FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

QoS based workflow scheduling on heterogeneous resources

Hamid Arabnejad

MAP-i Doctoral Program in Computer Science

Supervisor: Jorge Manuel Gomes Barbosa

May 22, 2016

QoS based workflow scheduling on heterogeneous resources

Hamid Arabnejad

MAP-i Doctoral Program in Computer Science

May 22, 2016

Resumo

Os sistemas computacionais heterogéneos permitiram à comunidade científica um maior acesso ao processamento e análise de dados das aplicações científicas, representadas em muitas aplicações como *workflows*. Para obter um bom desempenho dos sistemas computacionais na execução destes problemas, o mapeamento de tarefas a recursos e o seu agendamento, são operações essenciais e cada vez mais exigentes. A execução eficiente de aplicações científicas pode ser obtida por um mapeamento e agendamento ótimo das tarefas aos elementos de processamento. O problema de mapeamento e agendamento é ainda mais complexo quando são considerados parâmetros de Qualidade de Serviço (QoS) definidos pelo utilizador. Esta classe de problemas é conhecida como sendo *NP-Completo*. Assim, uma parte substancial da investigação nesta área propõe algoritmos meta-heurísticos e de pesquisa de domínio que permitem controlar a qualidade das soluções geradas. Contudo, estas abordagens necessitam de tempos de processamento mais elevados de modo a obterem bons resultados, muitas vezes próximo do ótimo, tornando-os de utilização limitada em sistemas onde a decisão tem de ser tomada num intervalo restrito de tempo.

Esta tese investiga estratégias de gestão de recursos em sistemas de computação heterogéneos, são apresentados várias heurísticas para mapeamento e agendamento de aplicações representadas em *workflow* baseadas em parâmetros de qualidade de serviço. Genericamente, as estratégias de mapeamento e agendamento podem ser classificadas em duas categorias principais: execução individual e concorrente. Nesta tese ambas as classes são consideradas e são propostas novas estratégias para cada uma dessas classes. A principal característica das estratégias propostas é a baixa complexidade computacional, tornando viável a sua utilização na gestão de sistemas heterogéneos de elevado desempenho. Um outro factor chave considerado nesta tese consiste na simulação baseada em dados reais de uma plataforma computacional. Foi utilizado o simulador SIMGRID que implementa um modelo de rede de comunicações correspondente ao modelo teórico *bounded multi-port*. Neste modelo, um processador pode comunicar com vários outros processadores em simultâneo, mas cada comunicação está limitada pela largura de banda e para qualquer comunicação que ocorra em troços partilhados, a largura de banda é considerada partilhada. Este esquema corresponde ao funcionamento das ligações TCP numa LAN. Para validação, as estratégias propostas são comparadas com outros algoritmos do estado-da-arte, com os mesmos objetivos e alguns de maior complexidade computacional de modo a demonstrar a relevância dos resultados obtidos. Em termos dos objetivos do problema de mapeamento e agendamento, são considerados dois parâmetros de QoS antagónicos, que são o tempo de execução e o custo da solução.

Resumidamente, as principais contribuições desta tese são: a) proposta de vários algoritmos de baixo custo computacional, baseados em parâmetros de QoS, para o mapeamento e agendamento de *workflows* em sistemas heterogéneos; b) utilização de um modelo real de sistema heterogéneo na simulação; e c) apresentação de resultados para aplicações geradas aleatoriamente bem como para aplicações do mundo real.

Abstract

Heterogeneous computing systems have given the scientific community access to greater resources for the execution and data analysis of scientific applications. To maximize performance in the execution of these applications, often described as workflows, task scheduling has become an essential, but highly demanding, tool. Efficient execution of scientific applications can be achieved by optimal job assignment to the platform's resources. The scheduling problem becomes even more challenging when the user's Quality of Service (QoS) requirements are considered objectives of the scheduling problem. The scheduling problem is well known as NP-complete. Therefore, most researchers in this field try to obtain a good solution by using meta-heuristic or search-based approaches that allow the user to control the quality of the produced solutions. However, these approaches usually impose significantly higher planning costs in terms of the time consumed to produce good results, making them less useful in real platforms that need to obtain map decisions quickly.

This thesis investigates strategies of resource management on heterogeneous computing systems and presents several heuristic approaches for the task scheduling of scientific workflow applications based on several Quality of Service (QoS) parameters. Generally, scheduling strategies can be classified into two main categories: single and concurrent workflow scheduling. In this thesis, both classes are considered, and new strategies are proposed for each class. The main advantage of the proposed strategies that they feature low time complexities, making them more useful in real platforms that need to obtain map decisions on the fly. Another key factor considered in this thesis is simulation based on real platform parameters. We use the SIMGRID toolkit as the basis for our simulation. SIMGRID provides a network model that corresponds to the 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. For validation purposes, each proposed strategy is compared with other state-of-the-art algorithms, having the same objectives, and some of them having higher time complexity, to highlight the relevance of the presented results. In terms of objectives of the scheduling problem, we consider two relevant and conflicting QoS parameters, namely, time and cost.

Briefly, the main achievements of this thesis are the proposal of low-time complexity workflow QoS-based scheduling algorithms on heterogeneous computing systems; the usage of a realistic model of the computing platform with shared links, as occurs in a common heterogeneous computing infrastructure; and the presentation of results for randomly generated graphs and for real-world applications.

Acknowledgements

I would like to express my sincere appreciation to my supervisor, Professor Jorge G. Barbosa. During of my PhD candidature, his full support and timely advice and comments kept me going on the right track. He always been available to help and encouragement. I am grateful to him for motivating and inspiring me to go deeply into the field of task scheduling and resource management of heterogeneous computing system. I consider myself honored for being his student.

I am grateful to Professor Frédéric Suter, Junior Researcher (CR1) at the IN2P3 Computing Center, who helped me to getting startted work with SIMGRID simulator. I am also grateful to Professor Radu Prodan, Associate Professor at Institute of Computer Science, University of Innsbruck, to giving chance work with his team to improve my research skills.

Love and encouragement from my wife, Mojgan, accompany me all the time. This thesis is dedicated to her.

