**Journal of Grid Computing manuscript No.**
(will be inserted by the editor)



# Towards the scheduling of multiple workflows on computational grids

**Luiz F. Bittencourt · Edmundo R. M. Madeira**

Received: date / Accepted: date

**Abstract** The workflow paradigm has become the standard to represent processes and their execution flows. With the evolution of e-Science, workflows are becoming larger and more computational demanding. Such e-Science necessities match with what computational grids have to offer. Grids are shared distributed platforms which will eventually receive multiple requisitions to execute workflows. With this, there is a demand for a scheduler which deals with multiple workflows in the same set of resources, thus the development of multiple workflow scheduling algorithms is necessary. In this paper we describe four different initial strategies for scheduling multiple workflows on grids and evaluate them in terms of schedule length and fairness. We present results for the initial schedule and for the makespan after the execution with external load. From the results we conclude that interleaving the workflows on the grid leads to good average makespan and provides fairness when multiple workflows share the same set of resources.

**Keywords** grid computing · workflow · scheduling

## 1 Introduction

Grid computing has turned into a widely used paradigm for processing high amounts of data over a geographically distributed system. Computational grids are receiving substantial attention from the research community since the middle of the 1990's [25]. A computational grid is a dynamic, heterogeneous, and shared system which brings new characteristics along with new problems to be addressed. Problems concerning scalability and resource management include new variables to be tackled, thus presenting new challenges. Among these new challenges, the task scheduling is an important one.

In a grid middleware, the scheduler is responsible for selecting in which resource each task will be executed, thus task execution times are strongly dependent on the scheduler decisions. In general, the task scheduling problem is NP-Hard [41], consequently there is no known algorithm which finds the optimal solution in polynomial time. Significant effort has been made towards the development of algorithms which can provide efficient solutions for the scheduling problem in distributed systems [23,28,8,31,44,38,37,50,32,27].

Tasks submitted to be processed in a grid can be splitted into two classes, in general: dependent tasks and independent tasks. While independent tasks have no communication between them, dependent tasks have a data flow, where each task depends on data computed by other tasks. A set of dependent tasks can be called as a workflow: a collection of tasks that have dependencies between them. This paradigm has been widely used to represent scientific processes [45]. With the emergence of grid computing, the scheduling of workflows, particularly for e-Science applications, is receiving substantial attention [4,22,5,43,7,42,49,9]. Some known applications which make use of the workflow paradigm are Montage [18], AIRSN [54], LIGO [17], Chimera [3], and CSTEM [20], including applications in chemistry, biology, physics, and computer science.

Luiz F. Bittencourt · Edmundo R. M. Madeira
University of Campinas - UNICAMP
Institute of Computing
P. O. 6176 - Campinas, São Paulo, Brazil
E-mail: bit@ic.unicamp.br

Edmundo R. M. Madeira
E-mail: edmundo@ic.unicamp.br
Phone: +55-19-3521-5862
FAX: +55-19-3521-5847

Considering grids as shared environments, they will eventually need to handle more than one workflow at the same time, as well as execute tasks proceeding from users outside the grid. With that, it urges the development of grid middlewares to efficiently enact more than one workflow in the available resources taking into account the external load in those resources. As an important part of the grid middleware, the scheduler also presents the necessity of handling multiple workflows. Despite many work has been done in workflow scheduling, the scheduling of multiple workflows stills being an open problem, and only initial studies exist in the literature [53,6]. The improvement of the execution of multiple workflows can speed up the result arrival when processes are submitted to the grid, contributing to the e-Science development.

The most common objective of a workflow scheduling algorithm is to minimize the schedule length (makespan). The schedule length is directly responsible for the execution time of the workflow. When multiple workflows share the same execution environment, besides the minimization of the makespan, there are conflicts which must be observed and tackled to guarantee the system efficiency. For example, how to schedule workflows such that no one will be allocated on the resources with disadvantage when compared to the other ones in terms of execution time. If there are multiple workflows to be scheduled and the algorithm considers only one at a time, the first one considered will be in advantage, thus its makespan will be minimized by a greater amount than the makespan of the last workflow to be scheduled. Thus, a scheduling algorithm should consider fairness to share the resources equally among the arriving workflows.

As we consider fairness and makespan two important issues when scheduling multiple workflows, in this paper we focus on the scheduling of multiple workflows in grids aiming at the minimization of the schedule length and maximization of fairness among processes. We describe four different approaches to schedule multiple workflows and analyze how each strategy impacts on the schedule length and on fairness when scheduling up to ten workflows at the same time. In addition to this, we evaluated how each algorithm performs in the grid shared environment by performing execution simulations when grid resources have external load.

The rest of the paper is organized as follows. A background on workflow representation and the scheduling problem are presented in Section 2. Related works are introduced in Section 3, while the multiple workflow scheduling strategies are described in Section 4. Simulation results, including evaluation of the initial makespan given by each algorithm and the execution time on a shared environment, are show in Section 5. Section 6 presents the concluding remarks as well as future directions.

## 2 Background

In this section we introduce some basic concepts needed for the understanding of this article. We first introduce how workflows are represented, then we present the considered grid architecture, and we finish the necessary background by introducing the task scheduling problem.

### 2.1 Workflow representation

The workflow paradigm has become commonly used to represent scientific applications in recent years [45]. Many workflow programming languages exist in the literature, and the workflow represented by those languages can be translated into a directed acyclic graph (DAG) in general. Workflow programming languages as SwiftScript [55] and Xavantes [13] may have loop and conditional structures that prevent the scheduler to know beforehand which (or how many) tasks will be executed. However, the static part of the workflow is still representable by a DAG, while the dynamic part of the workflow can be translated to compose the DAG at real-time, and then scheduled using the same scheduling algorithm. A middleware support to this kind of real-time translation would be necessary for dynamic DAGs to be scheduled by the algorithms presented in this paper.

In the workflow scheduling algorithms research, a DAG is the standard manner to represent a workflow, usually translated from a workflow programming language. In this representation, a process (workflow process) of dependent tasks is a DAG $G = (V, E)$, where:

> $V$ is the set of $n_v = |V|$ tasks, and $t_i \in V$ represents one atomic task that must be executed on a resource.
> $E$ is the set of $n_e = |E|$ directed edges, and $e_{i,j} \in E$ represents a data dependency between $t_i$ and $t_j$.

Labels on nodes represent computation costs and labels on edges represent communication costs. If $\exists e_{i,j} \in E$, a task $t_j$ can only start its execution after $t_i$ finishes and sends all the necessary data to $t_j$. An example of DAG is shown in Figure 1, where task 10 can start its execution only after tasks 7 and 8 finish and send the data to task 10. Note that, for example, tasks 7 and 8 are independent and can execute in parallel. The words *process, workflow*, and *DAG*, within this article, refer to

the process composed of dependent tasks and are used interchangeably.

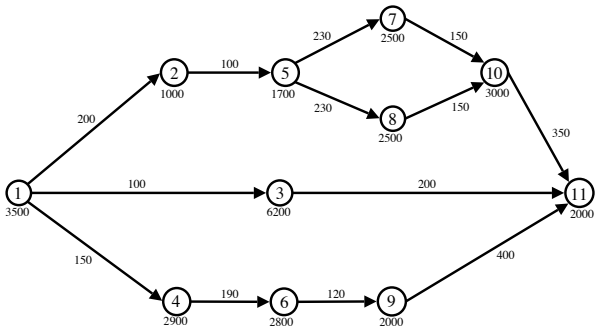


**Fig. 1** Example of DAG representing a process.

The first node of the DAG is called entry task ($t_{entry}$) and the last node is called exit task ($t_{exit}$). We assume that all DAGs have only one entry and one exit task. If a DAG has more than one entry or more than one exit, one entry task and one exit task, both with zero cost, are added to the graph, along with costless edges connecting them to the original entry/exit tasks.

## 2.2 The grid architecture

A computational grid is composed of heterogeneous resources connected by heterogeneous links. We consider that groups of resources compose the grid, and an example of the grid infrastructure architecture considered is shown in Figure 2. In this example, the thickness of the edges represents bandwidth, diameter of computational resources represents processing power, and gray tones of computer resources represent different operating systems. Therefore, the infrastructure is composed of heterogeneous resources arranged in groups, where each group can be a LAN or a cluster, for instance.

Each group is autonomous and each resource in a group is also autonomous. Additionally, a group may have only one resource, characterizing a personal computer or any kind of resource that is somehow independent. Groups are connected by heterogeneous links. Each resource has a computational power associated, and each link has a bandwidth capacity. Resources inside the same group are considered to have links with the same bandwidth between them, inspired in the common group architecture encountered in laboratories and computational clusters.

Additionally, the grid can be arranged in different manners, depending on the middleware being used. For instance, the middleware can create pools of autonomous groups coordinated by an upper level entity. In this

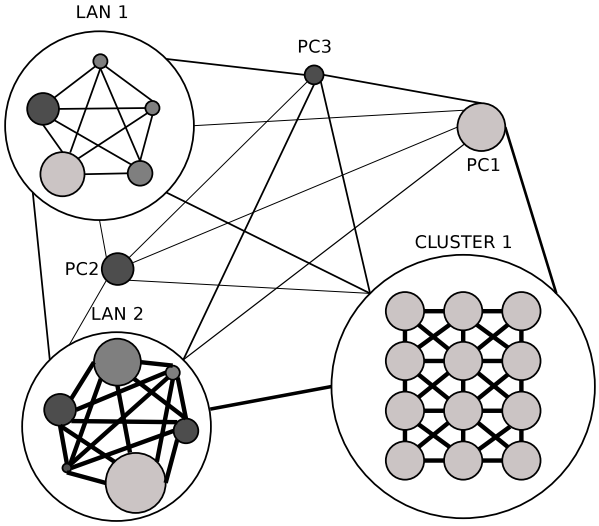


**Fig. 2** Example of the computational grid infrastructure considered.

case, a meta-scheduler in the top level would be responsible for the distribution of the requisitions among pools, and a low level scheduler would be responsible to distribute the tasks inside the pool. With this the system can become more scalable while mantaining the group architecture shown above. The algorithms presented here can be used in the low level, to schedule the tasks to the resources available.

We consider that each resource has a queue of tasks to be executed, and there is no simultaneous execution of tasks managed by the grid in the same resource. The scheduler inserts tasks on this queue, and once a task is started, it executes alone in the resource until it finishes. In this context, the word *alone* means that the task does not share the resource with another task submitted to the grid. However, in such a dynamic environment, it may share the resource with tasks not managed by the grid middleware, characterizing an external load.

