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

Scheduling independent workflows on shared resources in a way that satisfy users Quality of Service is a significant challenge. In this study, we describe methodologies for off-line scheduling, where a schedule is generated for a set of known workflows, and on-line scheduling, where users can submit workflows at any moment in time. We consider the on-line scheduling problem in more detail and present performance comparisons of state-of-the-art algorithms for a realistic model of a heterogeneous system.
Keywords: Quality of Service, independent jobs, on-line scheduling, concurrent jobs

1.1 INTRODUCTION

Heterogeneous computing systems (HCSs) are composed of different types of computational units and are widely used for executing parallel applications, predominantly scientific workflows. A workflow consists of 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 its turnaround time, an effective scheduling strategy that decides when and which resource must execute the tasks of the workflow is necessary. When scheduling multiple independent workflows that represent user jobs and are thus submitted at different moments in time, the common definition of makespan must be extended to account for the waiting time and execution time of a given workflow. The metric to evaluate a dynamic scheduler of independent workflows must represent the individual execution time instead of a global measure for the set of workflows to reflect the Quality of Service (QoS) 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 considered in this study consists of workflow applications that are composed of a collection of several interacting components or tasks that must be executed in a certain order for the successful execution of the application as a whole. The scheduling operation, which consists in defining a mapping and an order of task execution, has been addressed primarily for single workflow scheduling, i.e., a schedule is generated for a workflow and a specific number of processors, used exclusively throughout the workflow execution. When several workflows are submitted, they are considered as independent applications that are executed on independent subsets of processors. However, because of task precedence, not all processors are fully used when executing a workflow, thus leading to low efficiency. One way to improve system efficiency is to consider concurrent workflows, i.e., sharing processors among workflows. In this context, there is no exclusive use of processors by a workflow; thus, throughout its execution, the workflow can use any processor available in the system. Although the processors are not used exclusively by one workflow, only one task runs on a processor at any one time.

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

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 and the amount of communication between nodes.

A workflow for this application is modeled by the DAG $G = (V, E)$, where $V = \{n_j, j = 1 \dots v\}$ represents the set of v tasks (or jobs) to be executed and E is a set of e weighted directed edges that represents communication requirements between tasks. Each $edge(i, j) \in E$ represents the precedence constraint that task n_j cannot start before 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 that must 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 Sect. 1.1.2.

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 the processors, i.e., the latencies, are given in a p-dimensional vector L. The communication cost of the $edge(i,j)$, which transfers data from task n_i (executed on processor p_m) to task n_j (executed on processor p_n), is defined as follows:

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

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

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

where $\overline{B}$ is the average bandwidth among all processor pairs and $\overline{L}$ is the average latency. This simplification is commonly considered to label the edges of the graph to allow for the computation of a priority rank before assigning tasks to processors [1].

Due to heterogeneity, each task may have a different execution time on each processor. Then, 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 follows:

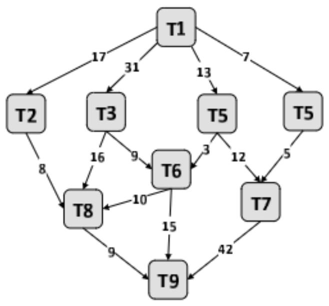
$$\overline{w_i} = \sum_{j=1}^{p} \frac{w_{i,j}}{p}. \tag{1.3}$$

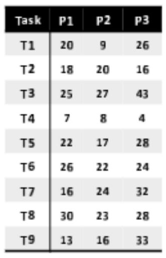
With respect to the communication costs, the average execution time is commonly used to compute the priority ranking for the tasks.

An example is shown in Fig. 1.1 that presents a DAG and a target system with three processors and the corresponding communication and computation costs. In Fig. 1.1, the weight of each edge represents its average communication cost and the numbers in the table represent the computation time of each task at each of the three processors. This model represents a general heterogeneous system.

In this section, we present some of the common attributes used in task scheduling, which we will use 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.





Task	P1	P2	P3
T1	20	9	26
T2	18	20	16
T3	25	27	43
T4	7	8	4
T5	22	17	28
T6	26	22	24
T7	16	24	32
T8	30	23	28
T9	13	16	33

Figure 1.1 Application model and computation time matrix of the tasks in each processor.

- $succ(n_i)$: 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 $AFT(n_{exit})$ is the *Actual Finish Time* of the exit node and $AST(n_{entry})$ is 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 $EST(n_{entry}, p_j) =$

$\max\{T_s, T_{Available}(p_j)\}$, 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 perspective, 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, defined by (1.4).

1.1.2 SYSTEM MODEL

Typically, for executing complex workflows, a high-performance cluster or grid platform is used. As defined in [2], a cluster is a type of parallel or distributed processing system that consists of a collection of interconnected stand-alone computing nodes working together as a single, integrated computing resource. A compute node can be a single or multiprocessor system with memory, input/output (I/O) facilities, accelerator devices, such as graphics processing units (GPUs), and an operating system. A cluster generally refers to two or more computing nodes that are connected together. The nodes can exist in a single cabinet or be physically separated and connected via a local area network (LAN). Figure 1.2 illustrates the typical cluster architecture.