Hamid Arabnejad

Contents

Chapter 1

Thesis Overview

1.1 Introduction

Heterogeneous computing can be defined as a range of different system resources that can be locally or geographically distributed and that are utilized to execute computationally intensive applications. In recent years, heterogeneous computing environments have been utilized by scientific communities to execute their scientific workflow applications. The growth of scientific workflows has also spurred significant research in the areas of generating, planning and executing such workflows in heterogeneous computing environments. Recently, utility computing has been rapidly moving towards a pay-as-you-go model, in which computational resources or services have different prices with different performance and Quality of Service (QoS) levels. In this computing model, users consume services and resources when they need them and pay only for what they use. Cost and time have become the two most important user concerns. Thus, the cost/time trade-off problem for scheduling workflow applications has become challenging. Scheduling consists of defining an assignment and mapping the workflow tasks onto available resources.

Scientific workflows are often large, consisting of thousands of individual tasks. Many such applications contain a set of jobs and files as input or output requirement for job execution. To achieve high performance for heterogeneous computing systems, it is essential to use a proper scheduling algorithm for the allocation and assignment of a workflow application task to available processors. Many complex applications in e-science and e-business can be modeled as workflows [DBG+03]. A popular representation of a workflow application is the Directed Acyclic Graph (DAG), in which nodes represent individual application tasks, and the directed edges represent inter-task data dependencies. A fundamental issue is how the workflow application should be executed on available resources in the platform in order to satisfy its objective requirements.

Task Scheduling is defined as a strategy to decide which (task selection) and where (processor selection) each application task should be executed, and it determines how the input/output data files are exchanged among them.

1.1.1 Motivation

The efficiency of executing parallel applications on heterogeneous computing systems critically depends on the methods used to schedule the tasks of a parallel application. Therefore, the task scheduling problem has been extensively explored by researchers in the past few decades.

The task scheduling problem is broadly classified into two major categories: Static Scheduling and Dynamic Scheduling. In Static scheduling, all information about tasks—such as execution and communication costs for each task and the relationship with other tasks—is known beforehand. In dynamic scheduling, such information is unavailable, and decisions are made at runtime. Moreover, Static scheduling is an example of compile-time scheduling, whereas Dynamic scheduling is representative of run-time scheduling. Static scheduling algorithms are universally classified into two major groups, namely Heuristic-based and Guided Random Search-based algorithms. Heuristic-based algorithms allow approximate solutions, often good solutions, with polynomial time complexity. Guided Random Search-based algorithms also give approximate solutions, but the solution quality can be improved by including more iterations, which therefore makes them more expensive than the Heuristic-based approach. The Heuristic-based group is composed of three subcategories: clustering, duplication and list-based scheduling [THW02]. In the case where the scheduling objective is the minimization of the *makespan*, i.e., the total execution time of the workflow application, clustering approaches try to reduce the communication time between tasks by assigning them to the same processor. To achieve this goal, tasks are grouped in different clusters, and all tasks in the same cluster must be executed in the same processor. Generally, clustering scheduling algorithms are mainly proposed for homogeneous systems to form clusters of tasks that are then assigned to processors. For heterogeneous systems, CHP algorithms [BR+04] and Triplet [CJ01] have been proposed, but they have limitations in higher-heterogeneity systems. In duplication, approaches are suggested to reduce the total execution time by assigning a task to more than one processor in order to reduce its communication time with its parents and, consequently, minimize the *makespan* of the application. The duplication approaches produce shorter *makespans*, but they have two disadvantages: a higher time complexity, i.e., cubic, in relation to the number of tasks, and the duplication of the execution of tasks, which results in more processor power used. This is an important characteristic not only because of the associated energy cost but also because, in a shared resource, fewer processors are available to run other concurrent applications. The list-based scheduling algorithms, on the other hand, produce the most-efficient schedules, without compromising the *makespan* and with a complexity that is generally quadratic in relation to the number of tasks.

Considering the number of involved workflow applications in the scheduling problem, the scheduling approach can be divided into two main classes: Single and Multiple. *Makespan* minimization is the most popular objective of scheduling problems. However, additional objectives can be considered when scheduling workflows onto a heterogeneous computing system, based on the user's QoS requirements. If we consider multiple QoS parameters, then the problem becomes more challenging. Many algorithms have been proposed for multi-objective scheduling, but in

most of them, meta-heuristic methods or search-based strategies have been used to achieve good solutions. However, these methods based on meta-heuristics or search-based strategies usually impose significantly higher planning costs in terms of the time consumed to produce good results, which makes them less useful in real platforms that need to obtain map decisions on the fly.

The quality of the scheduling approach is calculated by two main metrics: (a) producing good results and (b) having low time complexity when employed by the scheduler in a realistic scenario. Thus, it is a challenge to develop an efficient scheduling approach to produce good results with low time complexity. These metrics have motivated us to develop QoS-based scheduling approaches to produce good solutions with low time complexity.

1.1.2 Contributions

The efficient utilization of heterogeneous computing systems is mainly dependent on how applications are managed to execute on resources. This management largely entails scheduling (task-to-processor) decisions that are made based on the incorporation of the execution platform resources and application characteristics. The objective functions of a scheduling algorithm are defined by the user, and the goal of the scheduling approach is to satisfy user-defined QoS parameters. Our extensive investigation into scheduling problems produced several innovative approaches for heterogeneous computing systems and are listed briefly below:

- In addition to the common QoS objective of minimizing the total execution time, time and cost are considered here has the two main and conflicting objectives.
- For both single and multiple workflow application scheduling, new algorithms are proposed as follows.
  - For a single workflow scheduling subjected to minimizing the total execution time, a new list-based and heuristic scheduling algorithm called Predict Earliest Finish Time (PEFT) is proposed (chapter 3).
  - For multiple workflow scheduling subjected to optimizing the turnaround time of each application, a new heuristic scheduling algorithm called Fairness Dynamic Workflow Scheduling (FDWS) for scheduling dynamical workflow applications is proposed. The FDWS algorithm aims to reduce the individual turnaround time for each application, as its objective to reflect the QoS experienced by the users (chapters 4 and 5).
  - For a single workflow application subjected to minimizing execution time while constrained to a user-defined budget, a new heuristic algorithm named Heterogeneous Budget Constrained Scheduling (HBCS) is proposed (chapter 6).
  - For multiple workflow scheduling subjected to meeting budget constraints defined by users and for each application optimizing the turnaround time, two generic strategies for both task and processor selection phases for on-line scheduling of concurrent workflow applications are proposed (chapter 7).

