three processors. This model represents a general heterogeneous system.



Task	P1	P2	P3
T1	22	21	36
T2	22	18	18
T3	32	27	43
T4	7	10	4
T5	29	27	35
T6	26	17	24
T7	14	25	30
T8	29	23	36
T9	15	21	8
T10	13	16	33

Figure 1.1 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 to in the following sections.

- $pred(n_i)$: 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 nodes, a dummy entry node with zero weight and zero communication edges is added to the graph.

- $succ(n_i)$: 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 node, if a DAG has multiple exit nodes, a dummy exit node with zero weight and zero communication edges from current multiple exit nodes to this dummy node is added.

- $makespan$ or $Schedule\ Length$: it is the elapsed time from the beginning of the execution of the entry node to the finish time of the exit node in the scheduled DAG, and is defined by:

$$makespan = AFT(n_{exit}) - AST(n_{entry}) \tag{1.4}$$

  where $AFT(n_{exit})$ denotes the *Actual Finish Time* of the exit node and $AST(n_{entry})$ denotes the *Actual Start Time* of the entry node.

- $level(n_i)$: the level of task n_i is an integer value representing the maximum number of edges composing the paths from the entry node to n_i. For the entry node the level is $level(n_{entry}) = 1$ and for other tasks it is given by:

$$level(n_i) = \max_{q \in pred(n_i)} \{level(q)\} + 1 \tag{1.5}$$

- $Critical\ Path(CP)$: the CP of a DAG is the longest path from the $entry$ node to the $exit$ node in the graph. The length of this path $|CP|$ is the sum of the computation costs of the nodes and inter-node communication costs along the path. The $|CP|$ value of a DAG is the lower bound of the schedule length.

- $EST(n_i, p_j)$: denotes the *Earliest Start Time* of a node n_i on a processor p_j and is defined as:

$$EST(n_i, p_j) = \max\left\{T_{Available}(p_j), \max_{n_m \in pred(n_i)} \left\{AFT(n_m) + c_{m,i}\right\}\right\} \tag{1.6}$$

  where $T_{Available}(p_j)$ is the earliest time at which processor p_j is ready. The inner max block in the EST equation is the time at which all data needed by n_i has arrived at the processor p_j. For the entry task $EST\big(n_{entry}, p_j\big) = \max\{T_s, T_{Available}(p_j)\}$, where T_s is the submission time of the DAG in the system.

- $EFT(n_i, p_j)$: denotes the *Earliest Finish Time* of a node n_i on a processor p_j and is defined as:

$$EFT(n_i, p_j) = EST(n_i, p_j) + w_{i,j} \tag{1.7}$$

  which is the *Earliest Start Time* of a node n_i on a processor p_j plus the execution time of task n_i on processor p_j.

The *objective function* of the scheduling problem from the user point of view, a single workflow, is to determine an assignment of tasks of this workflow to processors such that the *Schedule Length* is minimized. After all nodes in the workflow are scheduled, the schedule length will be the makespan as expressed by Equation 1.4.

1.1.2 System model

Typically for executing complex workflows, one uses a high performance cluster or a grid platform. As defined in [2], a cluster is a type of parallel or distributed processing system which consists in a collection of interconnected stand-alone computers (nodes) working together as a single, integrated computing resource. A compute node can be a single or multiprocessor system (PCs, workstations, or SMPs) with memory, I/O facilities, GPU devices and an operating system. A cluster generally refers to two or more computers (nodes) connected together. The nodes can exist in a single cabinet or be physically separated and connected via a LAN. Figure 1.2 illustrates the typical cluster architecture.

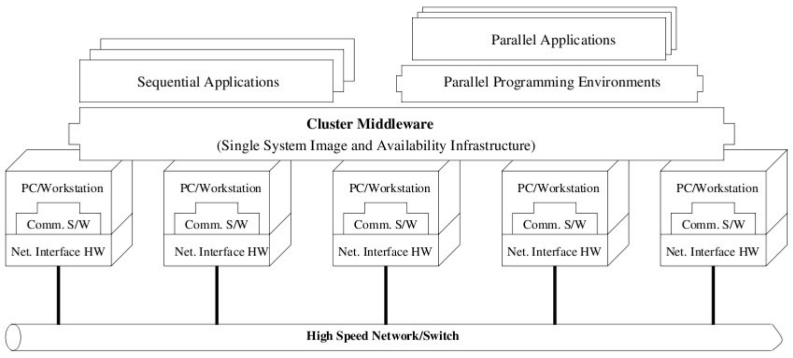

Parallel Applications
Sequential Applications
Parallel Programming Environments
Cluster Middleware
(Single System Image and Availability Infrastructure)
PC/Workstation
Comm. S/W
Net. Interface HW
High Speed Network/Switch

Figure 1.2 Conceptual cluster architecture

The algorithms for concurrent workflow scheduling may be useful when there are a significant number of workflows compared to the computational nodes available, otherwise the workflows could use a set of processors exclusively without concurrency. Therefore, in the context of the experiments reported here, we consider a cluster formed by nodes of the same site, connect by a single bandwidth switched network. In a switched network, the execution of tasks and communications with other processors can be achieved for each processor simultaneously and without contention. These characteristics allows to simplify the computation of the communication costs in the DAG (Figure 1.1) by considering the average communication parameters.

The target system can be as simple as a set of devices (e.g., CPUs and GPUs) connected by a switched network that guarantees parallel communications between different pairs of devices. The machine is heterogeneous because CPUs can be from different generations and also because other very different devices such as GPUs can be included. Another common machine is the one that results from selecting processors from several clusters from the same site. Although a cluster is homogeneous, the set of processors selected forms a heterogeneous machine. The processor latency can differ in a heterogeneous machine, but such differences are negligible. For low communication-to-computation ratios (CCRs), the communications are negligible; for higher CCRs, the predominant factor is the network bandwidth, and as mentioned above we consider that the bandwidth is the same throughout the entire network. Additionally, the execution of any task is considered nonpreemptive.