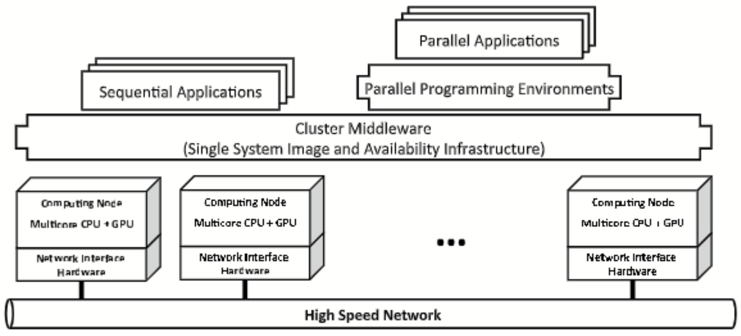


Figure 1.2 Conceptual cluster architecture.

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 in this study, we consider a cluster formed by nodes of the same site, connected 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 allow for the simplification of the communication costs computation in the DAG (Fig. 1.1) by considering the average communication parameters.

The target system can be as simple as a set of devices (e.g., central processing units (CPUs) and GPUs) connected by a switched network that guarantees parallel communication between different pairs of devices. The machine is heterogeneous because CPUs can be from different generations and other very different devices, such as GPUs, can be included. Another common machine is the one that results from selecting processors from several clusters at 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 communication costs are negligible; for higher CCRs, the predominant factor is the network bandwidth, and as mentioned above, we assume 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. Because some metrics may conflict with others, any system design cannot accommodate all metrics simultaneously; thus, a balance according to the final goals must be found. The metrics used in this study are described below.

Makespan

Also referred to as schedule length, makespan is the time difference between the application start time and its completion. Most scheduling algorithms use this metric to evaluate their results and their solutions as compared to other algorithms. A smaller makespan implies 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 spent by the workflow application waiting to get started. It is used to measure the performance and service satisfaction from a user perspective.

Turnaround Time Ratio

The turnaround time ratio (TTR) measures the additional time spent by each workflow in the system to be executed in relation to the minimum makespan obtained for that workflow. The TTR for a workflow is defined as:

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

where P is the set of processors of the HCS. The denominator in the TTR equation is the minimum computation cost of the tasks that compose the critical path (CP), which is the lower bound of the execution time for a workflow.

Normalized Turnaround Time

The normalized turnaround time (NTT) is obtained by the ratio of the minimum turnaround time and actual turnaround time for a given workflow G and an algorithm a_i, defined as follows:

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

where A is the set of algorithms being compared and $a_i \in A$. For an algorithm a_i, NTT provides the distance that its scheduling solutions are from the minimum TTR obtained for a given workflow G. NTT is distributed in the interval $[0, 1]$. The algorithm with a lower spread in NTT with values near one, 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 being scheduled. The algorithm with higher percentage of wins implies that it obtains better results from the user perspective, 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 to improve the execution time of several applications in an HCS system. However, most of these algorithms were designed for off-line scheduling or static scheduling, i.e., all the applications are known at the same time. This approach, although relevant, imposes limitations on 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 address dynamic application scheduling. In the following, a review of off-line scheduling is presented, followed by a review of on-line scheduling.

1.2.1 OFF-LINE SCHEDULING OF CONCURRENT WORKFLOWS

In off-line scheduling, the workflows are available before the execution starts, i.e., at compile time. After a schedule is produced and initiated, no other workflow is considered. This approach, although limited, is applicable in many real-world applications, e.g., when a user has a set of nodes to run a set of workflows. This methodology is applied by the most common resource management tools, where a user requests a set of nodes to execute his/her jobs exclusively.

Several algorithms have been proposed for off-line scheduling, where workflows compete for resources, and the goal is to ensure a fair distribution of those resources, while minimizing the individual completion time of each workflow. Two approaches based on a fairness strategy for concurrent workflow scheduling were presented in [3]. Fairness is defined based on the slowdown that each DAG would experience (the slowdown is the ratio of the expected execution time for the same DAG when scheduled together with other workflows to that when scheduled alone). They proposed two algorithms, one fairness policy based on finish time and another fairness policy based on current time. Both algorithms first schedule each DAG on all processors with static scheduling (like HEFT [1] or Hybrid.BMCT [4]) as the pivot scheduling algorithm, save their schedule assignment, and keep their 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 the highest slowdown and then selects the first ready task that has not been scheduled in this DAG. The main point is to evaluate the slowdown value of each DAG after scheduling a task and make a decision regarding which DAG should be selected to schedule the next task. The difference between the two proposed fairness-based algorithms is that the fairness policy based on finish time calculates the slowdown value of the selected DAG only, whereas the slowdown value is recalculated for every DAG in the fairness policy based on current time.

In [5], several strategies were proposed based on the proportional sharing of resources. This proportional sharing was defined based on the critical path length, width, or work of each workflow. A type of weighted proportional sharing was also proposed that represents a better tradeoff between fair resource sharing and makespan reduction of the workflows. The strategies were applied to mixed parallel applications, where each task could be executed on more than one processor. The proportional sharing, based on the work needed to execute a workflow, resulted in the shortest schedules on 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.

In [6], a path clustering heuristic was proposed 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, the authors propose and compare four algorithms: a) sequential scheduling, where workflows are scheduled one after another; b) gap search algorithm, which is similar to the former but 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 workflow and then scheduled. The evaluation was made in terms of schedule length and fairness and concluded that interleaving the workflows leads to lower average makespan and higher fairness when multiple workflows share the same set of resources. This result, although relevant, considers the average makespan, which does not distinguish the impact of the delay on each workflow, as compared to exclusive execution.

In [7], the algorithms for off-line scheduling of concurrent parallel task graphs on a single homogeneous cluster were evaluated extensively. 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 goal is to optimize 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.