  - For a single workflow application with time and cost constraint QoS parameters as the scheduling objectives, a new and heuristic approach named Deadline–Budget Constrained Scheduling (DBCS) is proposed. The objective of the proposed DBCS algorithm is to find a feasible schedule map that satisfies the user-defined deadline and budget constraint values (chapter 8).
  - A Multi-Workflow Deadline-Budget scheduling algorithm (MW-DBS) is proposed to schedule multiple and concurrent workflow applications that may be submitted at different moments in time and with the individual user's budget and deadline constraints. The MW-DBS is able to increase the number of successful applications that met their time and cost constraint values; however, from the service provider's viewpoint, the major issue is how much revenue is made. Thus, as an additional objective, the MW-DBS is also designed to obtain higher revenue for the provider through a higher rate of completed applications (chapter 9).

- The main issue considered in proposed strategies is time complexity, which determines the ability to use these strategies in real systems. Most scheduling algorithms use search-based or meta-heuristic strategies to achieve good solutions. However, these methods based on meta-heuristics or search-based strategies usually impose significantly high planning costs in terms of the time consumed to produce good results, which makes them less useful in real platforms that need to obtain map decisions on the fly. In this thesis, all proposed strategies are heuristic approaches that have low time complexity and yet obtain comparable results to search-based or meta-heuristic approaches.

- To achieve greater accuracy in the results comparison, we used the SIMGRID toolkit[1] [CLQ08] to afford a realistic simulation considering a bounded multi-port model in which bandwidth is shared by concurrent communications.

1.1.3 Structure of the thesis

This thesis is organized in two main parts. In the first part, chapter 1 presents an overview of the workflow scheduling problem. First, the application model, system model and QoS workflow scheduling problem are described briefly. Then, the experimental environment used in this thesis is explained. Next, a brief survey of the scheduling algorithm is presented. Finally, the main contribution reached with this thesis is presented, and the last section of this chapter presents future work.

In the second part, the set of articles produced under the scope of this thesis are presented between chapters 2 and 9. This part contains eight articles that describe the work conducted in detail, including the methodologies used, the results obtained and their discussion.

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

1.2 Problem Statement

In this section, we present a brief description of the application model, the system model and the QoS workflow scheduling problem.

1.2.1 Application Model

A typical workflow application can be represented by a Directed Acyclic Graph (DAG), a directed graph with no cycles. A DAG can be modeled by a three-tuple $G =< T, E, Data >$. Let n be the number of tasks in the workflow. The set of nodes $T = \{t_1, t_2, \cdots, t_n\}$ corresponds to the tasks of the workflow. The set of edges E represent their data dependencies. A dependency ensures that a child node cannot be executed before all its parent tasks finish successfully and transfer the required child input data. $Data$ is a $n \times n$ matrix of communication data, where $data_{i,j}$ is the amount of data that must be transferred from task t_i to task t_j. The average communication time between the tasks t_i and t_j is defined as:

$$\overline{C}_{(t_i \to t_j)} = \overline{L} + \frac{data_{i,j}}{\overline{B}} \tag{1}$$

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 [THW02].

Due to heterogeneity, each task may have a different execution time on each processor. Then, $ET(t_i, p_j)$ represents the Execution Time to complete task t_i on processor p_j in available processors set P. The average execution time of task t_i is defined as:

$$\overline{ET}(t_i) = \frac{\sum_{p_j \in P} ET(t_i, p_j)}{|P|} \tag{2}$$

where $|P|$ denotes the number of resources in processors set P.

In a given DAG, a task with no predecessors is called an *entry task* and a task with no successors is called an *exit task*. We assume that the DAG has exactly one entry task t_{entry} and one exit task t_{exit}. If a DAG has multiple entry or exit tasks, a dummy entry or exit task with zero weight and zero communication edges is added to the graph.

In addition to these definitions, next we present some of the common attributes used in task scheduling, which will be used in the following sections.

- $pred(t_i)$ and $succ(t_i)$ denote the set of immediate predecessors and immediate successors of task t_i, respectively. $FT(t_i)$ is defined as the Finish Time of task t_i on the processor assigned by the scheduling algorithm.

- Schedule length or *makespan* denotes the finish time of the last task of the workflow and is defined as $makespan = FT(t_{exit})$.

- $EST(t_i, p_j)$ and $EFT(t_i, p_j)$: denotes Earliest Start Time (EST) and the Earliest Finish Time (EFT) of a task t_i on processor p_j, respectively, and are defined as:

$$EST(t_i, p_j) = \max\Big\{ T_{Available}(p_j), \max_{t_{parent} \in pred(t_i)} \{AFT(t_{parent}) + C_{(t_{parent} \to t_i)}\} \Big\} \tag{3}$$

$$EFT(t_i, p_j) = EST(t_i, p_j) + ET(t_i, p_j) \tag{4}$$

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 t_i arrives at the processor p_j. The communication time $C_{(t_{parent} \to t_i)}$ is zero if the predecessor node t_{parent} is assigned to processor p_j. For the entry task, $EST(t_{entry}, p_j) = 0$. Then, to calculate EFT, the execution time of task t_i on processor p_j (ET) is added to its Earliest Start Time.

The financial cost $Cost(t_i, p_j)$ of executing task t_i on specific processor p_j, during the time span of $ET(t_i, p_j)$ is the sum of three cost components:

$$Cost(t_i, p_j) = EC(t_i, p_j) + TC(t_i) + SC(t_i) \tag{5}$$

where $EC(t_i, p_j)$ denotes the cost of running task t_i on processor p_j and is defined as $EC(t_i, p_j) = ET(t_i, p_j) \times Price(p_j)$ where $Price(p_j)$ denotes the processor price per time unit. $TC(t_i)$ denotes the cost of transferring data required for task t_i. In addition, $SC(t_i)$ denotes the data storage cost of task t_i. These cost components are determined by the target platform infrastructure.

$TotalCost$ is the overall cost for executing an application and is defined as:

$$TotalCost = \sum_{t_i \in T} AC(t_i) \tag{6}$$

where $AC(t_i)$ is defined as Assigned Cost of task t_i. After assigning a selected processor p_{sel} to execute task t_i, the assigned cost value is equal to $AC(t_i) = Cost(t_i, p_{sel})$. In the case of intra-cluster data transfer, zero monetary costs for communications between tasks are considered, i.e $TC(t_i) = 0$. And also we considered zero cost for task storage usage, $SC(t_i) = 0$, as this factor is common to all algorithms and does not influence the comparison of results.