## 2.3 The task scheduling problem

The task scheduling problem consists in selecting one resource for each task in the DAG, which represents the workflow, optimizing an objective function. This problem was extensively studied in homogeneous systems [32, 37, 50]. With the heterogeneous systems emergence, the scheduling of task graphs has been studied taking into consideration the heterogeneity of processors and communication links [8, 31, 48]. Moreover, the objective function can have one or more criteria to be optimized, such as the schedule length (makespan), communication costs, resource utilization, fairness, and so forth.

Scheduling algorithms can be classified as schedulers for independent tasks and schedulers for dependent tasks. The scheduling of independent tasks does not need to tackle with tasks communication costs and dependencies, thus each task can be scheduled independently. We are interested in the scheduling of dependent tasks, since workflows are composed of tasks with dependencies. In this type of scheduling, the algorithm must take into consideration the communication costs proceeding from data dependencies between tasks. These costs are implicit when tasks are sent to execution over a set of distributed resources, with the heterogeneity of links playing an important role in this aspect.

Scheduling algorithms can also be classified as static schedulers and dynamic schedulers. Static schedulers schedule all tasks using the information initially available about the resources, while dynamic schedulers may schedule tasks in turns, updating the information about resources at each scheduling turn. It is important to note that static scheduling algorithms are useful to the development of dynamic schedulers, which can use the static scheduler to schedule parts of the workflow in turns. In that sense, we consider that a good static scheduler is the starting point for a good dynamic scheduler. In this work we deal with static schedulers for dependent tasks.

Given the NP-Completeness of the scheduling problem, some techniques are used to try to approximate the optimal solution using algorithms with polynomial time complexity. Some techniques and their characteristics are listed below.

- Heuristics: can have low complexity and fast execution, but provide no guarantee related to the solution quality.
- Meta-heuristics: execution time depends on the stop condition (usually the number of iterations) imposed by the programmer. To achieve good quality solutions the execution time is usually higher than that of heuristics. Also provide no guarantee related to the solution quality: local optima is frequently the final solution.
- Linear programming: execution time and solution quality depend on relaxing restrictions and reducing the number of variables.
- Approximation algorithms: hard to obtain low complexity approximation algorithms for general problems, but they provide guarantee related to the solution quality, giving bounds from the optimal solution. More general the problem, harder to obtain a satisfactory approximation.

The approach most commonly found in the literature of DAG scheduling is the use of heuristics, which can combine good results in general and low complexity. List scheduling, clustering and task duplication are well-known DAG scheduling techniques for both homogeneous and heterogeneous systems. Scheduling heuristics can be classified in three categories, according to the technique used:

**List scheduling** is a priority-based technique. First, each node of the graph receives a priority level. While there are unscheduled nodes (or tasks), the highest-priority ready task is selected (task selection phase) and scheduled on a processor which minimizes a pre-defined cost function (processor selection phase).

**Clustering** is a two-phase technique. In the first phase, called clustering phase, nodes are arranged in groups of tasks (clusters), aiming at reducing communication between nodes. In the second phase, called mapping phase, each cluster is assigned to an available resource according to the objective function. Thus, tasks which are in the same cluster are scheduled on the same processor.

**Task duplication** is a technique where a task can be duplicated, i.e., scheduled on more than one resource, hence executed more than once. For example, the task being scheduled may be duplicated to improve dependability, and the instance which finishes first sends the result to its successors.

In order to schedule the workflow tasks, in this work we use the Path Clustering Heuristic (PCH) [7], which combines the clustering and list scheduling techniques.

When scheduling multiple DAGs at the same time on the same set of resources, it is important to distribute the tasks on the resources in a way that the processing capacities are fairly shared among the workflows. This way, no workflow may have advantage in the execution, considering that all workflows have the same priority. In this article we study the scheduling of multiple workflows aiming at the minimization of the workflows' execution time and the maximization of fairness.

## 3 Related work

A computational grid is a heterogeneous, dynamic and widely distributed system. These characteristics bring to the grids some peculiarities when compared to dedicated systems. The study of inherent characteristics of grids, including task scheduling on grids, is receiving substantial attention nowadays [13, 29, 24, 14, 52, 28, 10, 34, 42, 11, 26, 46, 51, 21, 43].

Task scheduling is a largely studied topic [41]. It is an NP-Complete problem, thus its optimization ver-

sion is NP-Hard. Therefore, most efforts in scheduling algorithms are concentrated on heuristics and meta-heuristics [22,35,15,30]. There are some works which use approximation algorithms [40,27] and combinatorics approaches [5]. The task scheduling has been studied in homogeneous systems [32,37,50,23]. With the development of heterogeneous systems and the grid, it has been necessary to consider the heterogeneity in processing power and bandwidth [1,8,19,16,31,4,48,38,44].

The Heterogeneous Earliest Finish Time (HEFT) [48] is a well known scheduling heuristic for single DAGs in heterogeneous systems which uses the list scheduling technique. First, HEFT computes the $rank_u$ value for each task, traversing the graph backwards. The unscheduled task with the highest $rank_u$ value is scheduled on the processor that gives the smallest estimated finish time for the task. HEFT looks for idle time slots in the current schedule when scheduling a task. The Critical Path on a Processor (CPOP) [48] algorithm is similar to HEFT. The main difference is that all nodes on the DAG's critical path are scheduled on the same processor. HEFT and CPOP can give good results in non shared environments where DAGs are not executed at the same time.

Condor [39] is a well known inter-domain resource manager which is used in grids through the Condor-G [46]. The DAG Manager (DAGMan) [26] is the Condor DAG management system. The DAGMan [26] is actually a meta-scheduler for DAGs. It receives the DAG and performs the task selection, sending them independently to be scheduled by a lower level scheduler. Therefore, the scheduler schedules tasks without knowledge about their dependencies, which may lead to higher communication costs.

In [42] a case study on dynamic scheduling of workflows in grids is presented. A rescheduling strategy is proposed, where a new DAG with the tasks that will be rescheduled is generated at each iteration. After that, the new generated DAG is scheduled, actually performing a re-schedule of the original DAG tasks. The re-scheduling of DAGs can be complementary to the work presented here, but the re-scheduling of multiple workflows has peculiarities, such as inter-process deadlocks [6], which must be addressed. An initial approach to re-schedule multiple workflows is presented in [6].

A two level scheduler for wide area networks (WANs) is proposed in [11]. A top level scheduler selects in which local area network (LAN) the entire DAG will be executed. There is no mechanism to split the DAG into more than one LAN, which can lead to higher execution times. Also, all second level scheduler must reply to the top level scheduler. Thus, more requisitions means more data to be transmitted to/from the schedulers, leading to higher scheduling times.

In [14] the authors propose a scheduling strategy for the GraDS project, including an approach to schedule workflows and application reschedule. The workflow scheduling in the GraDS project uses a strategy of classifying each resource according to its affinity with each component of the process, where the lower the classification, better the resource matches with the component of the process. The scheduler puts these information in a performance matrix where the element $e_{i,j}$ of the matrix shows the affinity of the component $i$ of the process with the resource $j$. Finally, three heuristics are run over the matrix to determine the scheduling of each component.

Workflow scheduling in conjunction with service composition in grids are discussed in [52]. An algorithm to schedule composed services is proposed. The composed services are modeled as a workflow, and then this workflow is scheduled using a list scheduling heuristic.

An approximation algorithm for task scheduling is presented in [27]. The algorithm uses the resource utilization as the performance criterion, not focusing on makespan minimization. Since approximation algorithms are hard to be developed, the authors make some assumptions to make the problem more specific, like considering that all tasks of the workflow have the same weight.

Although there exist many works dealing with workflow scheduling in grids [51], the multiple workflow scheduling is not receiving the deserved attention. Grids are shared environments and will eventually process more than one workflow at the same time. In [53] the authors show an initial analysis on the behavior of multiple workflow scheduling strategies for heterogeneous static systems. How to schedule such processes taking into account their execution time, scalability, and fairness considering the performance degradation of grid resources is necessary, which is the focus of this article.

## 4 Multiple workflow scheduling

The starting point in the scheduling of multiple workflows is to decide if the DAGs will be scheduled independently or if they will be combined in a single DAG and scheduled after that. To schedule more than one DAG we can adopt three strategies, in general:

- Schedule the DAGs independently, one after another.
- Schedule the DAGs independently in turns, interleaving parts of each DAG being scheduled.
- Merge the DAGs into a single one, and schedule this resulting DAG.

To schedule multiple workflows we assume that, at a given time, we have tasks of $N$ workflows to be scheduled. Note that this does not necessarily mean that we need to schedule all tasks of all workflows at the starting time of the scheduling work. When one or more workflows arrive we consider all non-executed tasks of workflows that arrived before. If we only consider workflows that just arrived, we would not take advantadge of the communication times left by the workflows already scheduled but not executed yet. On the other hand, if we wait for multiple workflows to arrive, this may delay the execution of the first workflow. Thus, when a workflow arrives, we consider the non-executed tasks of the workflows that arrived before as workflows to be re-scheduled. This way, the new workflow is mixed with the workflows already scheduled to take advantadge of their communication times. Which tasks and workflows would be considered for re-scheduling is defined by the middleware, which may decide it based on how much time is left to each workflow to finish, for instance.

In this section we outline four different algorithms to schedule multiple workflows:

- **Sequential algorithm**: schedules one DAG after another on the available resources. One DAG can only be scheduled to a resource after all tasks already scheduled on that resource. This algorithm uses the first strategy.
- **Gap search algorithm**: also uses the first strategy, scheduling DAGs independently. However, it searches for spaces between tasks already scheduled, thus a task from a DAG being scheduled can be executed before tasks already scheduled given that it does not interfere in their starting time.
- **Interleave algorithm**: uses the second strategy. The algorithm schedules pieces of each DAG in turns, interleaving their tasks in the schedule of the available resources.
- **Group DAGs algorithm**: uses the third strategy. Before scheduling the DAGs, the algorithm merge them into a single DAG. This sinlge DAG is scheduled, actually performing the scheduling of all DAGs that compose it.

In order to implement and evaluate these scheduling approaches, we must define a scheduling algorithm which prioritizes and selects the tasks to be scheduled, and also select the resources where these tasks will run. In this work, we used a modified version of the Path Clustering Heuristic (PCH) [6] to do this job, which is explained in the next section.

### 4.1 The PCH scheduling algorithm

Before describing the four strategies to schedule multiple workflows, we show the heuristic used by these four strategies to schedule the DAGs.