In [8], two workflow scheduling algorithms were presented, multiple workflow grid scheduling, MWGS4 and MWGS2, with four and two stages, respectively. The four stages version comprises labeling, adaptive allocation, prioritization and parallel machine scheduling. The two stages version applies only adaptive allocation and parallel machine scheduling. Both algorithms, MWGS4 and MWGS2, are classified as off-line strategies and both schedule a set of available and ready jobs from 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. These strategies were shown to 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 exhibits dynamic behavior where users can submit the workflows at any time. When scheduling multiple independent workflows that represent user jobs and are thus submitted at different moments in time, the completion time (or turnaround time) includes both the waiting time and execution time of a given workflow, extending the makespan definition for single workflow scheduling [9]. The metric to evaluate a dynamic scheduler of independent workflows must represent the individual completion time instead of a global measure for the set of workflows to measure the QoS experienced by the users related to the finish time of each user application.

Some algorithms have been proposed for on-line workflow scheduling; they will be described briefly in this section. Three other algorithms were proposed specifically to schedule concurrent workflows to improve individual QoS. These algorithms, on-line workflow management (OWM), rank hybrid (Rank_Hybd), and fairness dynamic workflow scheduling (FDWS), are described here and compared in the results section. The first two algorithms improve the average completion time of all workflows. In contrast, FDWS focuses on the QoS experienced by each application (or user) by minimizing the waiting and execution times of each individual workflow.

In [10], the min-min average (MMA) algorithm was proposed to efficiently schedule transaction-intensive grid workflows involving significant 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. In this case, the aim is to optimize the overall throughput rather than the individual workflow performance.

Because min-min is a popular technique, we consider one implementation of min-min for concurrent workflow scheduling in our results.

In [11], an algorithm was proposed for scheduling multiple workflows, with multiple QoS constraints, on the cloud. The resulting multiple QoS-constrained scheduling strategy of multiple workflows (MQMW) minimizes the makespan and the cost of the resources and increases the scheduling success rate. The algorithm considers two objectives, time and cost, that can be adapted to the user requirements. MQMW was compared to Rank_Hybd, and Rank_Hybd performed better when time was the major QoS requirement. In our study application, we consider time as the QoS requirement and thus consider Rank_Hybd in our results section.

In [12], a dynamic algorithm was proposed 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 goal of minimizing a collective metric. This model is applied to real-world applications, such as video surveillance and image registration, 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 study.

1.2.2.1 Rank Hybrid algorithm A planner-guided strategy, the Rank_Hybd algorithm, was proposed by Yu and Shi [13] to address dynamic scheduling of workflow applications that are submitted by different users at different moments in time. The Rank_Hybd algorithm ranks all tasks using the $rank_u$ priority measure [1], which represents the length of the longest path from task n_i to the exit node, including the computational cost of n_i, and is expressed as follows:

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

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

Algorithm 1.1

```
getReadyPool algorithm {
    if (a new workflow has arrived)
        {calculate $rank_u$ for all tasks of the new workflow}
    $Ready\_Pool \leftarrow$ Read all ready tasks from all DAGs
    $multiple \leftarrow$ number of DAGs with ready tasks in $Ready\_Pool$
    if ($multiple$ == 1)
        {Sort all tasks in $Ready\_Pool$ in descending order of $rank_u$}
    else
        {Sort all tasks in $Ready\_Pool$ in ascending order of $rank_u$}
    return $Ready\_Pool$
}
```

In each step, the algorithm reads all of the ready tasks from the 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; if the ready tasks belong to the same DAG, the task with the highest rank is selected. The Rank_Hybd heuristic is formalized in Algorithm 1.2.

Algorithm 1.2

```
Rank Hybrid algorithm {
    while (there are workflows to schedule){
        $Ready\_Pool \leftarrow$ getReadyPool()
        $Resources_{free} \leftarrow$ get all idle resources
        while ($Ready\_Pool \neq \phi$ AND $Resources_{free} \neq \phi$ ){
            $task_{selected} \leftarrow$ the first task in $Ready\_Pool$
            $resource_{selected} \leftarrow$ the processor with the lowest Finish
                Time for $task_{selected}$ on $Resources_{free}$
            Assign $task_{selected}$ to $resource_{selected}$
            Remove $resource_{selected}$ from $Resources_{free}$
            Remove $task_{selected}$ from $Ready\_Pool$
}}}
```

With this strategy, Rank_Hybd allows the DAG with the lowest rank (lower makespan) to be scheduled first to reduce the waiting time of the DAG in the system. However, this strategy does not achieve high fairness among the workflows because it always gives preference to shorter workflows to finish first, postponing the longer ones. For instance, if a longer workflow is being executed and several short workflows are submitted to the system, the scheduler postpones the execution of the longer DAG to give priority to the shorter ones.

1.2.2.2 On-line Workflow Management The on-line workflow management algorithm (OWM) for the on-line scheduling of multiple workflows was proposed in [14]. 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, the task with the highest rank ($rank_u$). Then, until there are some unfinished DAGs in the system, the OWM algorithm selects the task with the 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 will result in the smallest EFT. If the selected processor is free at that time, the OWM algorithm assigns the selected task to the selected processor; otherwise, the selected task stays in the ready list to be scheduled later. The OWM heuristic is formalized in Algorithm 1.3.

In the results presented by Hsu et al. [14], the OWM algorithm performs better than the Rank_Hybd algorithm [13] and the Fairness_Dynamic algorithm (a modified version of the fairness algorithm proposed by Zhao and Sakellariou [3]) in handling on-line workflows. Similar to Rank_Hybd, the OWM algorithm uses a fairness strategy; however, instead of scheduling smaller DAGs first, it selects and schedules tasks from the longer DAGs first. Moreover, OWM has a better strategy by filling the ready list with one task from each DAG so that all of the DAGs have the chance to be selected in the current scheduling round. In their simulation environment, the number of processors was always equal to the number of workflows so that the scheduler typically has a suitable number of processors on which 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 than the number of processors, this is for more heavily loaded systems.