1.2.2 System Model

Basically, in the service-oriented architecture for the Heterogeneous Computing System, as shown in Figure 1, applications can be submitted to the system by any user. The typical aim of this structure is to schedule tasks of each user workflow application to available resources based on the user's QoS (quality of service) demands. Submitted applications are collected by the Application

Data Base (DB) with their specifications and QoS requirements. In this architecture, a *Grid Scheduler* (GS) receives applications from Application DB and generates a tasks-to-resource map for each application based on their certain QoS objective requirements. To make a proper decision in the resource selection strategy for each application's task, GS needs information about the status of the available resources. The *Resource Service Information* is responsible for observing and collecting information about the current situation of resources, such as resource capacities, memory size, network bandwidth, availability, functionality and, especially, the available time slots for processing tasks. The Globus Monitoring and Discovery System (MDS) [CFFK01] is an example of Resource Service Information. In addition to resource information, application information such as lists of ready-to-execute tasks and the user's QoS requirements for each application are also necessary for making a feasible schedule. The *Ready Task pool* module collects tasks that are ready to execute among accepted workflow applications in application DB. A task is ready when all required information is prepared, i.e., its parents are executed. Also, the *QoS Parameter* module contains the users' QoS requests for their workflow applications. These two modules are used to select the task and related application at each step of the scheduling process. The *Service Executor* module implements task assignment by submitting a task to the selected resource and by monitoring task execution on resources; it then receives a notification of success or failure. The Globus GRAM (Grid Resource Allocation and Management) [CFK+98] is a good example of a service executor module.

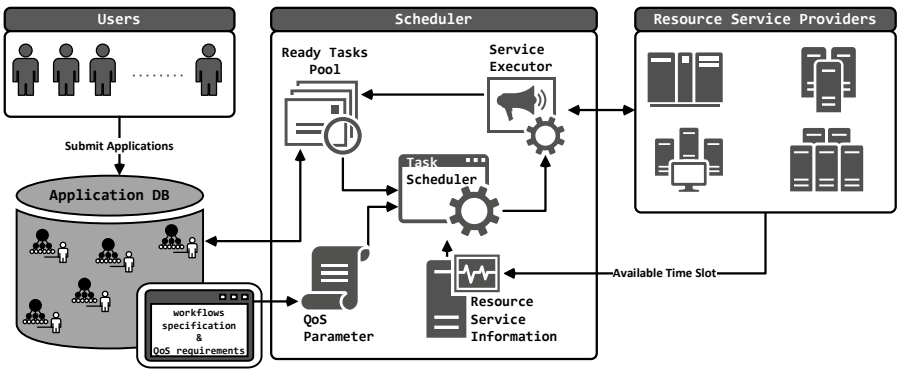


Figure 1: A General View of workflow Scheduler System

Finally, the main element of GS is the *Task scheduler*, which finds the suitable task-to-processor map to execute each ready task based on its QoS attributes and detailed information of each service.

1.2.3 QoS workflow scheduling

Workflow scheduling is the main issue in the management of workflow application execution on heterogeneous computing systems. Given the workflow application model from section 1.2.1 and the system model from section 1.2.2, workflow scheduling is defined as the process of ordering

tasks and mapping each task to a suitable processor to be executed, while satisfying the user's QoS objectives. An efficient scheduling algorithm can have a significant impact on the performance of the platform system. In general, the scheduling problem belongs to a class of problems known as NP-complete [CB76]. For this class of problems, finding the optimal solution is not attainable, even with a meta-heuristic approach with polynomial time complexity. The problem becomes even more challenging if we consider more than one QoS parameter to be optimized as an objective function in the scheduling problem, as instead of a single schedule map, there will be several solutions, such as a pareto-front with dominating solutions. Many scheduling algorithms using a variety of approaches have been proposed. Usually, the algorithms using search-based or meta-heuristic approaches try to find better or near-optimal solutions by increasing the number of iterations or the searching domain, which results in a higher time complexity, making them less useful in a real platform in which a quick answer is needed. On the other hand, heuristic strategies give us an acceptable solution with low time complexity. However, designing heuristic approaches to achieve acceptable solutions with low time complexity is a challenging problem.

To design a scheduling algorithm, the key factors are the QoS parameters. The meaning of QoS is defined by each user individually and could be different based on the application type and hardware capacity of the platform. The involvement of QoS as an objective in the scheduling strategy may change the processor selection phase in the algorithm. For instance, the common QoS objective of minimizing the total execution time makes selecting the processor with the earliest finish time for assigning a task a suitable strategy. But if the cost constraint is added to the problem, this strategy needs to be improved to consider the cost consumption in each step of the processor selection phase in order to meet the total cost constraint on the final solution. The final objective could be the optimization or constraint of QoS parameters in the scheduling problem. QoS requirements such as time limits (deadline) or cost constraints (budget) for application execution should be managed by workflow scheduling approaches. In [CSM+04], three QoS parameters, namely, time, cost and reliability, were presented. *Time* is defined as the total time required to complete the execution of a workflow application. *Cost* represents the charge of usage resources for processing and executing the workflow application. *Reliability* is related to the number of failures in the execution of workflow applications. In addition to these QoS parameters, which are referred to as General QoS parameters in [PSL03], the authors introduced other types of QoS parameters, namely, Internet-Service-Specific QoS parameters, such as availability, security and accessibility.

In this thesis, we consider mainly time and cost as two conflicting QoS objectives.

1.3 Experimental Environment

In this section, we describe in detail the environment used in our research. Generally, the simulation environment can be divided into two main parts: workflow structure and simulator.

1.3.1 Workflow Structure

To evaluate the relative performance of the scheduling algorithms, in addition to workflow structure, which defines the task dependency, we need a model of task execution time on the processors. This model should also be related to the type of available resources. Generally, it can be divided into two main categories: consistent and inconsistent.

In the consistent model of resources, if a processor has the lowest execution time for one task, then the same is true for any other task, but in the inconsistent model, the relationship among the task's computational requirements and machine capabilities are such that no structure such as that in the consistent case is enforced. In a real grid infrastructure, both models can be seen. For a complete evaluation on different data sets, in this thesis, we generated and used both types in our published papers, as described next.