The Path Clustering Heuristic is a DAG scheduling heuristic which uses the clustering technique to generate groups (clusters) of tasks and the list scheduling technique to select tasks and processors. The PCH groups paths of the DAG and schedule them initially on the same resource, with the objective of reducing communication costs. This scheduling heuristic has shown good performance when communication between tasks (communication to computation ratio - CCR) is medium or high [7]. Workflows with such communication characteristics are commonly encountered in e-Science applications [47, 12].

The first step of the PCH algorithm is to compute, for each task, some attributes based on information given by the middleware and by the DAG specification. How this information is gathered and provided is out of the scope of this work. As most works on DAG scheduling in the literature, we consider that the programming model and/or the middleware can provide information about the size of each task to the scheduler. These sizes can be obtained by application benchmarks, by a history of executions/input data/data sizes, by estimatives given by the programmer, or by estimatives according to past execution and current data sizes.

The task attributes used by the algorithm are defined as follows.

- ***Weight (Computation Cost):***

$$w_i = \frac{instructions_i}{power_r}$$

$w_i$ represents the computation cost of the task $i$ in the resource $r$. $Power_r$ is the processing power of the resource $r$, in instructions per second.

- ***Communication Cost:***

$$c_{i,j} = \frac{data_{i,j}}{bandwidth_{r,t}}$$

$c_{i,j}$ represents the communication cost between tasks $i$ and $j$, using the link between the resources $r$ and $t$, where they are scheduled. If $r = t$ , $c_{i,j} = 0$.

- $succ(t_i)$ is the set of immediate successors of the task $t_i$.

- ***Priority:***

$$P_i = \begin{cases} w_i, & \text{if } i \text{ is the last task} \\ w_i + \max\limits_{t_j \in succ(n_i)} (c_{i,j} + P_j), & \text{otherwise} \end{cases}$$

$P_i$ is the priority level of the task $i$. Note that the priority attribute $P_i$ of a task $i$ represents the largest path starting in $t_i$ and ending in $t_{exit}$.

- ***Earliest Start Time:***

$$EST(t_i, r_k) = \begin{cases} Time(r_k), & \text{if } i = 1 \\ \max\{Time(r_k), EST_{pred}\}, & \text{otherwise} \end{cases}$$

$EST(t_i, r_k)$ represents the earliest start time of the task $i$ in resource $k$. $Time(r_k)$ is the time when the resource $k$ is ready for task execution, and $EST_{pred} = \max\limits_{t_h \in pred(t_i)} (EST_h + w_h + c_{h,i})$. Note that the EST of $t_i$ represents the largest path starting in $t_{entry}$ and ending in $t_i$.

- ***Estimated Finish Time:***

$$EFT(t_i, r_k) = EST(t_i, r_k) + \frac{instructions_i}{power_k}$$

$EFT(t_i, r_k)$ represents the estimated finish time of the task $i$ in the resource $k$.

To calculate the initial values of the attributes, we assume an homogeneous virtual system with an unbounded number of processors where each processor has the best capacity encountered in the real system and each link has the best bandwidth encountered in the real system.

With the initial values of the attributes calculated, the algorithm starts to create clusters of tasks to begin the scheduling process. The algorithm uses the priority attribute to select the first task to be added to the first cluster (clustering phase). The first node (or task) $t_i$ selected to compose a cluster $cls_k$ is the unscheduled node with the highest priority. It is added to $cls_k$, then the algorithm starts a depth-first search on the DAG starting on $t_i$, selecting $t_s \in succ(t_i)$ with the highest $P_s + EST_s$ and adding it to $cls_k$, until $t_s$ has a non scheduled predecessor. The task $t_s$ that has a non scheduled predecessor is not included in the $cls_k$. With this, clusters always have only tasks with all predecessors already scheduled or to be scheduled along with them.

For each cluster created, the algorithm selects a resource to schedule it. The processor selection step is performed after each clustering step. Thus, the algorithm creates a cluster, selects a processor to the created cluster, recalculates the nodes attributes and repeats these steps until all nodes are scheduled.

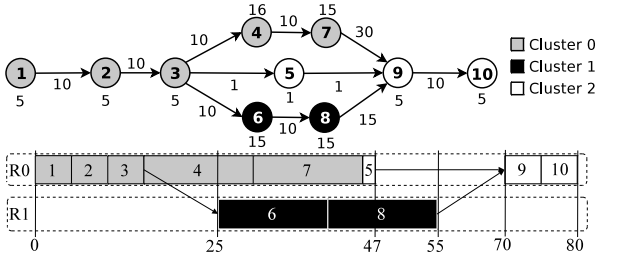


**Fig. 3** Example of scheduling using the modified PCH algorithm.

Hence, after creating each cluster the algorithm must decide where it will be scheduled. In the processor selection phase, the PCH looks for the resource which minimizes the EFT of the cluster. Thus, the criterion to choose the processor to a cluster $cls_k$ is to minimize the $EFT_{cls_k}$, defined as

$$EFT_{cls_k} = \max_{t_i \in cls_k} EFT_{t_i}$$

After scheduling each cluster, tasks attributes are recomputed. With this, the algorithm has a new view of the DAG and tasks priorities. The next tasks to be selected for the next cluster depends on how the current cluster was scheduled, since the ESTs of the tasks in any path from the current cluster to the exit node are affected by the new estimated finish time of the scheduled cluster.

Figure 3 shows an example of clusters created by PCH and the resulting scheduling. For the sake of simplicity, we consider in this example two resources with the same processing capacities. First, cluster 0 is created, where the depth-first search starts in task 1, going through tasks 2, 3, 4, 7, and, when it reaches task 9, the search stops and this task is not added to the cluster. Then this cluster is scheduled on resource 0. Cluster 1 is composed of tasks 6 and 8, and it is scheduled on resource 1, which results in the smallest EFT for task 8. Finally, cluster 2, composed of tasks 5, 9, and 10 is scheduled on resource 0.

The clustering and processor selection steps of PCH, as presented in this section, are used in the four algorithms for multiple workflow scheduling, which are presented in the next sections.

### 4.2 Sequential scheduling

The first and straightforward algorithm to schedule multiple DAGs is to schedule all DAGs in sequence using the PCH. This means that, given two DAGs $G_1$ and $G_2$, a task $t \in G_2$ can only be scheduled on a resource $r$ after all tasks of $G_1$ scheduled on $r$. Thus, the first available time on each resource after scheduling a DAG

$G$ is the EFT of the last task of $G$ scheduled on it. This approach is illustrated in Algorithm 1.

**Algorithm 1** Sequential scheduling

```
1: DAGs ⇐ Workflows to be scheduled.
2: for all G ∈ DAGs do
3:    while there are not scheduled tasks in G do
4:       cls ⇐ cluster of G to be scheduled
5:       r ⇐ resource selected by PCH for cls
6:       schedule the cluster cls on r
7:       Time(r) ⇐ EFT_cls
8:    end while
9: end for
```

This is the scheduling mechanism generally used when scheduling multiple DAGs (naturally, using different heuristics to make the schedule). The DAGs are scheduled in order of arrival, one after another, thus the scheduling is triggered by the event of an arrival of one DAG. Therefore, the scheduler does not modify the current schedule nor insert tasks of new processes among the already scheduled ones.

### 4.3 Gap search scheduling

An alternative to scheduling the DAGs sequentially, but yet scheduling one by one, is to use a gap search approach [6]. This method takes advantage of the gaps left between scheduled tasks, which are result of communication dependencies. These idle periods can be used to process tasks from other workflows, potentially reducing the overall makespan of all DAGs.

When the multiple workflow scheduling with gap search starts, if there are no other workflows scheduled in the available resources, there is no need to look for gaps in the schedule. On the other hand, if there are tasks currently assigned to one or more resources, the scheduling uses the gap searching algorithm to select a resource for each cluster.

Let $S_r = \{t_1, t_2, ..., t_k\}$ be the schedule (queue) of the resource $r$. We define the size of a cluster $c$ on $S_r$ as follows:

$$size_{c,r} = EFT(t_m^c, r) - EST(t_1^c, r).$$

In other words, the size of a cluster $c$ on a given resource $r$ is the difference between the EFT of the last task of the cluster and the EST of the first task of the cluster on $r$. The gap for the cluster $c = \{t_1^c, t_2^c, ..., t_m^c\}$ in the resource $r$ is defined as:

$$g_{c,r} = \min_{t_j \in S_r} (j)$$

such that $(EST(t_{j+1}, r) - EFT(t_j, r)) \times s\_margin > size_{c,r}$ and $EST(t_{j+1}, r) - EST(t_1^c, r) > size_{c,r}$.

The gap search has the objective of inserting another DAG in the current schedule without interfering or delaying the already scheduled DAGs. In the dynamic grid environment, the performance of the resources may vary during the process execution. The *security margin*, $s\_margin$, is a space left in the found gap to give room for possible losses of performance in the resources. With this, if the resource performance is worse than expected, the cluster inserted in the gap can execute with minor interference in the tasks previously scheduled on that resource. The security margin is relative to the size of the gap. For example, if we want 10% of security margin, then $s\_margin = 0.9$.

When the algorithm searches for gaps for a cluster $c$, it may encounter a gap which leads to a deadlock among tasks of the same workflow. This occurs if there exists a task $t_d$ which is on the path from any task in $c$ to the exit task such that $t_d$ is after the gap found in the considered resource. To avoid deadlocks, one verification is made when a gap is found. To accomplish this, after composing each cluster, the algorithm generates a set of dependent tasks for each task in the cluster. The set of dependent tasks of a task $t_i$, $D_{t_i}$, is composed of all tasks in any path from $t_i$ to $t_{exit}$ (the last task of the process). Before assigning a gap to a cluster $c$, the algorithm verifies if there are no tasks ahead of the gap in the schedule which $t_m^c$ (the last task of the cluster) depends on. If there is such a task, the gap cannot be assigned to that cluster, and the algorithm proceeds with the search. The gap searching algorithm is shown in Algorithm 2.