1.1.3 Performance metrics

Performance metrics are used to evaluate the effectiveness of the scheduling strategy. As some metrics may conflict with each other, any system design can not accommodate all metrics at same time and a balance according to final goal have to be found. In what follows, some of well-know metrics are chosen and presented.

Makespan
Also referred to as schedule length, it is the time difference between application start time and its completion. Most of scheduling algorithms use this metric to show and evaluate their results and their solutions over other algorithms. Smaller makespan means better performance.

Turnaround Time
Turnaround Time is the difference between submission and final completion of an application. Different than *makespan*, turnaround time includes the time the workflow application spends waiting to get started. It is used to measure the performance and service satisfaction from a user perspective.

Turnaround Time Ratio
The TTR metric measures the additional time each workflow spends in the system to be executed in relation to the minimum Makespan obtained for that workflow. The TTR for a workflow is defined as:

$$\text{TTR} = \frac{\text{TurnaroundTime}}{\sum_{n_i \in CP_{MIN}} \min_{p_j \in P} (w_{(i,j)})} \tag{1.8}$$

The denominator in the TTR equation is the minimum computation cost of the tasks that compose the critical path, which is a lower bound of execution time for a workflow.

Normalized Turnaround Time
The NTT metric is obtained by the ratio of the minimum turnaround time and the actual turnaround time for a given workflow G and an algorithm a_i, as defined by the following equation:

$$\text{NTT}(G, a_i) = \frac{\min_{a_k \in A}\{\text{TurnaroundTime}(G, a_k)\}}{\text{TurnaroundTime}(G, a_i)} \tag{1.9}$$

where A is the set of algorithms under comparison and $a_i \in A$. This metric for an algorithm a_i gives the distance that its scheduling solutions are from the minimum TTR obtained for a given workflow G. NTT is distributed in the interval $[0, 1]$ and the algorithm with lower dispersion of NTT values, and close to 1, is the algorithm that generates more results closer to the minimum, i.e. the best algorithm.

Win(%)

The percentage of wins is used to compare the frequency of best results for Turnaround Time for the set of workflows under scheduling. An algorithm with higher percentage of wins means that it obtains better results from the user point of view, i.e., it obtains more frequently the shortest elapsed time from submission to completion of a user job. Note that the sum of this value for all algorithms may be higher than 100%; this is because when more than one algorithm wins, for a given workflow, it is accounted for all those winning algorithms.

1.2 Concurrent workflow scheduling

Recently, several algorithms have been proposed for concurrent workflow scheduling in order to improve the execution time of several applications in a HCS system. However, most of these algorithms were designed for off-line scheduling or static scheduling, that is all the applications are known at the same time. This approach although relevant, imposes limitations in the management of a dynamic system where users can submit jobs at any time. For this purpose there are a few algorithms that were designed to deal with dynamic application scheduling. Next, a review on off-line scheduling is presented followed by a review on on-line scheduling.

1.2.1 Off-line scheduling of concurrent workflows

Off-line scheduling is characterized by having the workflows available before the execution starts, also referred at compile time. After a schedule is produced and applied, no other workflow is considered. This approach although limited is applicable in many real world applications. For example, when a user has a set of nodes to run a set of workflows. In fact this methodology is applied by the most common resource management tools where a user requests a set of nodes to execute exclusively his/her jobs.

Several algorithms have been proposed for off-line scheduling where workflows compete for resources, where the aim is at ensuring a fair distribution of those resources and also trying to minimize the individual completion time of each workflow. Zhao and Sakellariou [21] presented two approaches based on a fairness strategy for concurrent workflow scheduling. The fairness is defined on the basis of the slowdown that each DAG would experience (the slowdown is the ratio of the expected execution times 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 a static scheduling (like HEFT [17] or Hybrid.BMCT [16]) as the pivot scheduling algorithm, save its schedule assignment and keep its makespan as the slowdown value of the DAG. Next, all workflows are sorted in descending order of their slowdown. Then, until there are unfinished workflows in the list, the algorithm selects the DAG with highest slowdown and then selects the first ready task that has not been scheduled in this DAG. The key idea is to evaluate the slowdown value of each DAG after scheduling a task and take 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 the selected DAG only, whereas in the fairness policy based on Current Time, the slowdown value is recalculated for every DAG.

N'takpé and Suter [14] proposed several strategies based on the proportional sharing of resources. This proportional sharing was defined based on the critical path length, width or work of each workflow. It was also proposed a weighted proportional sharing that represents a better tradeoff between fair resource sharing and makespan reduction of the workflows. The strategies were proposed and applied to mixed parallel applications, where each task can be is executed on more than one processor. The proportional sharing based on the work need to execute a workflow led the shortest schedules in average but was also the least fair with regard to resource usage, i.e., the variance of the slowdowns experienced by the workflows was the highest.

Bittencourt and Madeira [4] proposed a Path Clustering heuristic that combines the clustering scheduling technique to generate groups (clusters) of tasks and the list scheduling technique to select tasks and processors. Based on this methodology they propose and compare four algorithms, that are: a) sequential scheduling, where workflows are scheduled one after another; b) gap search algorithm, similar to the former but it searches for spaces between already scheduled tasks; c) interleave algorithm, where pieces of each workflow are scheduled in turns; and d) group workflows, where the workflows are joined to form a single one and then scheduled as a single one. The evaluation was made in terms of schedule length and fairness and conclude that interleaving the workflows leads to lower average makespan and higher fairness when multiple workflows share the same set of resources. This results although relevant, consider the average makespan which do not distinguish the impact of the delay on each workflow compared to exclusive execution.