Inconsistent Model : Ali et al. [ASMH00] presented the expected time to compute (ETC) estimation model, which has been widely used by researchers to evaluate scheduling algorithms. The ETC matrix provides an estimation for the execution time of each task in a heterogeneous computing system, taking into account two key properties: *Machine Heterogeneity* (MH) and *Task Heterogeneity* (TH). Machine heterogeneity evaluates the variation of execution times for a given task across the heterogeneous computing resources. Low machine heterogeneity indicates a system with similar computing resources, while high machine heterogeneity represents a system with different power capabilities. Task heterogeneity represents the variation in the task execution times for a given machine.

Consistent Model : In the consistent model, to evaluate the relative performance of the algorithms, the synthetic DAG generation program[2] is used for randomly generated workflows. The computational complexity of a task is modeled 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 selected randomly between 2^6 and 2^9. As a result, different tasks exhibit different communication/computation ratios. The DAG generator program defines the DAG shape based on four parameters: *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. A large value will induce 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.

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

1.3.2 Simulator

We resorted to simulation to evaluate the proposed algorithms in this thesis. Simulation allows us to perform a statistically significant number of experiments for a wide range of application configurations (in a reasonable amount of time).

We use the SIMGRID toolkit[3] [CGL+14] as the basis for our simulation. SIMGRID is a simulation framework for studying the behavior of large-scale distributed systems such as Grids, HPC and P2P systems. 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. SIMGRID is carefully designed to be scalable and extensible. It is possible to run a simulation composed of 2,000,000 processors on a computer with 16GB of memory [HLP15]. Relying on a well-established simulation toolkit allows us to leverage sound models of a heterogeneous computing system, such as that described in Fig. 1. In many research papers on scheduling, the authors assume a contention-free network model in which processors can simultaneously send data 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 [VL09].

1.3.3 Heterogeneous Computing Systems

To execute complex workflows, a high-performance cluster or grid platform is typically used. As defined in [BB99], 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. The nodes can exist in a single cabinet or be physically separated and connected via a local area network (LAN). Figure 2 illustrates a conceptual cluster architecture.

The target *heterogeneous computing 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 one that results from selecting processors from several clusters at the same site. Although a cluster is homogeneous, the set of processors selected form a heterogeneous machine. The processor latency can differ in a heterogeneous machine, but such differences are

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

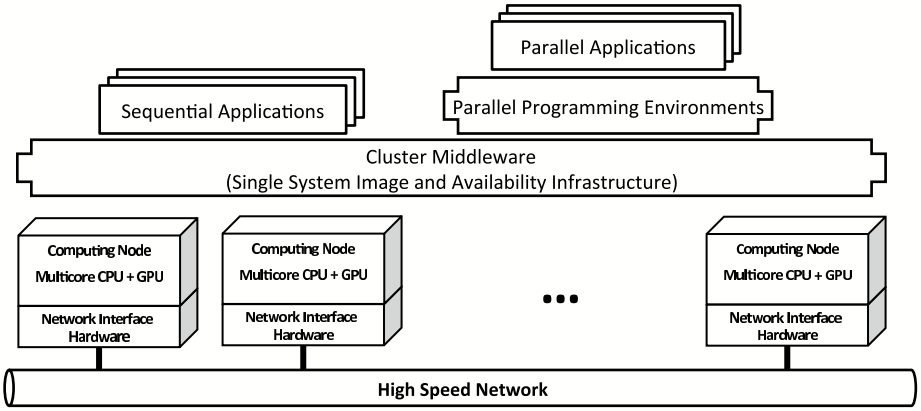


Figure 2: Conceptual cluster architecture.

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 that the bandwidth is the same throughout the entire network. Additionally, the execution of any task is considered nonpreemptive.

To make our simulations even more realistic, we consider platforms derived from clusters in the Grid5000 platform deployed in France[4] [CCD+05]. Grid5000 is an experimental testbed distributed across 10 sites and aggregating a total of approximately 8,000 individual cores. Table 1 gives the name of each cluster along with its number of processors, processing speed expressed in GFlop/s and heterogeneity. Each cluster uses an internal Gigabit-switched interconnect. The heterogeneity factor (hf) of a site is determined by the ratio between the speeds of the fastest and slowest processors.

This approach allows us to have heterogeneous configurations 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.

1.4 Scheduling Algorithm Taxonomy

Workflow scheduling has been extensively investigated. Considering the number of workflow applications in the scheduling problem, the scheduling strategies can be classified into two main classes: Single and Multiple workflow scheduling. In terms of time complexity, workflow scheduling strategies have been proposed under two main categories: heuristic and search-based or meta-heuristic approaches. The heuristic-based algorithms allow approximate solutions—often good solutions, but not necessarily the best ones—with low time complexity. On the other hand, search-based or meta-heuristic algorithms may achieve better solutions by performing more iterations, which results in higher running time than heuristic methods. In this section, we present a brief survey of task scheduling algorithms.

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

Site Name	Cluster Name	Number of CPUs	Power in GFlop/s	Site Heterogeneity
grenoble	adonis	12	23.681	$hf = 1.12$
	edel	72	23.492	
	genepi	34	21.175	
rennes	paradent	64	21.496	$hf = 2.34$
	paramount	33	12.910	
	parapluie	40	27.391	
	parapide	25	30.130	
lille	chicon	26	8.9618	$hf = 2.73$
	chimint	20	23.531	
	chinqchint	46	22.270	
	chirloute	8	24.473	
nancy	graphene	144	16.673	$hf = 1.24$
	griffon	20	20.678	
sophia	helios	56	7.732	$hf = 3.04$
	sol	50	8.939	
	suno	45	23.530	
lyon	capricorne	56	4.723	$hf = 1.20$
	sagittaire	79	5.669	
bordeaux	bordeplage	51	5.229	$hf = 2.55$
	bordereau	93	8.892	
	borderline	10	13.357	