Algorithm 2 receives as input the cluster and the schedule of the resource currently being considered for the gap search. In the beginning (line 1), the algorithm computes the size of the cluster $c$ being scheduled. After that, it checks how many tasks already exist in the schedule of $r$ (line 2). In line 3 it verifies if the cluster being scheduled fits before the first task scheduled on the resource. This may occur if the first task on the resource's schedule has $EST > 0$ and if $c$ is smaller than this gap in the beginning of the schedule, considering the security margin. If the gap is sufficiently big to contain $c$, the algorithm verifies if there is room for the cluster discounting its EST, since we do not want to interfere on the execution of tasks already in the schedule. At the same time, in line 5, the algorithm verifies if inserting the cluster in the gap found results in a deadlock. If there is no room for the cluster in the beginning of the schedule or if there is deadlock, the algorithm iterates over the tasks on the schedule (line 10), looking for a gap between the task on the current position, $t_i$, and

**Algorithm 2** search_gap($S_r$, $c$)

**Input:** $S_r$: current schedule of resource $S_r$
$c$: cluster to be scheduled

1: $size_{c,r} \Leftarrow EFT(t^c_m, r) - EST(t^c_1, r)$
2: $k \Leftarrow$ number of tasks in $S_r$
3: **if** $(EST(t_1, r) \times s\_margin) > size_{c,r}$ **then**
4: Compute ESTs and EFTs for $t^c_j \in c$ on the current gap
5: $D_{t^c_m} \Leftarrow$ tasks ahead which $t^c_m$ depends on
6: **if** $(EST(t_1, r) - EST(t^c_1, r) > size_{c,r})$ and $D_{t^c_m} = \emptyset$ **then**
7: $g_{c,r} = 0$; **return** $g_{c,r}$
8: **end if**
9: **end if**
10: **for** $i = 1$ to $k - 1$ **do**
11: **if** $((EST(t_{i+1}, r) - EFT(t_i, r)) \times s\_margin) > size_{c,r}$ **then**
12: Compute ESTs and EFTs on current gap $\forall t^c_j \in c$
13: $D_{t^c_m} \Leftarrow$ tasks ahead which $t^c_m$ depends on
14: **if** $(EST(t_{i+1}, r) - EST(t^c_1, r) > size_{c,r})$ and $D_{t^c_m} = \emptyset$ **then**
15: $g_{c,r} = i$; **return** $g_{c,r}$
16: **end if**
17: **end if**
18: **end for**
19: $g_{c,r} = k$; **return** $g_{c,r}$ *//no gap found*

the next task, $t_{i+1}$ (lines 11 to 17). If no gaps are found, the algorithm returns the last position on the schedule of $r$ (line 19).

The gap search is done in the processor selection phase for every resource available (Algorithm 3), using the clusters generated by PCH. The algorithm searches for gaps on every resource, and it inserts the current cluster $c$ on the position returned by the gap searching algorithm (lines 2 and 3). The insertion is done after the task on the stated position, starting with $t_1 \in S_r$ in position 1. In line 4 the algorithm computes the EFT of the cluster $c$. Finally, the algorithm returns the resource with the smallest EFT for $c$ (line 6).

**Algorithm 3** get_best_resource($c$)

**Input:** $c$: cluster to be scheduled

1: **for all** $r$ in *resources* **do**
2: $g_{c,r} \Leftarrow$ search_gap($S_r$, $c$)
3: *schedule* $\Leftarrow$ Insert *cluster* on $S_r$ in position $g_{c,r}$
4: $time_r \Leftarrow$ calculate_EFT($t^c_m$);
5: **end for**
6: **return** resource $r$ with the smallest $time_r$

An example is shown in Figure 4, where numbers next to nodes and edges are computation and communication costs, respectively. Two processes, $P0 = \{t_1, ..., t_{10}\}$ and $P1 = \{t_A, ..., t_F\}$, are scheduled with gap searching. For the sake of simplicity, the example considers that resources $R0$ and $R1$ have same performance. The first cluster of $P0$ scheduled is $cls_{0,P0} = \{t_1, t_2, t_3, t_4, t_7\}$. Then $cls_{1,P0} = \{t_6, t_8\}$ is scheduled.

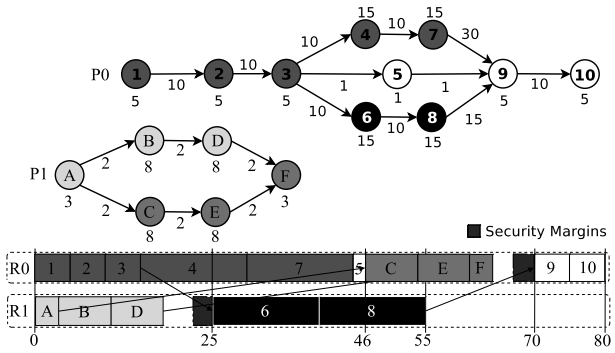


**Fig. 4** Two DAGs and the resulting schedule when using the gap searching algorithm.

Finally, $cls_{2,P0} = \{t_5, t_9, t_{10}\}$ is scheduled. After scheduling $P0$, $P1$ is scheduled, starting with $cls_{0,P1} = \{t_A, t_B, t_D\}$ on the gap found before $t_6$. Then, $cls_{1,P1} = \{t_C, t_E, t_F\}$ is scheduled on the gap between $t_5$ and $t_9$.

In the example, the cluster $cls_{1,P1} = \{t_C, t_E, t_F\}$ can be scheduled on the gap between $t_5$ and $t_9$ with a security margin of at most 20%, considering that $EST(t_C, R0) = 46$, $EFT(t_F, R0) = 65$, $EFT(t_5, R0) = 46$, and $EST(t_9, R0) = 70$. More precisely, this gap can be used if $s\_margin > \frac{19}{24}$, since $size(\{t_C, t_E, t_F\}, R0) = 19$ and $EST(t_9, R0) - EFT(t_5, R0) = 24$.

### 4.4 Interleave

A round robin algorithm is another option to schedule multiple DAGs. In this approach, a task (or group of tasks) of each DAG is selected to be scheduled at each round. After all DAGs have their group of tasks scheduled, a new group is selected and another round of scheduling starts. This way, the DAGs are scheduled in turns, and their tasks are interleaved in the available resources. The process interleaving can prioritize or not the already scheduled processes. This prioritization can be made considering process characteristics, such as tasks weights, communication costs, and size of the critical path.

In the multiple process scheduling scenario, as the processes arrive to be scheduled, the task selection step considers the not scheduled tasks of already scheduled processes and all tasks of the arriving processes. To perform this selection, this algorithm considers that the first workflow to arrive will be the first to have tasks selected to be scheduled, then the second process has its tasks selected, and so on. In short, the algorithm selects a cluster of tasks of each process in the order they arrived, scheduling each selected cluster until all processes have all tasks scheduled. These clusters are generated using PCH, as explained before. During the

scheduling by interleaving clusters, the algorithm also performs the gap search, trying to make a better use of the available resources. With the process interleaving, the spaces in the schedule originated by data transmission will be used to process other workflows. Algorithm 4, which is executed when a DAG $G$ arrives to be scheduled, shows this procedure.

**Algorithm 4** Interleaving overview

1: $CLS_{queue} \Leftarrow$ non executed clusters of scheduled processes
2: $DAGs \Leftarrow$ processes with tasks in $CLS_{queue}$.
3: $DAGs \Leftarrow \{DAGs\} \cup \{G\}$.
4: assign priorities to each graph in $DAGs$ according to a priority policy.
5: **while** there are unscheduled tasks **do**
6: $G \Leftarrow$ select next DAG with the highest priority
7: schedule $N_G$ clusters of $G$ using the gap search, being $N_G$ in accordance with the priority policy.
8: **end while**

In the first line, Algorithm 4 collects the clusters which were not sent to execution yet, while in line 2 it creates the group $DAGs$, which contains the processes where these clusters are part of. The process $G$, which is the last process that arrived to be scheduled, is added to the $DAGs$ group in line 3. In line 4, a pre-defined priority policy assigns to each process a priority level to be used when selecting processes and deciding how many clusters of each process will be scheduled. The iteration between lines 5 and 8 performs the scheduling, selecting the graphs in priority order (line 6) and schedule the clusters of the selected DAG(line 7), inserting the cluster in the queue of the resource which results in the smaller $EST$ for the successor of its last task.

The priority policy modifies the order that the processes are scheduled and the number of clusters scheduled for each process in each iteration of the algorithm. The value $N_G$ used in line 7 represents the number of clusters of the current process. This value is defined in the priority policy, and each DAG has its own $N_G$. Additionally, $N_G$ may vary during the scheduling process. For example, a graph which has executed most of its tasks may have its priority increased by the priority policy. Moreover, the priority policy may define if a DAG which is already scheduled can be rescheduled when new processes arrive, or the maximum number of workflows that can be rescheduled when a new process arrive.

In the experiments performed in this work, we used the FCFS (First Come, First Served) priority, with $N_G = 1 \; \forall \; G \; \in DAGs$. Thus, the scheduling of the clusters is performed in a round robin manner, starting in the first DAG that arrived and ending when there are no more unscheduled clusters. Note that the larger the cluster,

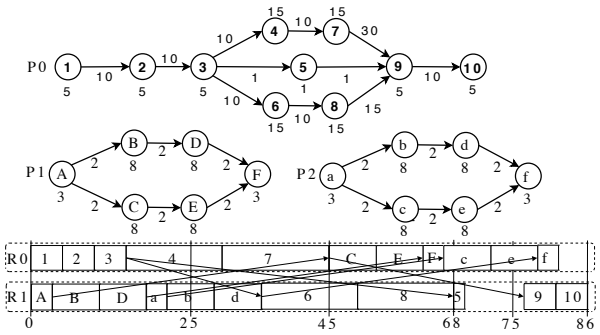


**Fig. 5** Example of scheduling using process interleaving.

the higher the number of scheduled tasks of the DAG. With this, the communication between tasks on the same cluster is suppressed, and small clusters are interleaved with clusters from other processes, hence having their communication times used to process tasks from other workflows.

Figure 5 shows an example of scheduling using the interleaving algorithm, where $P_0 = \{t_1, ..., t_{10}\}$, $P_1 = \{t_A, ..., t_F\}$, and $P_2 = \{t_a, ..., t_f\}$, with $P_2$ being another instance of $P_1$. To make the example simpler, we assumed that both resources have the same processing power. The first cluster to be scheduled is $cls_{1,P0} = \{1, 2, 3, 4, 7\}$. Then, clusters $cls_{1,P1} = \{A, B, D\}$ and $cls_{1,P2} = \{a, b, d\}$ are scheduled. Back to the first process, $cls_{2,P0} = \{6, 8\}$ is scheduled. After that, clusters $cls_{2,P1} = \{C, E, F\}$ and $cls_{2,P2} = \{c, e, f\}$ are scheduled. Finally, cluster $cls_{3,P0} = \{5, 9, 10\}$ is scheduled. Note that there is only one gap left between all tasks, consequently resulting in a small idle time in the resources.

While the gap searching algorithm does not interfere in the already scheduled processes, the interleaving allows mechanisms to prioritize workflows according to specific policies. For example, a process $P_h$ with high cost data dependencies may have a cluster scheduled first, leaving its communication gaps to be used by other process in the interleaving. Or, if a DAG has only few clusters left to be scheduled, it can have its priority increased to have its execution finished soon.