Casanova *et al.* [7] evaluated extensively algorithms for the off-line scheduling of concurrent parallel task graphs, on a single homogeneous cluster. The graphs, or workflows, that have been submitted by different users share a set of resources and are ready to start their execution at the same time. The aim is at optimizing user-perceived notions of performance and fairness. The authors proposed three metrics to quantify the quality of a schedule, related to performance and fairness among the parallel task graphs.

Carbajal *et al.* [6] presented two workflow scheduling algorithms named MWGS4 (Multiple Workflow Grid Scheduling 4 stages) and MWGS2 (Multiple Workflow Grid Scheduling 2 stages) that consist of four and two stages: labeling, adaptive, allocation, prioritization and parallel machine scheduling. Both algorithms, MWGS4 and MWGS2, are classified as off-line strategies. Both schedule a set of available and ready jobs belonging to a batch of jobs. All jobs that arrive during a time interval will be processed in a batch and start to execute after the completion of the last batch of jobs. It was shown that the proposed strategies outperform other strategies in terms of mean critical path waiting time, and critical path slowdown.

1.2.2 On-line scheduling of concurrent workflows

On-line scheduling is characterized by having a dynamic behaviour where the workflows can be submitted by users at any time. When scheduling multiple independent workflows that represent user jobs and, consequently, are submitted at different instants of time, the completion time (or turnaround time) takes into account the waiting time as well as the execution time of a given workflow, extending the makespan definition for single workflow scheduling [11]. The metric to evaluate a dynamic scheduler of independent workflows has to represent the individual completion time instead of a global measure for the set of workflows, in order to measure the Quality of Service (QoS) experienced by the users that is related to the finish time of each user application.

Some algorithms have been proposed for on-line workflow scheduling, that are described briefly next, and three others that were proposed specifically to schedule concurrent workflows aiming at improving individual QoS. These algorithms, Online Workflow Management (OWM), Rank Hybrid (Rank_Hybd) and Fairness Dynamic Workflow Scheduling (FDWS), are detailed and compared in this section. The first two algorithms give privilege to the improvement of the average completion time of all workflows. On the contrary, FDWS focuses on the Quality of Service (QoS) experienced by each application (or user) by minimizing the waiting and execution times of each individual workflow.

Liu *et al.* [12] proposed the MMA(Min-Min Average) algorithm for efficiently schedule transaction-intensive grid workflows involving considerable communication overheads. The MMA algorithm is based on the popular Min-Min algorithm but uses a different strategy for transaction-intensive grid workflows with the capability of adapting to the change of network transmission speed automatically. Transaction-intensive workflows are multiple instances of one workflow and the aim is at optimizing the overall throughput, instead of the individual workflow performance. As Min-Min is one popular technique we consider, in our results, one implementation of Min-Min for concurrent workflow scheduling.

Xu *et al.* [19] proposed an algorithm for scheduling multiple workflows with multiple QoS constrains on the Cloud. The Multiple QoS Constrained Scheduling Strategy of Multiple Workflows(MQMW) minimizes the makespan and the cost of the resources and increases the success scheduling rate. The algorithm considers two objectives, time and cost, that can be adapted to the user requirements. MQMW was compared to Rank_Hybd and when time was the major QoS requirement, Rank_Hybd performed better. Here, we are considering time as the QoS requirement and therefore we consider Rank_Hybd in our results section.

Barbosa and Belmiro [3] proposed a dynamic algorithm to minimize the makespan of a batch of parallel task workflows with different arrival times. The algorithm was proposed for on-line scheduling but with the aim of minimizing a collective metric. This model is applied to real world applications such as video surveillance, image registration, etc, where the workflows are related and only the collective result is meaningful. This approach is different from the independent workflows execution that we consider in this work.

1.2.2.1 Rank Hybrid algorithm Yu and Shi in [20] proposed a planner-guided strategy, named Rank_Hybd algorithm, to deal with dynamic scheduling of workflow applications that are submitted by different users at different instants of time. The Rank_Hybd algorithm ranks all tasks using the $rank_u$ priority measure (Equation 1.10). 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. The Rank_Hybd heuristic is formalized in Algotithm 2.

Algorithm 1 CheckWorkflows()

```
1:  if a new workflow has arrived then
2:      calculate $rank_u$ for all tasks of the new workflow
3:  end if
4:  $Ready\_Pool$ ← Read all ready tasks from all DAGs
5:  $Resource\_free$ ← get all idle resources
6:  Assign $rank_u$ as task priority to each task in the $Ready\_Pool$
7:  $multiple$ ← number of different DAGs which have ready tasks in $Ready\_Pool$
8:  if $multiple$ == 1 then
9:      sort all tasks in $Ready\_Pool$ in descending order of priority
10: else
11:     sort all tasks in $Ready\_Pool$ in ascending order of priority
12: end if
13: return $Ready\_Pool$
```

Algorithm 2 The Rank Hybrid algorithm

```
1:  while there are not scheduled workflows do
2:      $Ready\_Pool$ ← CheckWorkflows()
3:      while $Ready\_Pool$ AND $Resource\_free$ are not empty do
4:          $task_{select}$ ← the first task in $Ready\_Pool$ list
5:          $resource_{select}$ ← the processor with lowest finish time among of all
            resources in $Resource\_free$ list
6:          Assign $task_{select}$ to $resource_{select}$
7:          remove $resource_{select}$ from $Resource\_free$ list
8:          remove $task_{select}$ from $Ready\_Pool$ list
9:      end while
10: end while
```

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 among workflows because it always give preference to finish smaller workflows first and postpones the bigger ones. For instance, if a longer workflow is being executed and several small workflows are submitted to the system, the scheduler postpones the execution of the longer DAG to give priority to the smaller ones.