Table 1: Description of the Grid5000 clusters

1.4.1 Single workflow scheduling

Single workflow scheduling algorithms are designed to schedule only a single workflow at a time. If all information about tasks such as execution and communication costs for each task and the relationship with other tasks are known beforehand, the scheduling method is categorized as a *Static scheduling* strategy; if such information is not available and decisions are made at runtime, it is categorized as a *Dynamic scheduling* strategy. Dynamic scheduling is adequate for situations where the system and task parameters are not known at compile time, which requires decisions to be made at runtime but with additional overhead. A sample environment is a system where users submit works, at any time, to a shared computing resource [MAS+99]. In this situation, a dynamic algorithm is required because the workload is only known at runtime, as is the status of each processor when new tasks arrive and, consequently, cannot optimize any QoS parameters based on the entire workflow. By contrast, a static approach can optimize or be constrained to QoS parameters, which are defined as schedule objectives or constraints, by considering all tasks independently of execution order or time because the schedule is generated before execution begins. In this thesis, in the case of single workflow scheduling, we consider that the information about the system and the workflow are known at compile time. Generally, the scheduling algorithms for a single workflow contain three main phases: the *prioritizing* phase, to give a priority to each task; the *task selection* phase, which selects a task for scheduling; and the *processor selection* phase, for selecting a suitable processor in order to meet the schedule objective functions. The last two

phases are repeated until all tasks are scheduled to suitable processors. The scheduling problem is further characterized as single- or multi-objective, as one or two QoS parameters are considered objectives.

1.4.1.1 Time-Optimization

In general, the primary goal of the scheduling algorithms for single workflow applications on heterogeneous computing systems is reducing the execution time, also called *makespan*. One of the most well-known heuristics, proposed in [THW02], named HEFT (Heterogeneous Earliest Finish Time), generates a schedule length comparable to those of other scheduling algorithms for a bounded number of heterogeneous processors, with a lower time complexity. HEFT uses an upward rank that represents the length of the longest path from each task to the exit node, including the computational cost of the task, to assign a rank value to each task in the workflow. Then, the task list is ordered by decreasing value of upward rank, and, in each step of the processor selection phase, the task on top of the task list is assigned to the processor that allows for the Earliest Finish Time. In [LSZ09], an adaptive dual-objective scheduling (ADOS) algorithm was proposed as a new semi-dynamic scheduling heuristic that takes into account both the makespan and resource usage. ADOS generates a random schedule as an initial solution, and by changing the initial assignment, tries to improve the makespan and/or resource usage. Additionally, during the workflow execution, if a task finishes later than expected, it uses a rescheduling strategy for the remaining jobs that are not yet running, so that the practicality of ADOS is increased. The AHEFT [YS07] algorithm is an HEFT-based adaptive rescheduling algorithm that consists of two major parts. First, AHEFT generates a schedule map based on the original HEFT scheduling algorithm. In the second part, AHEFT monitors the execution of the tasks. In this monitoring part, if significant events, such as joining and disjoining of resources to the grid resource pool, happened in the execution phase, the rescheduling phase tries to adapt to these new conditions in order to reduce the total makespan. The authors in [Kha12] proposed the CEFT (Constrained Earliest Finish Time) algorithm, which is based on the concept of constrained critical paths (CCPs). CEFT finds the CCPs in a given workflow, and by assigning and scheduling all tasks in each CCP on the same processor, it tries to reduce the communication costs between tasks in the workflow application. In [CJW+08, CJW+10], the authors proposed a scheduling algorithm called DAGMap, which consists of three phases: prioritizing, grouping and independent task scheduling. In the priority phase, DAGMap calculates the upward and downward rank for each task and, based on these rank values, determines the set of critical tasks. In the grouping phase, tasks are grouped by the upward priority and dependency relationship, while tasks in the same group are kept independent. Finally, in the task scheduling phase, it first calculates the Heterogeneity Factor (HF) to indicate the execution time deviation among independent tasks and then, based on the HF value for each task, DAGMap adopts the Max-Min or the Min-Min strategy to determine the target processor for the task. The authors in [HJ03] proposed the Heterogeneous Critical Parent Trees (HCPT) algorithm, which contains two main phases: listing tasks and processor assignment. In the first phase, the algorithm starts with an empty queue L and an auxiliary stack S that contains the critical path node pushed

in decreasing order of their Average Latest Start Time ($ALST$) value, i.e., the entry node is on top of S. Consequently, $top(S)$ is examined. If $top(S)$ has an unlisted parent (i.e., has a parent not in L), then this parent is pushed on the stack S. Otherwise, $top(S)$ is popped and enqueued into L. In the processor assignment phase, the algorithm tries to assign each task $t_i \in L$ to a processor p_j that allows the task to finish its execution as early as possible. The HPS algorithm proposed in [ITM05] has three phases, namely, *level sorting*, *task prioritization* and *processor selection*. In the level sorting phase, the given workflow is traversed in a top-down fashion to sort tasks at each level in order to group the tasks that are independent of each other. As a result, tasks in the same level can be executed in parallel. In the task prioritization phase, priority is computed and assigned to each task using the attributes *Down Link Cost* (DLC), *Up Link Cost* (ULC) and *Link Cost* (LC) of the task. The *DLC* of a task is the maximum communication cost among all the immediate predecessors of the task. The *DLC* for all tasks at level 0 is 0. The ULC of a task is the maximum communication cost among all the immediate successors of the task. The ULC for an exit task is 0. The *LC* of a task is the sum of *DLC*, ULC and maximum *LC* for all its immediate predecessor tasks. At each level, based on the *LC* values, the task with the highest *LC* value receives the highest priority, followed by the task with the next highest *LC* value and so on in the same level. In the processor selection phase, the processor that gives the minimum earliest finish time for a task is selected to execute that task. In [SZ04], the authors proposed a hybrid version of the HEFT scheduling algorithm. First, all tasks are sorted by their upward rank value in descending order. Then, tasks are divided into independent task groups. Then, by using the *Balanced Minimum Completion Time* (BMCT) strategy, the independent groups are scheduled. The BMCT strategy has two phases: in the first phase, it assigns all tasks to the processor that gives the earliest finish time; in the second phase, by changing the initial assignment, it tries to reduce the total execution time for a given workflow application. The Lookahead scheduling algorithm [BSM10] is based on the HEFT algorithm, and its main feature is its processor selection policy. To select a processor for the current task t, it iterates over all available processors and computes the earliest finish time for the child tasks on all processors. The processor selected for task t is the one that minimizes the maximum earliest finish time from all children of task t on all resources where task t is tried. This procedure can be repeated for each child of task t by increasing the number of levels analyzed. In addition to these heuristic approaches for workflow scheduling with makespan optimization as an objective function, there are other proposed approaches that are used in different strategies, such as duplication methods to decrease communication costs or using high time complexity approaches such as search-based or meta-heuristic to achieve the best or near-optimal solutions. However, their higher time complexity makes them less useful in a realistic platform, where a quick decision is needed.