### 4.5 Group DAGs

Besides the strategy of scheduling multiple workflows considering them as separate entities, another approach is to merge all DAGs into a single one by creating one entry and one exit node and connecting them to all DAGs. The new entry node has cost 0, as well as its edges, which are connected to each entry node of the DAGs being scheduled. Similarly, the new single exit node has cost 0, and it has edges coming from all

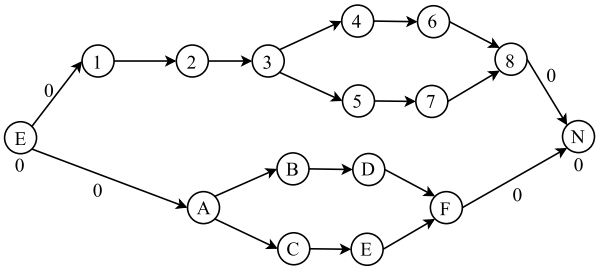


**Fig. 6** Example of grouping two DAGs into a single one by adding an entry and an exit node.

exit nodes of the DAGs being scheduled. Thus, a single DAG is created, and then scheduled onto the resources available. Consider two processes, $P0 = \{t_1, ..., t_8\}$ and $P1 = \{t_A, ..., t_F\}$. The resulting DAG after merging $P0$ and $P1$ is shown in Figure 6, where task $t_E$ is the new entry node with cost 0 and task $t_N$ is the new exit node with cost 0. The added edges also have cost 0, while the existing tasks and edges remain with their original cost.

Algorithm 5 shows the group scheduling approach.

**Algorithm 5** Group DAGs

1: $DAGs \Leftarrow$ workflows to be scheduled.
2: create DAG $G_{group}$
3: insert new costless tasks $t_{entry}$ and $t_{exit}$ in $G_{group}$.
4: **for all** $G \in DAGs$ **do**
5: create costless edge $(t_{entry}, t_{G_0})$
6: create costless edge $(t_{exit}, t_{G_n})$
7: **end for**
8: **while** there are not scheduled tasks in $G_{group}$ **do**
9: $cls \Leftarrow$ cluster of $G_{group}$ to be scheduled
10: schedule the cluster $cls$ in the resource selected by PCH
11: **end while**

The input of the algorithm is a group of workflows to be scheduled ($DAGs$). The first step of the algorithm is to create a new DAG ($G_{group}$), which will be the DAG composed of all workflows in $DAGs$. To accomplish this, the algorithm inserts the costless tasks $t_{entry}$ and $t_{exit}$ to this new DAG (line 3). After that, in the loop within lines 4 and 7, each entry node of each existing DAG is connected to the just created entry node $t_{entry}$. On the other hand, each exit node of the existing DAGs is connected to $t_{exit}$. After creating $G_{group}$, the algorithm proceeds with the scheduling using the PCH.

## 5 Simulation results

We evaluated the four described strategies for scheduling multiple workflows. The simulations focused on how these strategies perform when scheduling up to 10 workflows (or processes), named $P_0$ to $P_9$, on a grid environment considering the number of resources and the execution in conjunction with external load (i.e., user processes not managed by the grid). Thus, we analyzed the algorithms performance with variations in both the number of workflows and the number of resources, which are expected to vary in a real grid environment. Such information is important to evaluate the behaviour of each strategy in what concerns the load dynamicity and scalability. The results are split into *initial schedule results* (makespan after the scheduling is finished) and *execution time results* (makespan after execution with external load).

The simulations were run considering the group topology described in Section 2. The number of resources in each group was randomly taken between 1 and 10, and we show the simulation results for 2, 10 and 25 groups. Each resource had its processing capacity randomly taken from the interval $(50, 200)$. Link bandwidths were taken from the interval $(40, 80)$ for pairs of resources inside the same group, and from the interval $(5, 40)$ for pairs of resources of different groups. This is to ensure that bandwidth between resources of different groups has never been larger than the bandwidth of the same resources inside their groups. Inside each group, all nodes communicate directly, with each group being a fully connected graph. We show simulations using the PCH algorithm and some simulations using the HEFT algorithm [48].

We evaluated how the algorithms performed in what concerns the makespan and slowdown. The makespan allows the evaluation of which algorithm results in better schedules in the sense of the duration of the workflows execution, while the slowdown metric allows the comparison of the fairness in the resulting schedule. The metrics used are the average makespan, the slowdown, and the overall makespan, defined as follows:

***Average makespan*** *($average_{M_N}$):* average makespan for the first $N$ processes after scheduling all the 10 processes according to the process quantity $N$ considered, $1 \leq N \leq 10$, defined as:

$$average_{M_N} = \frac{1}{N} \sum_{k=0}^{N-1} makespan_{P_k}$$

Note that the average makespan considers the average makespan over the first $N$ workflows scheduled, but with all workflows already scheduled. For example, for $N = 3$ we consider the average makespan for processes 1, 2 and 3 with all 10 workflows scheduled.

***Slowdown:*** Slowdown of each process after scheduling all the 10 workflows. The slowdown of a process $P_k$ means how many times the resulting makespan of $P_k$ is

larger than the makespan of $P_k$ in its initial schedule if it was scheduled alone on the same set of resources.

***Overall makespan*** *($overall_M$):* The overall makespan of the schedule, calculated with all processes already scheduled, is defined as:

$$overall_M = \max_{P_k \in DAGs} makespan_{P_k}$$

For the overall makespan metric we show results with $2, 3, 4, 5, 6, 7, 8, 9, 10, 15$, and 25 groups.

Sixteen DAGs were taken for the simulations, where fifteen were randomly generated with number of nodes varying from 7 to 82, and the other one was a CSTEM DAG (Coupled Structural Thermal Electromagnetic Analysis and Tailoring of Graded Composite Structures). The computation cost of each task was randomly taken from the interval $(5000, 11000)$, while the communication costs where taken from the interval $(500, 1100)$. For the CSTEM DAG, the weights were randomly generated but with values proportional to those encountered in the original workflow [20]. All random numbers were taken from a uniform distribution. The results show averages over 500 executions and show a confidence interval of 95%. Additionally, the gap searching algorithm was run with $s\_margin = 0.95$. The processes are numbered in order of arrival, i.e., $P_0$ is the first process received by the scheduler, while $P_9$ is the last one.

### 5.1 Initial schedule

This section shows the results for the initial schedule, which is the schedule provided by the algorithms considering the attributes computed using the capacities of resources and links as well as the costs of tasks and dependencies received by the scheduler in conjunction with the DAG. Therefore, the initial schedule results do not consider the execution of the workflow tasks, but only the makespan given by the scheduler initially.

Figures 7, 8, and 9 show the $average_{M_N}$ for the first $N$ processes, $N$ ranging from 1 to 10.

For 2 groups the gap searching algorithm results in lower average initial schedule when considering $N$ from 2 to 10. The interleaving algorithm performs better than the group algorithm for all $N$, and it approximates the results of the gap search as higher is the number of processes considered. We can observe that the sequential and gap search algorithms makespans increase as later the process was scheduled, with the gap search algorithm performing better than the sequential one. On the other hand, the interleaving and the grouping algorithms result in more balanced makespans, showing more fair behavior.

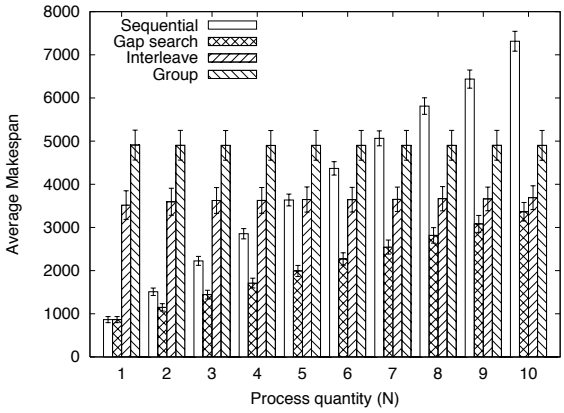


**Fig. 7** Average makespan ($averageM_N$) according to the number of processes in the initial schedule with 2 groups of resources.

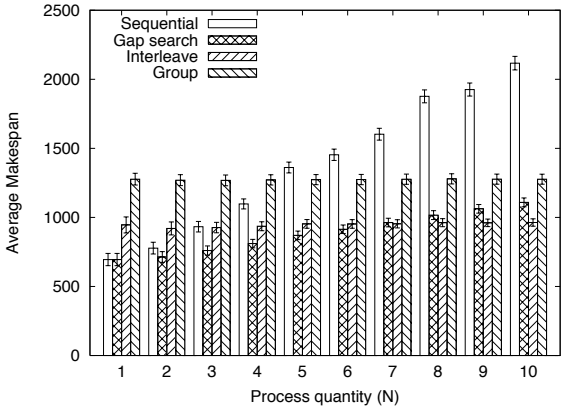


**Fig. 8** Average makespan ($averageM_N$) according to the number of processes in the initial schedule with 10 groups of resources.

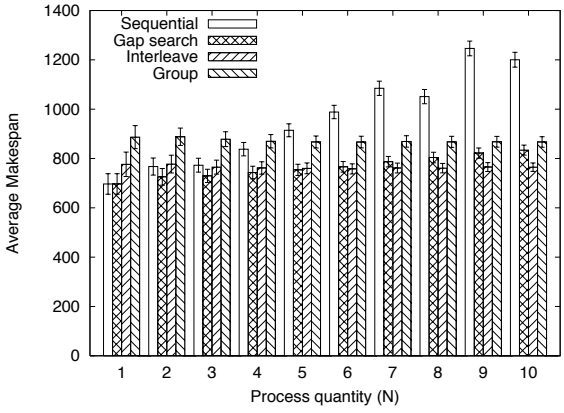


**Fig. 9** Average makespan ($averageM_N$) according to the number of processes in the initial schedule with 25 groups of resources.

For 10 groups (Figure 8) the interleaving algorithm results in lower averages than the gap search if we consider 7 or more processes, and the interleaving stills performing better than the group approach.