1.2.2.2 On-line Workflow Management Hsu *et al.* [9] proposed the Online Workflow Management (OWM) algorithm for on-line scheduling of multiple workflows. Unlike the Rank_Hybd algorithm that puts all ready tasks from each DAG into the ready list, OWM selects only a single ready task from each DAG, that of highest rank ($rank_u$) into the ready list. Then until there are some unfinished DAGs on the system, the OWM algorithm selects the task with highest priority from the ready list. Then it calculates the earliest finish time (EFT) for the selected task on each processor and selects the processor that leads to the smallest EFT. 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 scheduled later. The OWM heuristic is formalized in Algorithm 3.

Algorithm 3 The OWM algorithm

```
1: while there are not scheduled workflows do
2:     $Ready\_Pool$ ← CheckWorkflows()
3:     while $Ready\_Pool$ AND $Resource\_free$ are not empty do
4:         $task_{select}$ ← the first task in $Ready\_Pool$ list
5:         $resource_{select}$ ← the processor with lowest finish time among of all
           available resources
6:         if $FreeNumberOfCluster == 1$ AND EFT on busy cluster is lower
           than Finish Time on $resource_{select}$ then
7:             keep $task_{select}$ for next schedule call
8:         else
9:             Assign $task_{select}$ to $resource_{select}$
10:            remove $resource_{select}$ from $Resource\_free$ list
11:            remove $task_{select}$ from $Ready\_Pool$ list
12:        end if
13:    end while
14: end while
```

In the results presented in [9], the OWM algorithm achieves better performance than Rank_Hybd [20] and Fairness_Dynamic (i.e., a modified version of the fairness algorithm proposed by Zhao *et al.* in [21]) in handling on-line workflows. 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. However, OWM has a 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 commensurate to the number of workflows so that in most cases the scheduler always has a 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 significantly higher in relation to the number of processors, this is for heavier loaded systems.

1.2.2.3 Fairness dynamic workflow scheduling Arabnejad *et al.* proposed the Fairness Dynamic Workflow Scheduling (FDWS) algorithm in [1]. FDWS implements 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 workflows, i.e., the turnaround time that includes execution and waiting time.

The FDWS algorithm comprises three main components: (1) Workflow Pool, (2) Task Selection and (3) Processor Allocation. The Workflow Pool contains the submitted workflows that arrive as users submit their applications. At each scheduling round, this component finds all ready tasks from each workflow. 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, only a single ready task with highest priority from each DAG is added to the ready pool as OWM does. To assign the priority to tasks in the DAG, we use the upward rank $rank_u$ [17] that 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 1.10.

$$rank_u(n_i) = \overline{w_i} + \max_{n_j \in succ(n_i)} \{\overline{c_{i,j}} + rank_u(n_j)\} \tag{1.10}$$

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 $rank_u(n_{exit}) = 0$.

The Task Selection component applies a different rank to select the task to be scheduled from the ready pool. To be inserted into the ready pool the $rank_u$ rank is computed for each DAG individually. To select from the ready pool, $rank_r$ for task t_i belonging to DAG_j is computed, as defined by equation 1.11, and the task with highest $rank_r$ is selected.

$$rank_r(t_{i,j}) = \frac{1}{PRT(DAG_j)} \times \frac{1}{CPL(DAG_j)} \tag{1.11}$$

The $rank_r$ metric considers the Percentage of Remaining Tasks value (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. The CPL length implements a different strategy than Smallest Remaining Processing Time (SRPT) [10] that, 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. But the time needed to complete all tasks of the DAG does not take into account the width of the DAG. A wider DAG has a shorter CPL than other DAG with the same number of tasks, having also a lower expected finish time. Therefore, in such case, FDWS would give higher priority to DAGs with shorter CPL values.

Note that in Rank_Hybd 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.

The Processor Allocation component only considers the free processors and the processor with lowest finish time for the current task is selected. Here, we consider FDWS without processors queues in order to highlight the influence of the rank $rank_r$ in the scheduling results. The algorithm is formalized in Algorithm 4.