Algorithm 1.3

```
OWM algorithm {
    while (there are workflows to schedule){
        $Ready\_Pool \leftarrow$ getReadyPool()
        $Resources_{free} \leftarrow$ get all idle resources
        while ($Ready\_Pool \neq \phi$ and $Resources_{free} \neq \phi$){
            $task_{selected} \leftarrow$ the first task in $Ready\_Pool$
            $resource_{selected} \leftarrow$ the processor with the lowest Finish
                Time for $task_{selected}$ on $Resources_{free}$
            if (number of free clusters == 1 AND the Finish Time
                on a busy cluster $<$ Finish Time on $resource_{selected}$)
                {Keep $task_{selected}$ for next schedule call}
            else {
                Assign $task_{selected}$ to $resource_{selected}$
                Remove $resource_{selected}$ from $Resources_{free}$
                Remove $task_{selected}$ from $Ready\_Pool$
}}}}
```

1.2.2.3 Fairness Dynamic Workflow Scheduling The fairness dynamic workflow scheduling (FDWS) algorithm was proposed in [15]. FDWS implements new strategies for selecting the tasks from the ready list and for assigning the processors to reduce the individual completion time of the workflows, e.g., the turnaround time, including execution time 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 the highest priority from each DAG into the ready pool. Considering all ready tasks from each DAG leads to an unbiased preference for longer DAGs and the consequent postponing of smaller DAGs resulting in higher TTR and unfair processor sharing. In the FDWS algorithm, only a single ready task with highest priority from each DAG is added to the ready pool, similar to the OWM algorithm. To assign priorities to tasks in the DAG, it uses an upward ranking, $rank_u$ (1.10).

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, $rank_u$ is computed individually for each DAG. To select from the ready pool, $rank_r$ for task n_i belonging to DAG_j is computed, as defined by (1.11), and the task with highest $rank_r$ is selected:

$$rank_r(n_{i,j}) = \frac{1}{PRT(DAG_j)} \times \frac{1}{|CP(DAG_j)|}. \quad (1.11)$$

The $rank_r$ metric considers the percentage of remaining tasks (PRT) of the DAG and its critical path length ($|CP|$). The PRT prioritizes DAGs that are nearly completed and only have a few tasks to execute. The use of CP length results in a different strategy then the smallest remaining processing time (SRPT) [16]. With SRPT the application with the smallest remaining processing time is selected and scheduled at each step. The remaining processing time is the time needed to execute all remaining tasks of the workflow. However, the time needed to complete all tasks of the DAG does not consider the width of the DAG. A wider DAG has a shorter $|CP|$ than other DAGs with the same number of tasks; it also has a lower expected finish time. Therefore, in this case, FDWS would give higher priority to DAGs with smaller $|CP|$ values.

In both Rank_Hybd and OWM, only the individual $rank_u$ is used to select tasks into the workflow pool and to select a task from the pool of ready tasks. This scheme leads to a scheduling decision that does not consider the DAG history in the workflow pool.

The processor allocation component considers only the free processors. The processor with the lowest finish time for the current task is selected. In this study, we use the FDWS without processor queues to highlight the influence of the rank $rank_r$ in the scheduling results. The algorithm is formalized in Algorithm 1.4.

Algorithm 1.4

```
FDWS algorithm {
    while ($Workflow\_Pool \neq \phi$) {
        if (new workflow has arrived){
            Compute $rank_u$ for all tasks of the new Workflow
            Insert the Workflow into $Workflow\_Pool$ }
        $Ready\_Pool \leftarrow$ one ready task from each DAG (highest $rank_u$)
        Compute $rank_r(n_{i,j})$ for each task $n_i \in DAG_j$ in $Ready\_Pool$
        $Resources_{free} \leftarrow$ get all idle resources
        while ($Ready\_Pool \neq \phi$ and $Resources_{free} \neq \phi$) {
            $task_{selected} \leftarrow$ the task with highest $rank_r$ from $Ready\_Pool$
            $resource_{selected} \leftarrow$ the processor with the lowest Finish
                Time for $task_{selected}$ on $Resources_{free}$
            Assign $task_{selected}$ to $Resource_{selected}$
            Remove $task_{selected}$ from $Ready\_Pool$
}}}
```

1.2.2.4 On-line Min-Min and On-line Max-Min The min-min and max-min algorithms have been studied extensively in the literature [17], and therefore, we implemented an on-line version of these algorithms for our problem. In the first phase, min-min prioritizes the task with the minimum completion time (MCT). In the second phase, the task with the overall minimum expected completion time is chosen and assigned to its corresponding resource. In each calling, our on-line version first collects a single ready task from each available DAG with the highest $rank_u$ value and then puts all of these ready tasks into the ready pool of tasks. It then calculates the MCT value for each ready task. In the selection phase, the task with the minimum MCT value is selected and assigned to the corresponding processor. The calculation of the MCT value for the tasks in the ready pool only considers available (free) processors. The max-min algorithm is similar to the min-min algorithm, but in the selection phase, the task with the maximum MCT is chosen to be scheduled on the resource that is expected to complete the task at the earliest time.

1.3 EXPERIMENTAL RESULTS AND DISCUSSION

In this section, we compare the relative performance of the Rank_Hybd, OWM, FDWS, min-min and max-min algorithms. For this purpose, this section is divided into three parts: the DAG structure is described, the infrastructure is presented, and results and discussions are presented.