The tendency continues for 25 groups (Figure 9). The interleaving algorithm starts to give better ma-

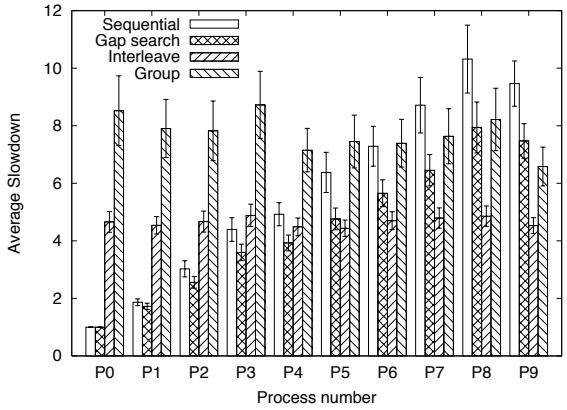


**Fig. 10** Slowdown of each process in the initial schedule with 2 groups of resources.

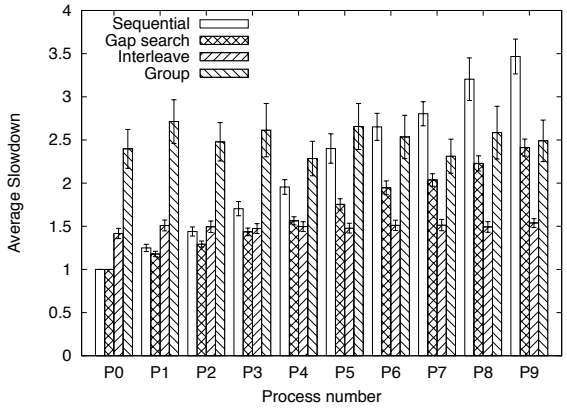


**Fig. 11** Slowdown of each process in the initial schedule with 10 groups of resources.

kespans with more than 4 processes. Also, the difference among the results of the algorithms is smaller, since there are more resources to spread the processes tasks, thus the sequential and gap search algorithms achieve makespans which are closer to the interleave makespans.

The slowdown results for each process with 2, 10, and 25 groups are shown respectively in Figures 10, 11, and 12.

For 2 groups of resources there are noticeable variations in the slowdown of different processes for all algorithms, except for the interleave approach, which maintains the slowdown of all processes in the same level. This means that the interleaving algorithm can result in a more fair schedule when considering the slowdown metric.

The same pattern is observed when there are 10 groups of resources. The interleaving algorithm schedules all processes in a manner that they have really close slowdown values. In this scenario, the grouping approach results in slowdowns closer to each other, but higher than that ones given by the process interleave.

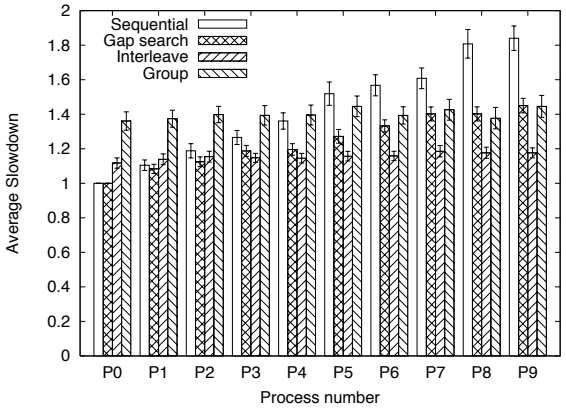


**Fig. 12** Slowdown of each process in the initial schedule with 25 groups of resources.

As for 2 and 10 groups, for 25 groups the interleaving algorithm results in uniform slowdown for all processes, and it remains close to 1.0, i.e., a low slowdown. The grouping algorithm also results in uniform slowdowns, but, again, higher than that ones given by the interleaving approach. To summarize the slowdown results, we show the Jain's fairness index [36] for the initial schedule in Table 1, where the Jain's fairness index for a set of values $X = \{x_1, x_2, ..., x_n\}$ is defined as follows:

$$J_f = \frac{(\sum_{i=1}^{n} x_i)^2}{n \sum_{i=1}^{n} x_i^2}$$

The Jain's fairness index ranges from $\frac{1}{n}$ (unfair) to 1 (fair). We can confirm that the interleave and group algorithms are the more fair ones, with the interleave being slightly more fair. Additionally, the interleave is the algorithm which shows less variation in the Jain's fairness index with the variation on the number of groups.

**Table 1** Jain's fairness index for the average slowdowns in the initial schedule

| | **Number of groups** | | |
|---|---|---|---|
| **Algorithm** | **2** | **10** | **25** |
| **Sequential** | 0.778268 | 0.881259 | 0.964416 |
| **Gap search** | 0.800188 | 0.934998 | 0.986881 |
| **Interleave** | 0.998989 | 0.999576 | 0.999722 |
| **Group** | 0.993800 | 0.997085 | 0.999609 |

Figure 13 shows the average overall makespan (the average maximum makespan over all processes) for 2, 10, and 25 groups. We can observe that the interleave and group algorithms result in smaller overall sched-

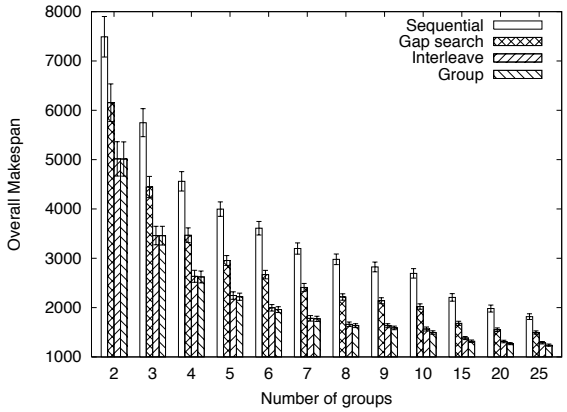


**Fig. 13** Maximum makespan over all processes in the initial schedule.

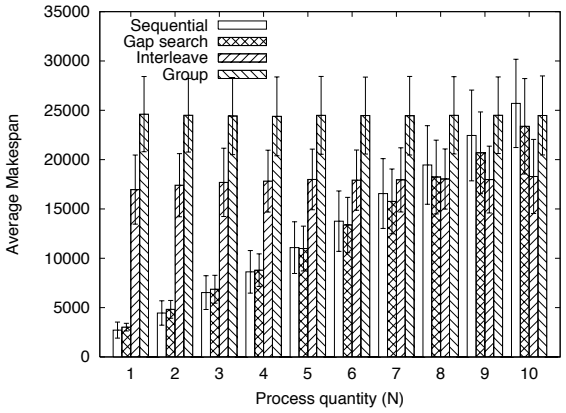


**Fig. 14** Average makespan ($averageM_N$) according to the number of processes after execution with 2 groups of resources.

ules than the ones provided by the sequential and gap searching algorithms.

The group and interleave approaches result in equivalent overall makespans, but, as shown in the slowdown results, the interleaving algorithm results in more fair schedules. This is because the group algorithm treats all DAGs as a single one, thus smaller processes, which have smaller paths, tend to be scheduled later because their tasks receive low priorities when calculating the tasks attributes. This behavior yields on higher slowdown for such processes, since when scheduled alone they have low makespans. This does not happen with the interleaving algorithm because tasks from smaller processes are taken in the same frequency as that from larger processes. Therefore, the scheduling of smaller processes finishes before, while larger processes remain being scheduled and interleaved. This leads to a more fair behavior, resulting in similar slowdowns for both small and big workflows. Furthermore, the interleave algorithm has shown to be scalable, and it still result in good schedules with higher number of groups.

## 5.2 Execution results

Besides scalability, another important point to evaluate when dealing with grid computing algorithms is how they behave when external load exists. In this section we evaluate the same metrics used hitherto, but for a simulated execution in a shared environment. To simulate external load, we assumed that external jobs arrival follows a Poisson distribution, while each job lifetime is $2.0/x$ [2], where $x$ is a random number between 0 and 1. This follows process behavior observed in real-life applications in [33].

Figures 14, 15, and 16 show the resulting execution time after the simulation with load independent from the grid. In general, we can observe the same pattern as that encountered in the initial schedule results.

We can observe a better performance of the sequential algorithm when compared to the other while there are few processes. This is because there is less interference on processes that arrive early from process scheduled after them, and losses of performance can be absorbed by the gaps in the schedule. For example, when comparing the sequential and gap searching algorithms after the execution with 2 groups of resources (Figure 14), there is a difference to the results for the initial schedule. After the execution, the gap search performance remains close to the performance of the sequential algorithm. This is due to the delay of tasks which filled the gaps of those processes, which are finishing after the estimated finish time plus the security margin. This suggests that in a situation of high external load with few resources available, the gap searching cannot take much advantage of the gaps to execute other processes.

The same reasoning can be applied to the other algorithms: since there is no free space in the schedule, the delay in tasks reflects on other tasks because there is no space to absorb the delays. However, the interleaving approach remains resulting in lower average makespan than the other algorithms when multiple processes are executed, including lower average makespan than the gap searching when we consider 8 or more processes.

When there are more groups of resources (10 and 25, in Figures 15 and 16), the sequential algorithm still increases the average makespan less than the other algorithms when compared to the initial schedule results. We can also note that the interleaving algorithm does not maintain its better schedules when there are more than 4 processes. This is because the interleaving can use the best resources to execute more tasks, since it leaves less gaps in the schedule than the other

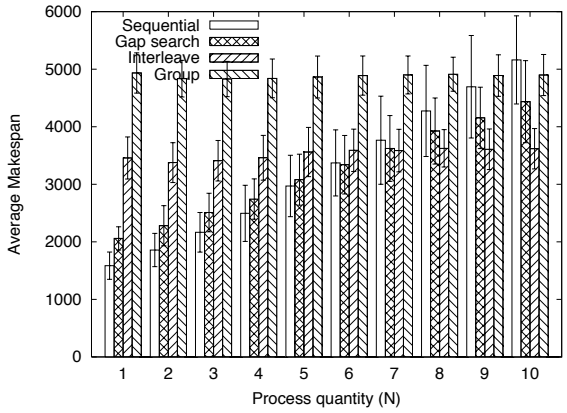


**Fig. 15** Average makespan ($averageM_N$) according to the number of processes after execution with 10 groups of resources.

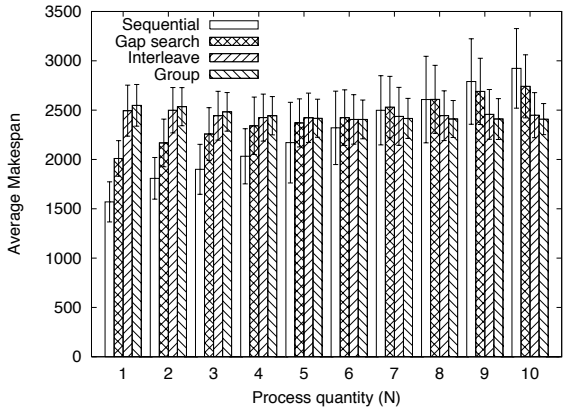


**Fig. 16** Average makespan ($averageM_N$) according to the number of processes after execution with 25 groups of resources.

approaches. Therefore, the delays are spread over the schedule, while in the other algorithms the tasks are scattered in more resources and a delay in one task has less impact in tasks of other processes.