Algorithm 4 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:   $Ready\_Pool \leftarrow$ one ready task from each DAG (highest $rank_u$)
7:   compute $rank_r(t_{i,j})$ 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:     $T_{sel} \leftarrow$ the task with highest $rank_r$ from $Ready\_Pool$
10:    $EFT(T_{sel}, P_j) \leftarrow$ Earliest Finish Time of $T_{sel}$ on free Processors $P_j$
11:    $P_{sel} \leftarrow$ the processor with lowest $EFT$ for task $T_{sel}$
12:    Assign Task $T_{sel}$ to processor $P_{sel}$
13:    remove Task $T_{sel}$ from $Ready\_Pool$
14:  end while
15: end while
```

1.2.2.4 On-line Min-Min and On-line Max-Min The Min-Min and Max-Min algorithms have been extensively studied in the literature, such as in [13], and therefore we implemented here an on-line version of these algorithms. Min-min, in a first phase, gives priority to the task with the minimum completion time (MCT). In a second phase, the task with the overall minimum expected completion time is chosen and assigned to the corresponding resource. Our on-line version, in each calling, first collects a single ready task from each available DAG with highest $rank_u$ value and put all these ready tasks into the ready pool of tasks; then calculates the MCT value for each ready task. In the selection phase, the task with minimum MCT value is selected and assigned to the corresponding processor. The calculation of the MCT value for tasks in the ready pool, only consider available (free) processors. The Max-Min algorithm is similar to the Min-Min algorithm, but in the selection phase, a task with maximum MCT is chosen to be scheduled on the resource which is expected to complete the task at earliest time.

1.3 Experimental results and discussion

This section presents performance comparison of the Rank_Hybd, OWM, FDWS, Min-Min and Max-Min algorithms. For this purpose, this section is divided into three parts where first the DAG structure is described, then the infrastructure is presented and in the last part results and discussions are also presented.

1.3.1 DAG structure

To evaluate the relative performance of the algorithms, we considered randomly generated workflow application graphs. For this purpose, we use a synthetic DAG generation program available at [15]. We model the computational complexity of a task as one of the three following forms, which are representative of many common applications: $a.d$ (e.g., image processing of a $\sqrt{d} \times \sqrt{d}$ image), $a.d \log d$ (e.g., sorting an array of d elements), $d^{3/2}$ (e.g., multiplication of $\sqrt{d} \times \sqrt{d}$ matrices) where a is picked randomly between 2^6 and 2^9. As a result different tasks exhibit different communication/computation ratios.

We consider applications that consist of 20 to 50 tasks. We use four popular parameters to define the shape of the DAG: *width*, *regularity*, *density*, and *jumps*. The width determines 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 induces a fat DAG, like a fork-join, with a high degree of parallelism. 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. The 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. The jumps 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 our experiment, for random DAG generation, we consider number of tasks $n = [20...50]$, $jump = [1, 2, 3]$, $regularity = [0.2, 0.4, 0.8]$, $fat = [0.2, 0.4, 0.6, 0.8]$ and $density = [0.2, 0.4, 0.8]$. With these parameters, for each DAG, we call the DAG generator and it picked the value for each parameter randomly from the parameter data set.

1.3.2 Simulated platforms