1.3.1 DAG STRUCTURE

To evaluate the relative performance of the algorithms, we used randomly generated workflow application graphs. For this purpose, we use a synthetic DAG generation program[1]. 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 chosen randomly between 2^6 and 2^9. As a result, different tasks exhibit different communication/computation ratios.

We consider applications that consist of 20-50 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, similar to a chain, with low task parallelism, and a large value induces a fat DAG, similar to a fork-join, with a high degree of parallelism. The regularity indicates the uniformity of the number of tasks in each level. A low value means that the levels contain very dissimilar numbers of tasks, whereas 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, where a low value indicates few edges and a large value indicates many edges. A jump indicates that an edge can go from level l to level $l + jump$. A jump of one is an ordinary connection between two consecutive levels.

In our experiment, for random DAG generation, we consider the number of tasks $n = \{20 \dots 50\}$, $jump = \{1, 2, 3\}$, $regularity = \{0.2, 0.4, 0.8\}$, $fat = \{0.2, 0.4, 0.6, 0.8\}$, and $density = \{0.2, 0.4, 0.8\}$. With these parameters, we call the DAG

[1]https://github.com/frs69wq/daggen

generator for each DAG, and it randomly chooses the value for each parameter from the parameter dataset.

1.3.2 SIMULATED PLATFORMS

We resort to simulation to evaluate the algorithms from the previous section. It 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[2] [18] as the basis for our simulator. SimGrid provides the required fundamental abstractions for the discrete-event simulation of parallel applications in distributed environments. 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 Fig. 1.2. 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 many processors as possible without experiencing any performance degradation. Unfortunately, that model, the *multi-port* model, is not representative of actual network infrastructures. Conversely, the network model provided by SimGrid corresponds 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 scheme corresponds well to the behavior of TCP connections on a LAN. The validity of this network model has been demonstrated in [19].

To make our simulations even more realistic, we consider platforms derived from clusters in the Grid5000 platform deployed in France[3] [20]. Grid5000 is an experimental testbed distributed across 10 sites and aggregating a total of approximately 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 an internal Gigabit-switched interconnect. The heterogeneity factor (σ) of a site is determined by the ratio between the speeds of the fastest and slowest processors.

From these five clusters, which 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 choosing three and five processors for each of the three actual clusters for a respective total of nine and 15 processors. We apply the same method to the Rennes site by selecting two and four processors per cluster for a total of eight and 16 processors. This approach allows us to have heterogeneous configurations in terms of both 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 at each site.

[2]http://simgrid.gforge.inria.fr
[3]http://www.grid5000.fr

Table 1.1 Description of the Grid5000 clusters from which the platforms used in our experiments are derived

Site Name	Cluster Name	Number of CPUs	Power in GFlop/s	Site Heterogeneity
grenoble	adonis	12	23.681	$\sigma = 1.12$
	edel	72	23.492	
	genepi	34	21.175	
rennes	paradent	64	21.496	$\sigma = 2.34$
	paramount	33	12.910	
	parapluie	40	27.391	
	parapide	25	30.130	

1.3.3 RESULTS AND DISCUSSION

In this section, the algorithms are compared in terms of TTR, percentage of wins and NTT. We present results for a set of 30 and 50 concurrent DAGs that arrive with time intervals that range from zero (off-line scheduling) to 90% of completed tasks, i.e., a new DAG is inserted when the corresponding percentage of tasks from the last DAG currently in the system is completed. We consider a low number of processors compared to the number of DAGs to analyze the behavior of the algorithms with respect to the system load. The maximum load configuration is observed for eight processors and 50 DAGs.

Figures 1.3, 1.4, 1.5, and 1.6 present 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 eight and nine processors and 30 and 50 DAGs, FDWS results in a lower distribution for TTR but with similar average values to Rank_Hybd and OWM. The small box for FDWS indicates that 50% of the results fall in a lower range of values, and therefore, the individual QoS for each submitted job is better. FDWS generated better solutions more often, but from the NTT graphs, we conclude that the distance of its solutions to the minimum turnaround time is similar to that of Rank_Hybd. For HCS configurations with more resources (15 and 16 processors for Grenoble and Rennes, respectively), the same behavior is observed for both cases of 30 and 50 concurrent DAGs.

In general, the max-min algorithm yielded poorer results. The min-min algorithm performed the same as Rank_Hybd and performed better than OWM for time intervals of 20 and higher.

For time intervals of 10 and higher, FDWS performed consistently better for higher numbers of concurrent DAGs. For the Rennes site, at 10 time intervals, 30 DAGs, and eight CPUs, the degree of 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%, 23.4%, 31.5%, and 71.0%. Increasing the time intervals between the DAGs arrival times reduces the concurrency, and thus, the improvements decrease. For the same conditions with 30