The slowdown metric for simulations with 2 to 25 groups are shown in Figures 17, 18, and 19, respectively. These figures show that the interleaving approach stills being fair after the execution, also giving lower slowdown values when compared to the other approaches for most of the processes.

To summarize the fairness results after the simulation with external load, we show the Jain's fairness index for the slowdowns after the execution in Table 2. We observe that the interleave and group algorithms still show a more fair result, with the Jain's fairness index being very close to that observed in the initial schedule (Table 1). On the other hand, the sequential and gap search show a worsening in the Jain's index fairness when compared to the initial schedule results. One explanation to this behavior is that the sequential and gap search algorithms tend to accumulate tasks from the same workflows in a smaller space in the schedule

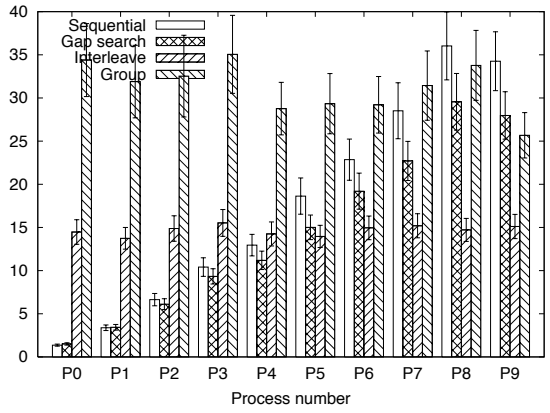


**Fig. 17** Slowdown of each process after execution with 2 groups of resources.

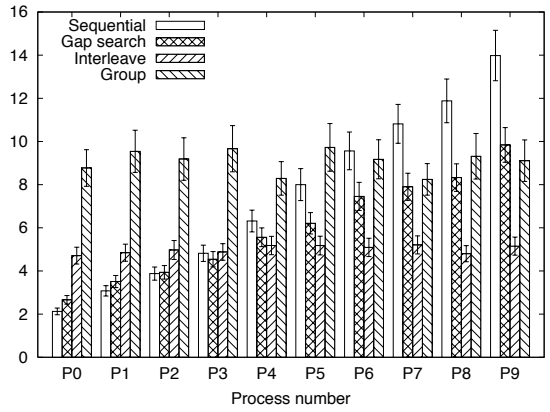


**Fig. 18** Slowdown of each process after execution with 10 groups of resources.

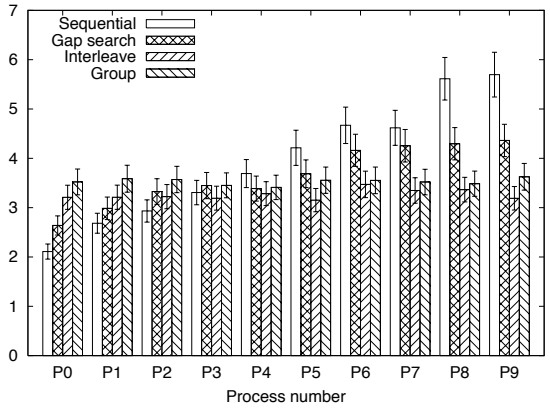


**Fig. 19** Slowdown of each process after execution with 25 groups of resources.

than the other two algorithms. With this, tasks from the same workflow stay closer in the schedule. Therefore, load peaks in the resources could potentially harm more tasks from the same workflow, leaving the schedule more unfair.

Figure 20 shows the maximum overall makespan after the execution of the workflows. We can observe the

**Table 2** Jain's fairness index for the average slowdowns after execution

| | Number of groups | | |
|---|---|---|---|
| **Algorithm** | **2** | **10** | **25** |
| **Sequential** | 0.682643 | 0.790583 | 0.921262 |
| **Gap search** | 0.703943 | 0.878235 | 0.976378 |
| **Interleave** | 0.998656 | 0.998800 | 0.999145 |
| **Group** | 0.992068 | 0.997034 | 0.999698 |

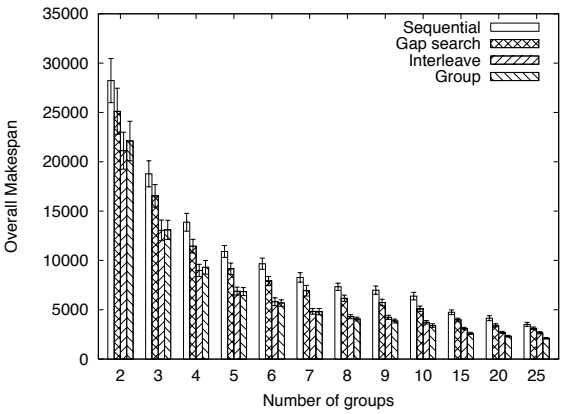


**Fig. 20** Maximum makespan over all processes after execution.

same pattern observed before the execution of the workflows.

Although there is a difference in terms of absolute values of makespan, which is inevitable with the external load sharing CPU time, in relative terms the execution results shown only a few variations from the initial schedule results. This suggests that both the PCH and the presented strategies are not significantly affected by the external load, being able to maintain their schedules as good as they were before the tasks execution.

## 5.3 Evaluation using HEFT

To have more confidence in the results shown and conclusions drawn, we present in this section some evaluations using the HEFT heuristic [48] as the scheduling algorithm, instead of the PCH. Figure 21 shows the slowdown results using HEFT as the heuristic for 10 groups of resources in the initial schedule, whiles Figure 22 shows the slowdown using HEFT for 10 groups of resources after execution. We can observe the same pattern shown by the PCH in Figure 11. These results reinforce the conclusions achieved when using the PCH heuristic.

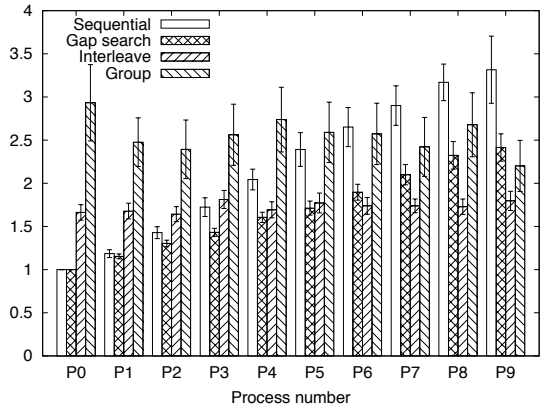


**Fig. 21** Slowdown of each process in the initial schedule with 10 groups of resources using the HEFT heuristic.

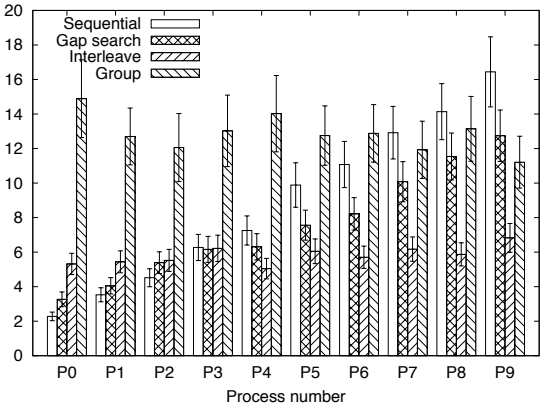


**Fig. 22** Slowdown of each process after execution with 10 groups of resources using the HEFT heuristic.

## 5.4 Discussion

As a general contribution of the presented results, we are able to conclude from these simulations that the interleaving approach is the best option when there are more than 4 processes to be scheduled and the number of resources is not low. When the number of resources is low (2 groups of resources, in the results shown), the gap searching resulted in good average makespan, thus it can be a good choice in this scenario. However, the gap searching alone seems to be more affected by external load than the sequential scheduling, with the later getting better than the former after the execution when there are more than 6 processes in 2 groups of resources. Table 3 gives a summary of the results observed for the slowdown and average makespan for each algorithm.

Although the sequential scheduling results in worse average makespans than the other algorithms when we consider the external load, we can observe that it is the less affected algorithm, since its averages are proportionally closer to the ones from the other algorithms than in the initial schedule. This is due to the spaces

**Table 3** Summary of the results.

| Algorithm | Average makespan | Fairness |
|---|---|---|
| **Sequential** | Lower for the first workflows, higher for the later workflows. Less affected by external load because of the gaps left in the schedule. | No. The first workflows have lower slowdowns. |
| **Gap search** | Lower for the first workflows, higher for the later workflows. Better performance with small number of resources. | No. The first workflows have lower slowdowns. |
| **Interleave** | All workflows have similar makespans. Scalable, mantains the good schedules and fairness with variations in the number of resources. | Yes. Results in similar slowdowns. for all workflows. |
| **Group** | All workflows have similar makespans. Smaller workflows may be scheduled late, affecting their slowdown. | Yes. Workflows have slightly different slowdowns. |

available between tasks in the sequential schedule, which can absorb the delays of tasks.

The four strategies evaluated, namely sequential scheduling, interleaving, grouping, and gap searching, shown assorted behaviors when we look at the makespan of different number of processes. In the initial schedule results we can observe that the sequential scheduling provides total advantage to the first process to be scheduled, as the gap searching also does but with better makespan because it uses communication gaps to execute tasks from other processes. On the other hand, the grouping and interleaving approaches result in worse makespans for the first processes, but with improvements on the makespan of the last processes. On average, this results in a better use of the resources, since the overall makespan of interleaving and grouping algorithms are lower than the overall makespan of gap searching and sequential algorithms. Additionally, the grouping and interleaving approaches are able to produce more fair schedules than the other two approaches. In terms of slowdown, the interleaving approach shown the best results, achieving lower slowdown in most cases.

## 6 Conclusion

We presented a study on how four strategies for scheduling multiple workflows on grids perform in terms of schedule length and fairness. This evaluation was made considering the initial makespan given by each algorithm and the final makespan after executing the workflows in a shared grid. The importance of such study on grids comes from the fact that they are shared environments and will eventually run more than one workflow at the same time. Furthermore, this topic has been neglected by most studies in scheduling, and only a few initial works exist [53, 6].

Further work in multiple workflow scheduling is undoubtedly necessary. Following the path of this work, the development of a prioritization scheme to work with the interleaving approach is an interesting topic. Processes already scheduled may have their priority changed dynamically to avoid starvation or to keep a good relation between fairness and execution time. Therefore, a dynamic and adaptive priority approach is an untouched problem which could be useful in grids executing multiple workflows at the same time. Additionally, the re-scheduling of multiple workflows is an interesting topic.