We use simulation for evaluating the algorithms in the previous section. Simulation allows us to perform a statistically significant number of experiments for a wide range of application configurations (in a reasonable amount of time). We use the SimGrid toolkit[1] [8] as the basis for our simulator. SimGrid provides the required fundamental abstractions for the discrete-event simulation of parallel applications in distributed environments, and it was specifically designed for the evaluation of scheduling algorithms. Relying on a well established simulation toolkit allows us to leverage sound models of a HCS such as the one described in Figure 1.2. Indeed, in many research papers on scheduling, authors assume a contention-free network model in which processors can simultaneously send to or receive data from as much

[1]http://simgrid.gforge.inria.fr

processors as possible without experiencing any performance degradation. Unfortunately such a model, called *multi-port*, is not representative of actual network infrastructures. Conversely, the network model provided by SimGrid correspond to a theoretical *bounded multi-port* model. In this model, a processor can communicate with several other processors simultaneously, but each communication flow is limited by the bandwidth of the traversed route, and communications using a common network link have to share bandwidth. This corresponds to the behavior of TCP connections on a LAN. The validaty of this network model has been demonstrated in [18].

To further increase the realism of our simulations, we consider platforms that derive from clusters in the Grid5000 platform deployed in France[2] [5]. Grid'5000 is an experimental testbed distributed across 10 sites and aggregating a total of around 8,000 individual cores. We consider two sites that comprise multiple clusters. Table 1.1 gives the name of each cluster along with its number of processors, processing speed expressed in flop/s and heterogeneity. Each cluster uses a Gigabit switched interconnect internally. The heterogeneity factor (σ) of a site is determined by the ratio between the speeds of the fastest and slowest processors.

Site Name	Cluster Name	Total CPU	Power Flop/s	site Heterogeneity
grenoble	adonis	12	23.681E9	$\sigma = 1.12$
	edel	72	23.492E9	
	genepi	34	21.175E9	
rennes	paradent	64	21.496E9	$\sigma = 2.34$
	paramount	33	12.910E9	
	parapluie	40	27.391E9	
	parapide	25	30.130E9	

Table 1.1 Description of the Grid'5000 clusters from which are derived the platforms used in our experiments.

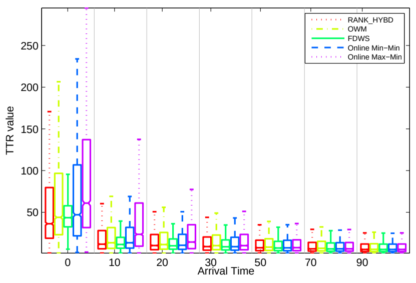
From these five clusters that comprise a total of 280 processors (118 in Grenoble and 162 in Rennes), we extract four distinct heterogeneous cluster configurations, two per site. For the Grenoble site, we build heterogeneous simulated clusters by picking 3 and 5 processors in each of the three actual clusters for a respective total of 9 and 15 processors. We apply the same method to the Rennes site by selecting 2 and 4 processors per cluster for a respective total of 8 and 16 processors. This allows us to have heterogeneous configurations, both in terms of processor speed and network interconnect, that correspond to a set of resources a user can reasonably acquire by submitting a job to the local resource management system in each site.

1.3.3 Results and discussion

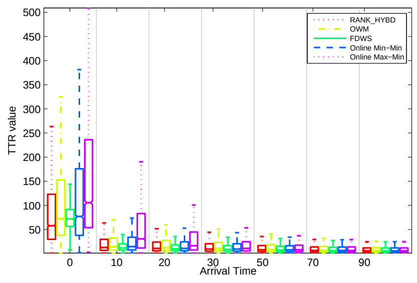
In this section, the algorithms are compared in terms of Turnaround Time Ratio, percentage of Wins and Normalized Turnaround Time. We present results for a set of 30 and 50 concurrent DAGs that arrive with a time interval that ranges from zero (off-

[2]http://www.grid5000.fr

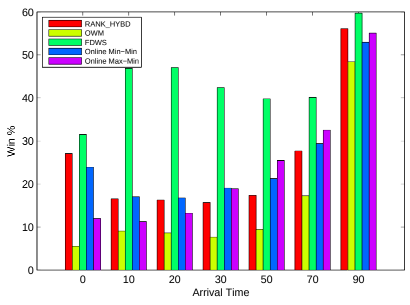
line scheduling) to 90 percent of completed tasks, that is a new DAG is inserted when the correspondent percentage of tasks from the last DAG currently in the system are done. We consider a low number of processors compared to the number of DAGs in order to analyse the behaviour of the algorithms concerning the system load. The maximum load configuration is observed for 8 processors and 50 DAGs.



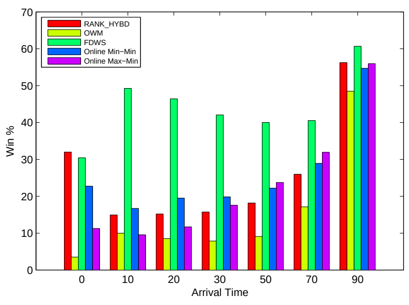
(a) 30 concurrent DAGs



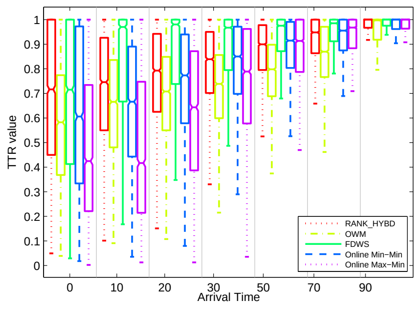
(b) 50 concurrent DAGs



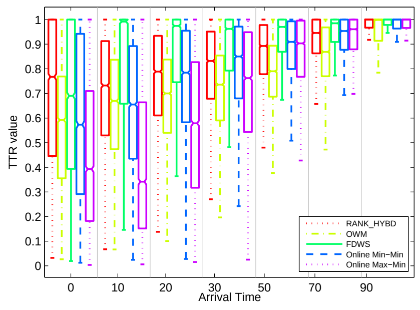