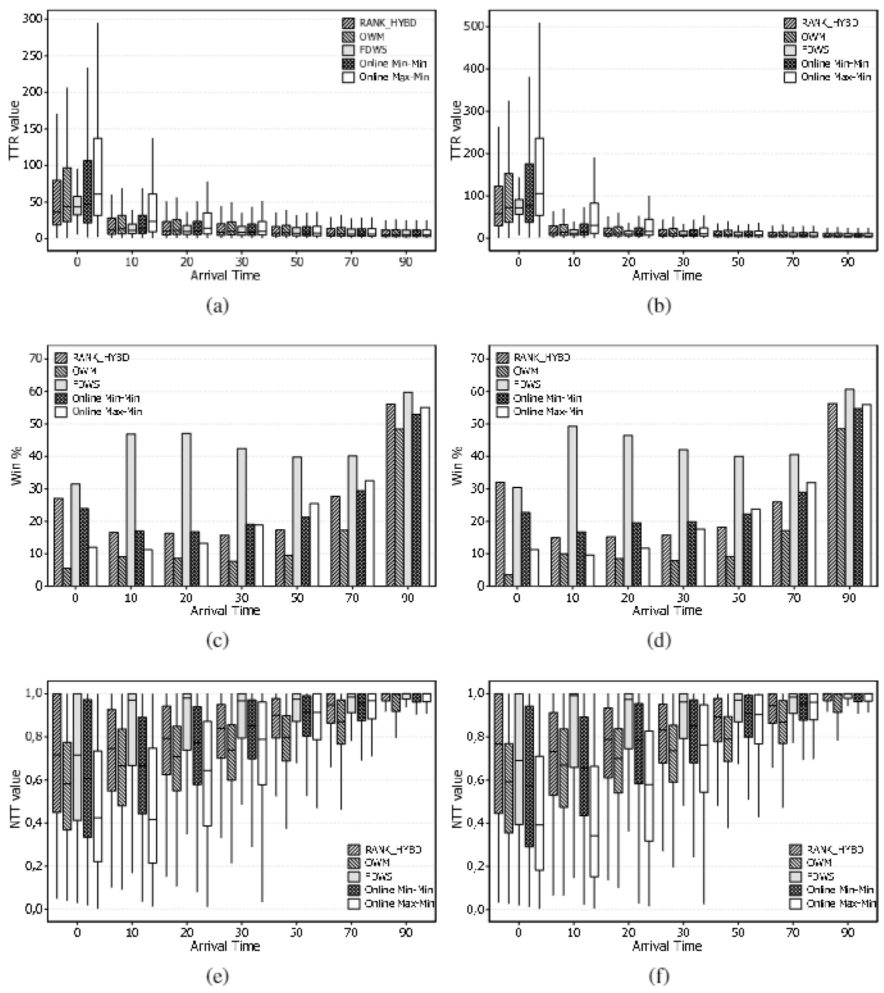


Figure 1.3 Results of TTR, percentage of wins and NTT on Grenoble site with 9 processors. (a)(c)(e) 30 concurrent DAGs. (b)(d)(f) 50 concurrent DAGs.

DAGs and a time interval of 50, the improvement of FDWS over the others, in the same order, are 5.5%, 11.7%, 4.8%, and 8.9%. For 50 DAGs and 50 time intervals, the improvements are 5.9%, 13.0%, 3.2%, and 11.1%. For the Grenoble site, with nine and 15 processors, the improvements are of the same order for the same time intervals and number of DAGs, with eight and 16 processors in the Rennes site.

With respect to the percentage of wins, FDWS always results in a higher rate of best results, for time intervals equal to or higher than 10. The results in the NTT graphs illustrate that FDWS also has a distribution closer to one, which indicates that its solutions are closer to the minimum turnaround time than the other algorithms.

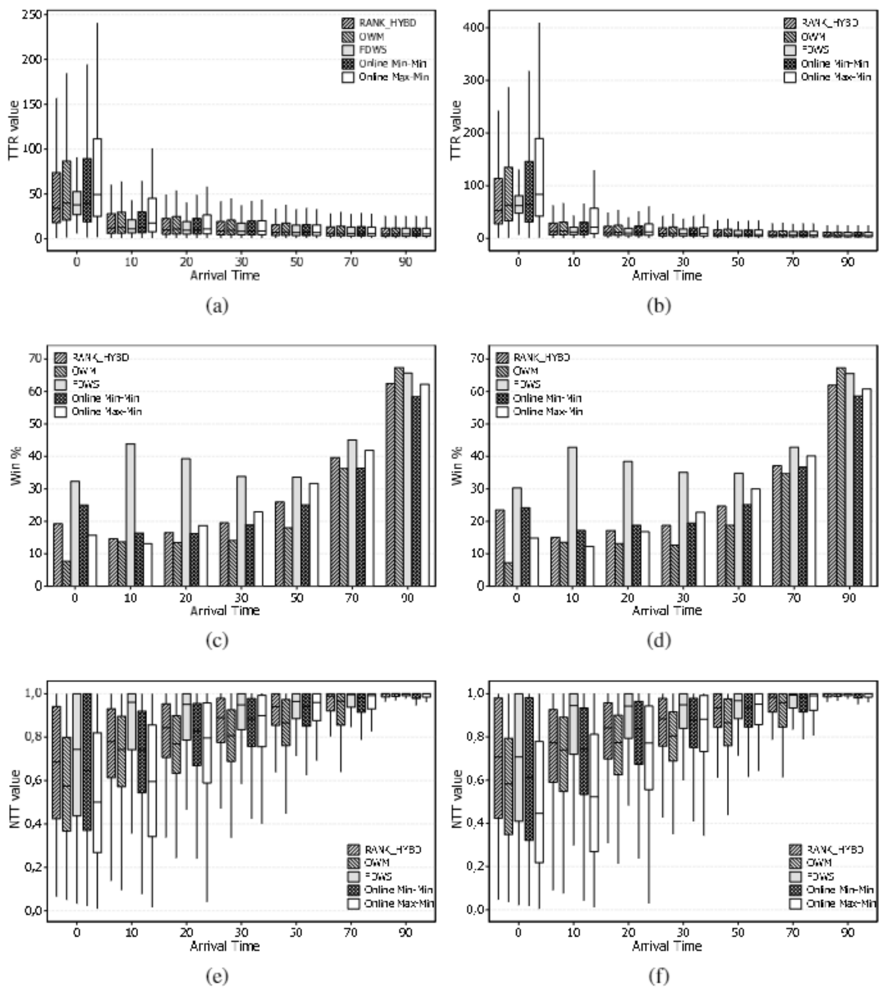


Figure 1.4 Results of TTR, percentage of wins and NTT on Grenoble site with 15 processors. (a)(c)(e) 30 concurrent DAGs. (b)(d)(f) 50 concurrent DAGs.