The scheduling problem becomes more challenging when two or more QoS parameters are considered. In this case, the scheduling algorithm tries to find a suitable schedule map between the workflow's tasks and available resources in order to meet its objective function, which could be to optimize or to constrain the problem to a single or multiple QoS parameters. Time, cost, energy and reliability are commonly considered QoS parameters in recent research work in this

area. In this thesis and in the proposed algorithms, time and cost are two QoS parameters we considered as objective functions or constraints on the scheduling algorithms. Therefore, in the following paragraphs, the related work is classified based on these two QoS parameters.

1.4.1.2 Cost-Optimization, Deadline-Constraint

In [YLW07], Yuan et al. proposed a time-cost tradeoff dynamic heuristic scheduling strategy to optimize the cost and time of the whole workflow. During the scheduling, all ready tasks are partitioned into n ready lists that contain independent tasks and that can be executed in parallel. The algorithm identifies time-critical and cost-critical tasks in each ready list. The appropriate processor is selected as follows: the time-critical tasks can be completed by the processor with the lowest finish time, and the cost-critical tasks can be executed on the cheapest processor that has a lower finish time compared to the assigned deadline for its ready list. The same authors, in [YLWZ09], presented a heuristic scheduling algorithm called DET (Deadline Early Tree), which minimizes cost with a deadline constraint. The algorithm partitions all tasks into different paths based on the Early Tree. The whole deadline is divided into time windows of critical tasks, which can be applied to all feasible deadlines. For critical tasks, a dynamic programming method is used to obtain the optimal cost solution. For non-critical tasks, an iterative search finds suitable time windows while keeping the precedence constraints among tasks, and a local cost optimization is applied within these time windows. The communication time between tasks is not considered in their model, i.e, the tasks in their model have dependency with zero transfer time. Yao et al., in [YLM10], propose an integer programming (IP) approach to minimize the execution time of the workflow under a time constraint (deadline) parameter. They used the IP strategy to distribute the deadline into time windows for all tasks and applied local optimization to find the most suitable processor for each task that satisfies these local time window constraints. Yu et al. [YBT+05b] proposed a QoS-based workflow scheduling algorithm utilizing a Markov Decision Process approach for the service Grid. It minimizes the total cost of the application, while meeting the deadline constraints imposed by the user. Their algorithm first categorizes tasks into two classes: synchronization tasks (nodes that have more than one parent or child) and simple tasks. Then, the original workflow is partitioned into sub-workflows, and, based on the two classes of tasks, sub-deadlines are assigned to each partition. Finally, the cost-optimized mapping for each partition is obtained, guaranteeing the application deadline. Chen et al. [CZ09] proposed an ant colony optimization (ACO) to schedule large-scale workflows with various QoS parameters such as reliability, time, and cost in computational grids. In their proposed algorithm, time and cost could be defined as constraint or optimizing parameters. In [CZ12], a discrete version of the comprehensive learning PSO (CLPSO) algorithm based on the set-based PSO (S-PSO) method for the cloud workflow scheduling problem is proposed. For an IaaS cloud model, in [ANE13], two scheduling algorithms named IaaS Cloud Partial Critical Paths (IC-PCP) and the IaaS Cloud Partial Critical Paths with Deadline Distribution (ICPCPD2) were proposed for cost minimization constrained to a deadline, extending their previous PCP algorithm in [ANE12]. Due to on-demand resource provisioning, the time constraint can always be met as long as the cloud provides an unlimited number of computational resources.

This model corresponds to an unbounded set of resources, which differs from our context that considers a bounded set of processors. In [STZN13], the authors proposed the Multiterminal Cut for Privacy in Hybrid Clouds (MPHC) algorithm as an extended version of the IC-PCP algorithm with privacy constraints. In [CLCG13], a Bi-Direction Adjust Heuristic (BDA) scheduling algorithm is proposed to minimize the resource renting cost over unbounded dynamic resources in the Cloud platform for executing a given workflow within a deadline. BDA has two major steps: in the first step, it generates an initial schedule map by ignoring the hourly charging strategy; in the second step, a bi-direction adjust process, composed of forward and backward scheduling procedures, is applied to allocate each task to the appropriate VM instance. The authors in [CB14] proposed Enhanced IC-PCP with the Replication (EIPR) algorithm, which replicates tasks in idle time slots to reduce the schedule length. The advantage of using idle time slots for task duplication is that it increases the resource utilization rate without any extra cost. Mao et al. [MH11] proposed an auto-scaling mechanism that automatically scales computing instances based on workload information to minimize the cost of the scheduling map while meeting application deadlines on cloud environments. With the same objective function, in [BM11b], a Hybrid Cloud Optimized Cost (HCOC) algorithm combines the usage of private and public clouds. HCOC decides which resources should be leased from the public cloud to increase the processing power of the private cloud to execute a workflow within its deadline. Fundamentally, HCOC is an iterative algorithm with a high time complexity.

1.4.1.3 Time-Optimization, Budget-Constraint

Fard et al. [FFP13] proposed a cost-constraint time optimization scheduling algorithm in public commercial clouds based on a set of rescheduling operations. First, a new Cost Efficient Fast Makespan (CEFM) algorithm is proposed to optimize both the makespan and cost of a workflow execution. The CEFM algorithm starts by assigning each task to a VM instance, which executes the task with the lowest finishing time. Then, based on this initial mapping, CEFM tries to reduce the total cost of the workflow execution without increasing the makespan. By using an output schedule map obtained by the CEFM approach, a Budget-Constrained Scheduling in Clouds (BCSC) algorithm is proposed to schedule a given workflow application with a nearly optimal makespan while meeting a specified budget constraint. The main idea behind BCSC is to use a tradeoff between a decremental cost obtained by rescheduling tasks to cheaper VM instances and the incremental workflow makespan. Zeng et al. [ZVL12] proposed ScaleStar, a budget-conscious scheduling algorithm to minimize the execution time of large-scale many-task workflows in Clouds with monetary costs. They proposed a metric named Comparative Advantage (CA), which effectively balances the execution time and monetary cost goals, and the resources are selected based on the CA metric. Wu et at. [WLY+15] proposed the Critical-Greedy algorithm scheduling to minimize the workflow makespan under a user-specified financial constraint for a single datacenter cloud. In the first step, the proposed Critical-Greedy algorithm generates an initial schedule where the cloud meets a given budget for the workflow application. Then, in the next step, by iterative searching, it tries to reschedule critical tasks in order to reduce the