(c) 30 concurrent DAGs



(d) 50 concurrent DAGs



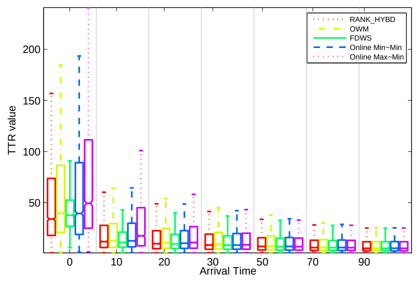
(e) 30 concurrent DAGs



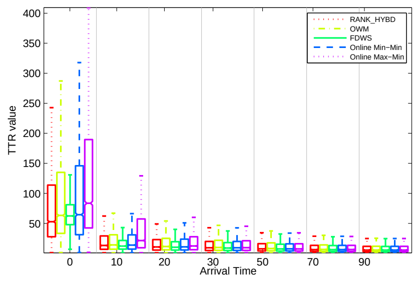
(f) 50 concurrent DAGs

Figure 1.3 Results on Grenoble site with 9 processors

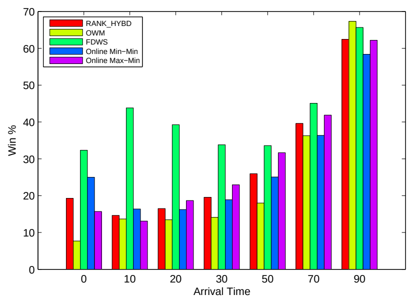
Figures 1.3, 1.4, 1.5 and 1.6 show results for the Grenoble and Rennes sites for two configurations and two sets of DAGs. For the case of zero time interval, equivalent to off-line scheduling, for 8 and 9 processors, 30 and 50 DAGs, FDWS obtains



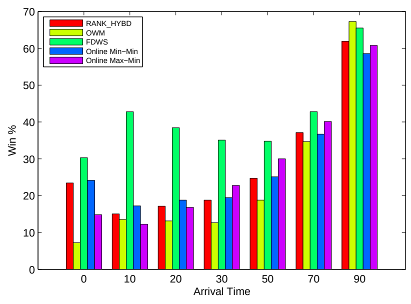
(a) 30 concurrent DAGs



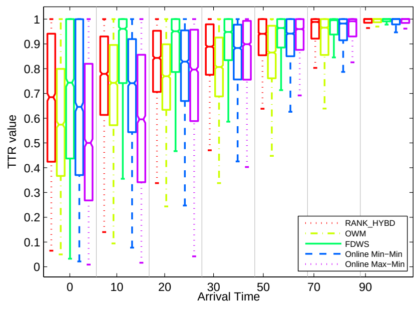
(b) 50 concurrent DAGs



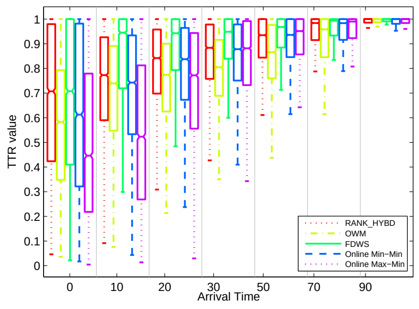
(c) 30 concurrent DAGs



(d) 50 concurrent DAGs



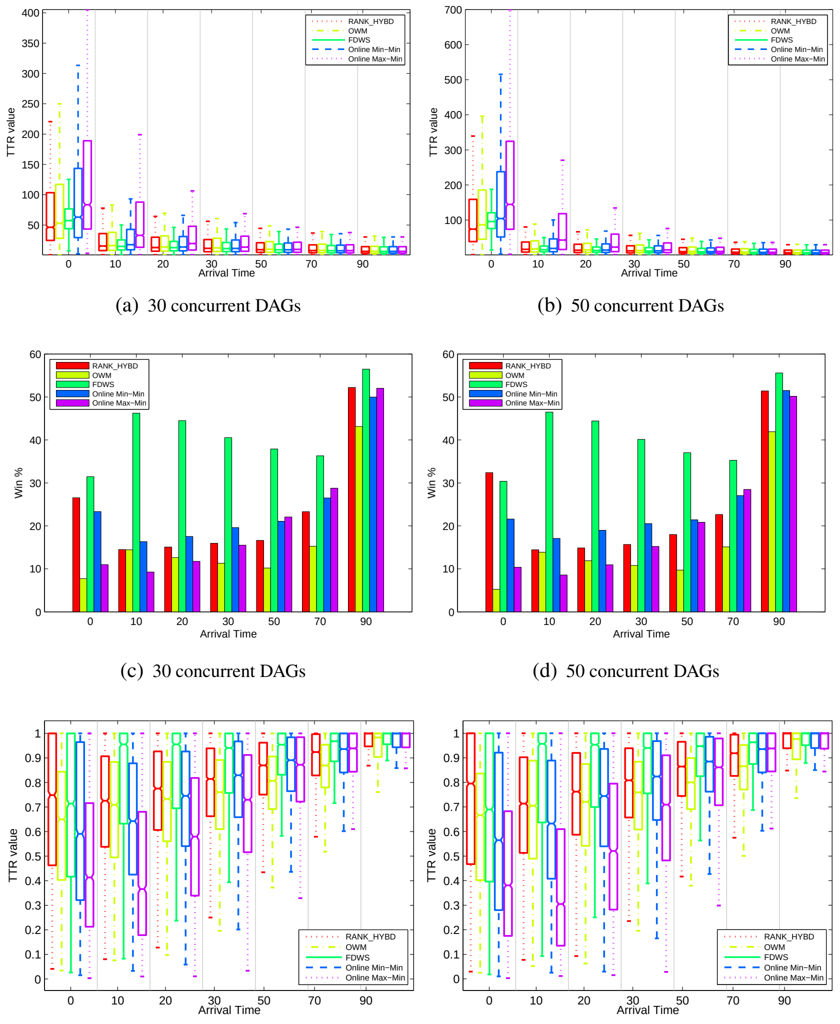
(e) 30 concurrent DAGs



(f) 50 concurrent DAGs

Figure 1.4 Results on Grenoble site with 15 processors

a lower distribution for TTR although with similar average values to Rank_Hybd and OWN. The small box for FDWS means that 50% of the results are in a lower range of values meaning that the individual QoS for each submitted job is better. FDWS generated more times better solutions but, from NTT graphs, we conclude that the distance of its solutions to the minimum Turnaround Time value is similar to Rank_Hybd. For HCS configurations with more resources, 15 and 16 processors for Grenoble and Rennes, respectively, the same behaviour is observed for both cases of 30 and 50 concurrent DAGs.



(a) 30 concurrent DAGs (b) 50 concurrent DAGs

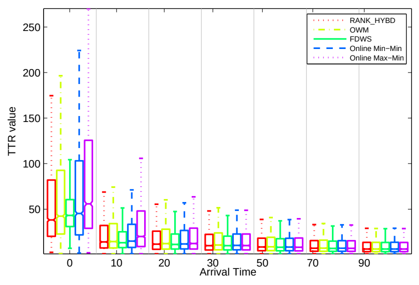
(c) 30 concurrent DAGs (d) 50 concurrent DAGs

(e) 30 concurrent DAGs (f) 50 concurrent DAGs

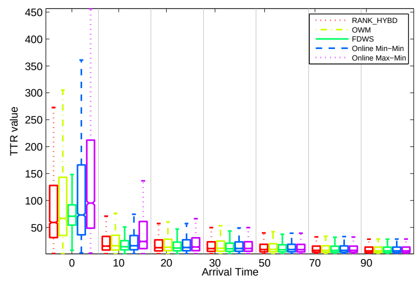
Figure 1.5 Results on Rennes site for 8 processors

The Max-Min algorithm obtained in general worse results. The Min-Min algorithm for values of time interval of 20 and above achieves equivalent performance to Rank_Hybd and better than OWM.

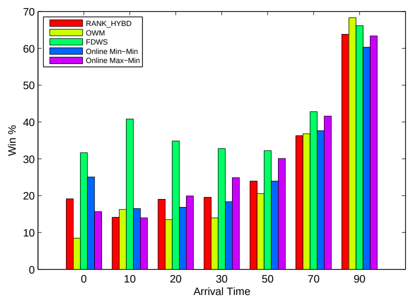
For values of time interval of 10 and above, FDWS achieves consistently better performances which are higher for higher number of concurrent DAGs. For the Rennes site, 10 time interval, 30 DAGs and 8 CPUs, the improvement of FDWS over Rank_Hybd, OWM, Min-Min and Max-Min are 16.2%, 19.3%, 27.4% and 63.3%, respectively. Increasing the number of DAGs to 50, the improvements are 17.5%,



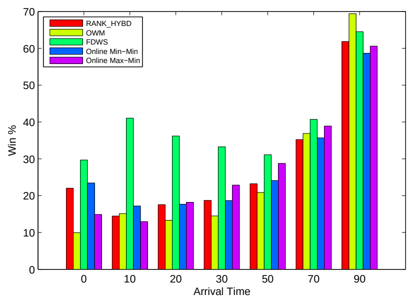
(a) 30 concurrent DAGs



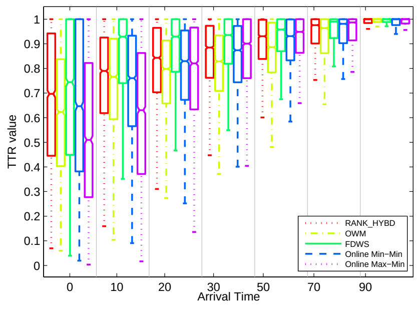
(b) 50 concurrent DAGs



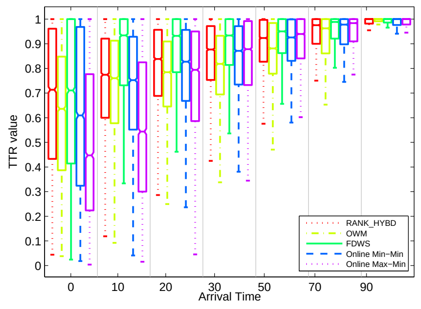
(c) 30 concurrent DAGs



(d) 50 concurrent DAGs



(e) 30 concurrent DAGs



(f) 50 concurrent DAGs

Figure 1.6 Results on Rennes site with 16 processors

23.4%, 31.5% and 71.0%, respectively. Increasing the time interval between DAGs arrival time, reduces the concurrency and the improvements decrease. For the same conditions with 30 DAGs and a time interval of 50 the improvements of FDWS over the others, by the same order, are 5.5%, 11.7%, 4.8% and 8.9%, respectively. For 50 DAGs and 50 units of time interval, the improvements are 5.9%, 13.0%, 3.2% and 11.1%, respectively. For the Grenoble site, with 9 and 15 processors, the improvements are of the same order for the same time intervals and number of DAGs, with 8 and 16 processors in the Rennes site.

Considering the Percentage of Wins FDWS always gives a higher rate of best results, for time intervals equal or higher than 10. From the NTT graphs it can be seen that FDWS also has a distribution closer to 1 meaning that its solutions are closer to minimum Turnaround Time than the remaining algorithms.

1.4 Conclusions