1.4 CONCLUSIONS

In this study, we presented a review of off-line and on-line concurrent workflow scheduling and compared five algorithms for on-line scheduling when the goal was to maximize the user QoS defined by the completion time of the individual submitted jobs. The five algorithms are FDWS [15], OWM [14], Rank_Hybd [13], on-line min-min, and on-line max-min, which can all handle multiple workflow scheduling in dynamic situations. Based on our experiments, FDWS leads to better performance in terms of TTR, win(%), and NTT, showing better QoS characteristics for a range

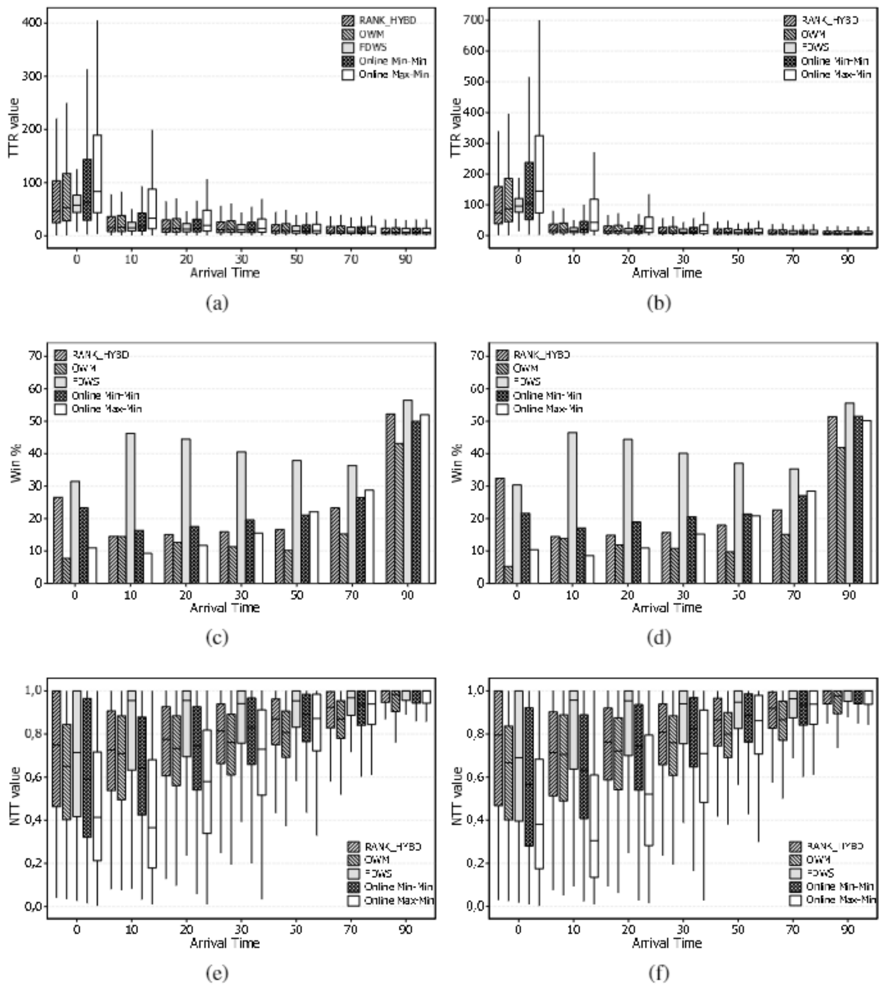


Figure 1.5 Results of TTR, percentage of wins and NTT on Rennes site for 8 processors. (a)(c)(e) 30 concurrent DAGs. (b)(d)(f) 50 concurrent DAGs.

of time intervals from 10 to 90. For the time interval of zero, equivalent to off-line scheduling, Rank_Hybd also performed well, but the schedules produced by FDWS had 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 the European Cost Action IC0805 Open Network for High-Performance