**Acknowledgements** The authors would like to thank FAPESP (05/59706-3), CAPES and CNPq for the financial support.

## References

1. L. Adzigogov, J. Soldatos, and L. Polymenakos. EMPEROR: An OGSA grid meta-scheduler based on dynamic resource predictions. *Journal of Grid Computing*, 3(1-2):19–37, 2005.
2. Y. Amir, B. Awerbuch, A. Barak, R. S. Borgstrom, and A. Keren. An opportunity cost approach for job assignment in a scalable computing cluster. *IEEE Transaction on Parallel and Distributed Systems*, 11(7):760–768, 2000.
3. J. Annis, Y. Zhao, J. Voeckler, M. Wilde, S. Kent, and I. Foster. Applying Chimera virtual data concepts to cluster finding in the Sloan Sky Survey. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 1–14, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.
4. R. Bajaj and D. P. Agrawal. Improving scheduling of tasks in a heterogeneous environment. *IEEE Transactions on Parallel and Distributed Systems*, 15(2):107–118, 2004.
5. D. M. Batista, N. L. S. da Fonseca, and F. K. Miyazawa. A set of schedulers for grid networks. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 209–213, Seul, Korea, 2007. ACM Press.
6. L. F. Bittencourt and E. R. M. Madeira. Fulfilling task dependence gaps for workflow scheduling on grids. In *3rd IEEE International Conference on Signal-Image Technology and Internet Based Systems*, Shanghai, China, 2007.
7. L. F. Bittencourt and E. R. M. Madeira. A performance-oriented adaptive scheduler for dependent tasks on grids. *Concurrency and Computation: Practice and Experience*, 20(9):1029–1049, 2008.
8. C. Boeres, J. V. Filho, and V. E. F. Rebello. A cluster-based strategy for scheduling task on heterogeneous processors. In

*16th Symposium on Computer Architecture and High Performance Computing, Foz do Iguacu, Brazil*, pages 214–221. IEEE, 2004.
9. J. Cao, S. A. Jarvis, S. Saini, and G. R. Nudd. Gridflow: Workflow management for grid computing. In *3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003), Tokyo, Japan*, May 2003.
10. H. Casanova, D. Zagorodnov, F. Berman, and A. Legrand. Heuristics for scheduling parameter sweep applications in grid environments. In *HCW '00: Proceedings of the 9th Heterogeneous Computing Workshop*, page 349, Washington, DC, USA, 2000. IEEE Computer Society.
11. H. Chen and M. Maheswaran. Distributed dynamic scheduling of composite tasks on grid computing systems. In *11th IEEE Heterogeneous Computing Workshop*, pages 119–128, Washington, DC, USA, 2002. IEEE Computer Society.
12. C.-M. W. Chun-Ho Liu and D. Y. C. Leung. Performance analysis of a linux pc cluster using a direct numerical simulation of fluid turbulence code. *International Journal of High Performance Computing Applications*, 19(4):365–374, 2005.
13. F. R. L. Cicerre, E. R. M. Madeira, and L. E. Buzato. A hierarchical process execution support for grid computing. *Concurrency and Computation: Practice and Experience*, 18(6):581–594, 2006.
14. K. Cooper, A. Dasgupta, K. Kennedy, et al. New grid scheduling and rescheduling methods in the grads project. In *IPDPS Next Generation Software Program - NSFNGS - PI Workshop*, pages 199,206. IEEE Computer Society, 2004.
15. R. C. Corrêa, A. Ferreira, and P. Rebreyend. Integrating list heuristics into genetic algorithms for multiprocessor scheduling. In *SPDP '96: Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing (SPDP '96)*, page 462, Washington, DC, USA, 1996. IEEE Computer Society.
16. F. de Oliveira Lucchese, E. J. H. Yero, F. S. Sambatti, and M. A. A. Henriques. An adaptive scheduler for grids. *Journal of Grid Computing*, 4(1):1–17, 2006.
17. E. Deelman, C. Kesselman, G. Mehta, L. Meshkat, L. Pearlman, K. Blackburn, P. Ehrens, A. Lazzarini, R. Williams, and S. Koranda. GriPhyN and LIGO, Building a Virtual Data Grid for Gravitational Wave Scientists. In *HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, page 225, Washington, DC, USA, 2002. IEEE Computer Society.
18. E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.
19. Y. Derbal. Entropic grid scheduling. *Journal of Grid Computing*, 4(4):373–394, 2006.
20. A. Dogan and F. Özgüner. Biobjective scheduling algorithms for execution time-reliability trade-off in heterogeneous computing systems. *Computer Journal*, 48(3):300–314, 2005.
21. F. Dong and S. G. Akl. Scheduling algorithms for grid computing: state of the art and open problems. Technical report, Queen's University School of Computing, Kingston, Canada, jan. 2006.
22. B. Duran and F. Xhafa. The effects of two replacement strategies on a genetic algorithm for scheduling jobs on computational grids. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 960–961, Dijon, France, 2006. ACM.
23. H. El-Rewini, H. H. Ali, and T. G. Lewis. Task scheduling in multiprocessing systems. *IEEE Computer*, 28(12):27–37, 1995.
24. I. Foster. Globus toolkit version 4: Software for service oriented systems. *IFIP International Conference on Network and Parallel Computing*, pages 2–13, 2005.
25. I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organization. *The International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.
26. J. Frey. Condor DAGMan: Handling inter-job dependencies. http://www.cs.wisc.edu/condor/dagman/, 2002.
27. N. Fujimoto and K. Hagihara. Near-optimal dynamic task scheduling of precedence constrained coarse-grained tasks onto a computational grid. In *2nd International Symposium on Parallel and Distributed Computing, Slovenia*, pages 80–87. IEEE, oct 2003.
28. N. Fujimoto and K. Hagihara. A comparison among grid scheduling algorithms for independent coarse-grained tasks. In *Symposium on Applications and the Internet Workshops, January 2004, Tokyo, Japan*, pages 674–680. IEEE Computer Society, 2004.
29. A. Goldchleger, F. Kon, A. Goldman, M. Finger, and G. C. Bezerra. InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines. *Concurrency and Computation: Practice and Experience*, 16(5):449–459, March 2004.
30. M. Grajcar. Genetic list scheduling algorithm for scheduling and allocation on a loosely coupled heterogeneous multiprocessor system. In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 280–285, New York, NY, USA, 1999. ACM Press.
31. T. Hagras and J. Janeček. An approach to compile-time task scheduling in heterogeneous computing systems. In *33rd International Conference on Parallel Processing Workshops*, pages 182–189. IEEE Computer Society, 2004.
32. M. Hakem and F. Butelle. Dynamic critical path scheduling parallel programs onto multiprocessors. In *19th International Parallel and Distributed Processing Symposium*. IEEE Computer Society, 2005.
33. M. Harchol-Balter and A. B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems*, 15(3):253–285, 1997.
34. X. He, X. Sun, and G. von Laszewski. Qos guided min-min heuristic for grid task scheduling. *J. Comput. Sci. Technol.*, 18(4):442–451, 2003.
35. E. S. H. Hou, N. Ansari, and H. Ren. A genetic algorithm for multiprocessor scheduling. *IEEE Trans. Parallel Distrib. Syst.*, 5(2):113–120, 1994.
36. R. Jain, D. Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Technical Report TR-301, DEC Research, September 1984.
37. Y.-K. Kwok and I. Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):506–521, 1996.
38. Y.-K. Kwok and I. Ahmad. Benchmarking the task graph scheduling algorithms. In *IPPS/SPDP*, pages 531–537, 1998.
39. M. J. Litzkow, M. Livny, and M. W. Mutka. Condor: A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems, USA*, pages 104–111, jun., 1988.
40. C. Phillips, C. Stein, and J. Wein. Task scheduling in networks. *SIAM Journal on Discrete Mathematics*, 10(4):573–598, 1997.
41. M. L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer Publishing Company, Incorporated, 2008.
42. R. Prodan and T. Fahringer. Dynamic scheduling of scientific workflow applications on the grid: a case study. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 687–694, New York, NY, USA, 2005. ACM Press.

43. M. Rahman, S. Venugopal, and R. Buyya. A dynamic critical path algorithm for scheduling scientific workflow applications on global grids. In *E-SCIENCE '07: Proceedings of the Third IEEE International Conference on e-Science and Grid Computing (e-Science 2007)*, pages 35–42, Washington, DC, USA, 2007. IEEE Computer Society.
44. R. Sakellariou and H. Zhao. A hybrid heuristic for dag scheduling on heterogeneous systems. In *13th Heterogeneous Computing Workshop*, pages 111,123. IEEE Computer Society, 2004.
45. I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, editors. *Workflows for e-Science. Scientific Workflows for Grids*. Springer Verlag, 2007.
46. D. Thain, T. Tannenbaum, and M. Livny. Condor and the grid. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., dec., 2002.
47. K. B. Theobald, R. Kumar, G. Agrawal, G. Heber, R. K. Thulasiram, and G. R. Gao. Developing a communication intensive application on the earth multithreaded architecture. In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 625–637, London, UK, 2000. Springer-Verlag.
48. H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.
49. B. A. Vianna, A. A. Fonseca, N. T. Moura, L. T. Menezes, H. A. Mendes, J. A. Silva, C. Boeres, and V. E. F. Rebello. A tool for the design and evaluation of hybrid scheduling algorithms for computational grids. In *Proceedings of the 2nd workshop on Middleware for grid computing*, pages 41–46, New York, NY, USA, 2004. ACM Press.
50. T. Yang and A. Gerasoulis. Dsc: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, 1994.
51. J. Yu and R. Buyya. A taxonomy of scientific workflow systems for grid computing. *SIGMOD Records*, 34(3):44–49, 2005.
52. S. Zhang, Y. Zong, Z. Ding, and J. Liu. Workflow-oriented grid service composition and scheduling. In *International Symposium on Information Technology: Coding and Computing, v. 2, April 2005, Las Vegas, Nevada, USA*, pages 214–219. IEEE Computer Society, 2005.
53. H. Zhao and R. Sakellariou. Scheduling multiple dags onto heterogeneous systems. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, Rohdes Island, Greece, 2006. IEEE.
54. Y. Zhao, J. Dobson, I. Foster, L. Moreau, and M. Wilde. A notation and system for expressing and executing cleanly typed workflows on messy scientific data. *SIGMOD Records*, 34(3):37–43, 2005.
55. Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, reliable, loosely coupled parallel computation. In *IEEE Congress on Services*, pages 199–206, Los Alamitos, CA, USA, 2007. IEEE Computer Society.