Here we have presented a review on the off-line and on-line concurrent workflow scheduling and compared five algorithms for on-line scheduling when the aim is to maximize the user QoS defined by the completion time of the individual submitted jobs. The algorithms are the Fairness Dynamic Workflow Scheduling (FDWS) [1], On-line Workflow Management (OWM) [9], Rank_Hybd [20], Min-Min and Max-Min algorithms that deal with multiple workflow scheduling in dynamic situations. Based on our experiments, FDWS leads to better performance in terms of Turnaround Time Ratio (TTR), Win(%) and Normalized Turnaround Time (NTT), showing better Quality of Service characteristics for a range of time intervals of 10 to 90. For the time interval of zero, equivalent to off-line scheduling, Rank_Hybd achieved also good performances although the schedules produced by FDWS achieved better QoS characteristics.

Acknowledgements

This work was supported in part by the Fundação para a Ciência e Tecnologia, PhD Grant FCT-DFRH-SFRH/BD/80061/2011. We would like also 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. H. Arabnejad and J.G. Barbosa. Fairness resource sharing for dynamic workflow scheduling on heterogeneous systems. In *10th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA'12)*, pages 633–639, 2012.
2. M. Bakery and R. Buyya. Cluster computing at a glance. *High Performance Cluster Computing: Architectures and Systems*, pages 3–47, 1999.
3. 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.
4. L.F. Bittencourt and E. Madeira. Towards the scheduling of multiple workflows on computational grids. *Journal of Grid Computing*, 8:419–441, 2010.
5. F. Cappello, E. Caron, M. Dayde, F. Desprez, E. Jeannot, Y. Jegou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, and O. Richard. Grid'5000: A Large Scale, Reconfigurable, Controlable and Monitorable Grid Platform. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 99–106, November 2005.
6. A.H. Carbajal, A. Tchernykh, R. Yahyapour, J.L.G. García, T. Röblitz, and J.M.R. Alcaraz. Multiple workflow scheduling strategies with user run time estimates on a grid. *Journal of Grid Computing*, 10:325–346, 2012.
7. H. Casanova, F. Desprez, and F. Suter. On cluster resource allocation for multiple parallel task graphs. *Journal of Parallel and Distributed Computing*, 70:1193–1203, 2010.
8. H. Casanova, A. Legrand, and M. Quinson. Simgrid: a generic framework for large-scale distributed experiments. In *Proceedings of the Tenth International Conference*

on Computer Modeling and Simulation, UKSIM '08, pages 126–131, Washington, DC, USA, 2008. IEEE Computer Society.

9. 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.
10. D. Karger, C. Stein, and J. Wein. Scheduling algorithms. *CRC Handbook of Computer Science*, 1997.
11. 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.
12. Ke Liu, Jinjun Chen, Hai Jin, and Yun Yang. A min-min average algorithm for scheduling transaction-intensive grid workflows. In *Proceedings of the Seventh Australasian Symposium on Grid Computing and e-Research-Volume 99*, pages 41–48. Australian Computer Society, Inc., 2009.
13. Muthucumaru Maheswaran, Shoukat Ali, HJ Siegal, Debra Hensgen, and Richard F Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *Heterogeneous Computing Workshop, 1999.(HCW'99) Proceedings. Eighth*, pages 30–44. IEEE, 1999.
14. 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.
15. DAG Generation Program. https://github.com/frs69wq/daggen, 2012.
16. 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.
17. 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.
18. Pedro Velho and Arnaud Legrand. Accuracy Study and Improvement of Network Simulation in the SimGrid Framework. In *Proccedings of the 2nd International Conference on Simulation Tools and Techniques (SIMUTools)*, Rome, Italy, Mar 2009.
19. Meng Xu, Lizhen Cui, Haiyang Wang, and Yanbing Bi. A multiple qos constrained scheduling strategy of multiple workflows for cloud computing. In *Parallel and Distributed Processing with Applications, 2009 IEEE International Symposium on*, pages 629–634. IEEE, 2009.
20. 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.
21. 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.