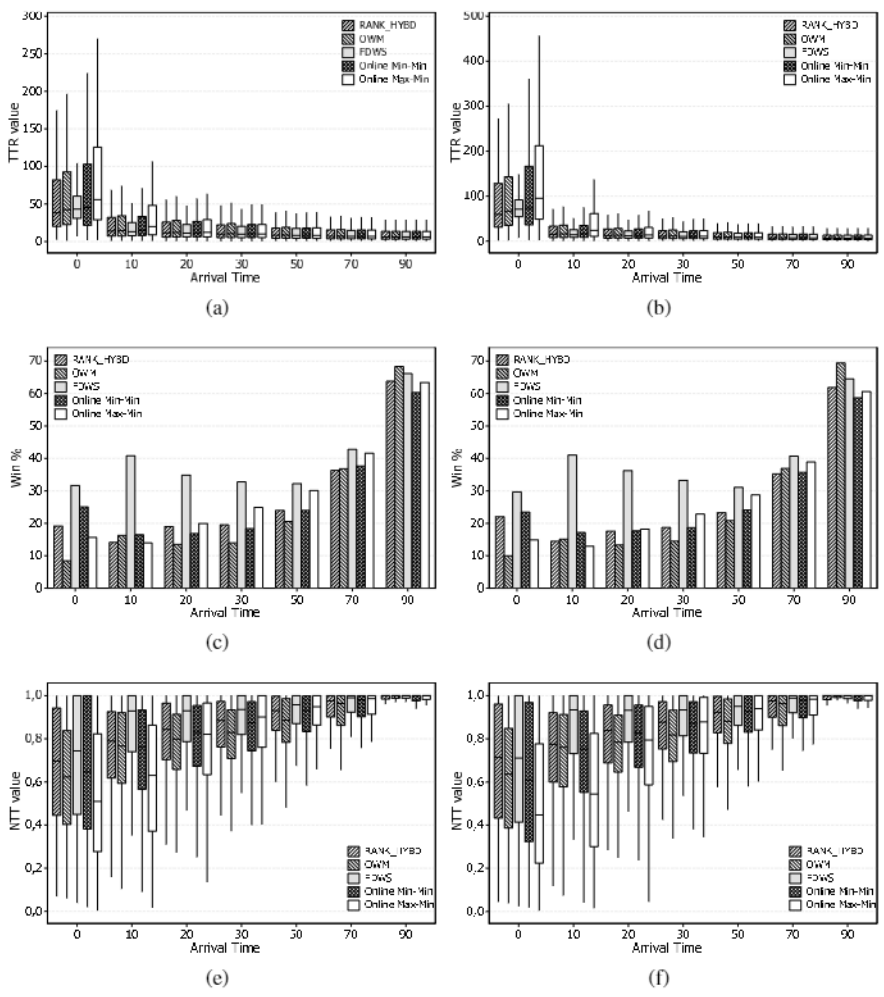


Figure 1.6 Results of TTR, percentage of wins and NTT on Rennes site with 16 processors. (a)(c)(e) 30 concurrent DAGs. (b)(d)(f) 50 concurrent DAGs.

Computing on Complex Environments, Working Group 3: Algorithms and tools for mapping and executing applications onto distributed and heterogeneous systems.

REFERENCES

1. H. Topcuoglu, S. Hariri, and M. Wu, “Performance-effective and low-complexity task scheduling for heterogeneous computing,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.

2. M. Bakery and R. Buyya, “Cluster computing at a glance,” *High Performance Cluster Computing: Architectures and Systems*, pp. 3–47, 1999.

3. H. Zhao and R. Sakellariou, “Scheduling multiple DAGs onto heterogeneous systems,” in *International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1–14, IEEE, 2006.

4. R. Sakellariou and H. Zhao, “A hybrid heuristic for dag scheduling on heterogeneous systems,” in *International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 111–123, IEEE, 2004.

5. T. N’takpé and F. Suter, “Concurrent scheduling of parallel task graphs on multi-clusters using constrained resource allocations,” in *International Symposium on Parallel and Distributed Processing (IPDPS)*, pp. 1–8, IEEE, 2009.

6. L. Bittencourt and E. Madeira, “Towards the scheduling of multiple workflows on computational grids,” *Journal of Grid Computing*, vol. 8, pp. 419–441, 2010.

7. H. Casanova, F. Desprez, and F. Suter, “On cluster resource allocation for multiple parallel task graphs,” *Journal of Parallel and Distributed Computing*, vol. 70, pp. 1193–1203, 2010.

8. A. Carbajal, A. Tchernykh, R. Yahyapour, J. García, T. Röblitz, and J. Alcaraz, "Multiple workflow scheduling strategies with user run time estimates on a grid," *Journal of Grid Computing*, vol. 10, pp. 325–346, 2012.

9. Y. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Computing Surveys*, vol. 31, no. 4, pp. 406–471, 1999.

10. K. Liu, J. Chen, H. Jin, and Y. 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*, pp. 41–48, Australian Computer Society, Inc., 2009.

11. M. Xu, L. Cui, H. Wang, and Y. Bi, "A multiple QoS constrained scheduling strategy of multiple workflows for cloud computing," in *International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pp. 629–634, IEEE, 2009.

12. J. Barbosa and B. Moreira, "Dynamic scheduling of a batch of parallel task jobs on heterogeneous clusters," *Parallel Computing*, vol. 37, no. 8, pp. 428–438, 2011.

13. Z. Yu and W. Shi, "A planner-guided scheduling strategy for multiple workflow applications," in *International Conference on Parallel Processing-Workshops (ICPP-W'08)*, pp. 1–8, IEEE, 2008.

14. C. Hsu, K. Huang, and F. Wang, "Online scheduling of workflow applications in grid environments," *Future Generation Computer Systems*, vol. 27, no. 6, pp. 860–870, 2011.

15. H. Arabnejad and J. Barbosa, "Fairness resource sharing for dynamic workflow scheduling on heterogeneous systems," in *International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pp. 633–639, IEEE, 2012.

16. D. Karger, C. Stein, and J. Wein, "Scheduling algorithms," *CRC Handbook of Computer Science*, 1997.

17. M. Maheswaran, S. Ali, H. Siegal, D. Hensgen, and R. F. Freund, "Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems," in *Proceedings of the Eighth Heterogeneous Computing Workshop*, pp. 30–44, IEEE, 1999.

18. 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*, pp. 126–131, IEEE Computer Society, 2008.

19. P. Velho and A. 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)*, March 2009.

20. 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, "Grid5000: A Large Scale, Reconfigurable, Controlable and Monitorable Grid Platform," in *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pp. 99–106, Nov. 2005.