total execution time. Sakellariou et al. [SZTD07] developed two scheduling approaches, LOSS and GAIN, to construct a schedule optimizing time and constraining cost. Both algorithms use initial assignments made by other heuristic algorithms to meet the time optimization objective. A reassignment strategy is then implemented to reduce the cost and meet the second objective, the user's budget. In the reassignment step, LOSS attempts to reduce the cost, and GAIN attempts to achieve a lower makespan while attending to the user's budget limitations. In the initial assignment, LOSS has lower makespans with higher costs, and GAIN has higher makespans with lower costs. The authors proposed three versions of LOSS and GAIN that differ in the calculation of the tasks weights. The LOSS algorithms obtained better performance than the GAIN algorithms, and among the three different types of LOSS strategy, we used LOSS1 to compare to our proposed algorithm. All of the versions of the LOSS and GAIN algorithms use a search-based strategy for reassignments; to obtain their goals, the number of iterations needed tends to be high for lower budgets in LOSS strategies and for higher budgets in GAIN strategies. Zheng et al., in [ZS12, ZS13], proposed the Budget-constrained Heterogeneous Earliest Finish Time (BHEFT) algorithm, which optimizes the execution time of a workflow application constrained to a budget. BHEFT uses upward rank to assign priority to each task in a given workflow and then selects tasks in the order of their priority value. In the service selection phase, BHEFT filters all available processors based on the Spare Application Budget (*SAB*). *SAB* is defined as the difference between the remaining unused budget and the total average cost for the unscheduled task. Based on *SAB*, BHEFT defines a maximum threshold value, namely, the Current Task Budget (CTB), and filters all processors for the current task. The filtered set is called *affordable services* for the current task. If the *affordable services* is not empty, then the current task will be assigned to the processor with the lowest earliest finishing time. Otherwise, based on the *SAB* parameter, the cheapest service or the one with the lowest finishing time will be selected. The aim of BHEFT is to cover the user's budget by the cheapest assignment option in the service selection phase. It also tries to minimize the makespan by selecting the lowest finishing time in the case of budget availability, i.e., $SAB \geq 0$. A budget-constrained scheduling heuristic called greedy time-cost distribution (GreedyTimeCD) was proposed in [YRB09]. The algorithm distributes the overall user-defined budget to the tasks based on the tasks' estimated average execution costs. The actual costs of allocated tasks and their planned costs are also computed successively at runtime. This is a different approach, which optimizes task scheduling individually. First, a maximum allowed budget is specified for each task. Then, a processor is selected that minimizes time within the task budget.

In [ZVL15], the Security-Aware and Budget-Aware (SABA) workflow scheduling strategy is proposed to schedule a workflow under a budget constraint within the user's security requirements. By taking into account the data security requirement, they define two dataset concepts, namely, *movable data* and *immovable data*, to impose security restrictions on data. Every task with dependencies on the *immoveable data* should be executed on the same data center. For the processor selection phase, a Comparative Factor (CF), defined as the time-to-cost ratio, is proposed and the VM instance with the maximum CF value is selected.

1.4.1.4 Time-Optimization, Cost-Optimization

In [DPAM02], the Nondominated Sorting Genetic Algorithm II (NSGA-II) was proposed. NSGA-II evaluates each generated solution and gives a rank value to each of them. The algorithm not only finds a wide range of solutions over the true Pareto-optimal region but also finds a set point with a good spread of solutions in a reasonable computational time. Yu et al., in [YKB07], proposed a bi-criteria workflow execution planning approach that makes use of a multi-objective evolutionary algorithm (MOEAs). The authors analyzed several state-of-the-art multi-objective genetic approaches. The results show that, like most meta-heuristic algorithms, these approaches could achieve near-optimal solutions, but with a higher time complexity. Their proposed algorithm focuses on two conflicting QoS parameters, namely, time and cost, while meeting the deadline and budget constraint values for a given application. The algorithm generates a set of alternative solutions from which users can choose their most appropriate schedule map according to their consumed cost and total execution time.

In [TKB09], the authors proposed a workflow execution planning approach using Multi-objective Differential Evolution (MODE) to obtain a diverse distribution of solutions in the solution space. The algorithm generates a set of trade-off solutions according to two QoS metrics, namely, time and cost. The output set, composed of widespread alternative solutions, gives more flexibility to users to estimate their preferences and choose a desired workflow schedule based on their QoS requirements.

In [SKD07], a bi-objective genetic algorithm formulation was proposed that tries to achieve a trade-off between resource costs and application performance. A Multi-Objective Genetic Algorithm (MOGA) is used to find the best schedules that correspond to the pareto-optimal set. In [DFP12], the Multi-Objective Heterogeneous Earliest Finish Time (MOHEFT) algorithm was proposed. MOHEFT is based on the well-known HEFT algorithm for optimizing workflow scheduling problems. MOHEFT is a heuristic-based approach that generates several workflow schedule maps as a set of trade-off solutions in each processor selection step. Based on a newly proposed crowding distance measure, solutions with higher crowding distance are selected and passed on to the next processor selection step for the next ready task. The same authors, in [DNP14], proposed a modified version of MOHEFT in order to optimize execution time and energy consumption.

Unlike these methods, other algorithms return a single solution as output, trying to find an optimal schedule map as the solution. A scheduling strategy based on GA and PSO algorithms was proposed in [TULA13] to optimize workflow execution time and cost. A market-oriented hierarchical scheduling strategy for multi-objective in cloud workflow systems was proposed in [WLN+13]. They analyzed meta-heuristic-based workflow scheduling algorithms, such as GA, ACO and PSO, in cloud environments aiming to satisfy the QoS requirements.

However, due to the time-consuming nature of these search-based and meta-heuristic approaches, these algorithms are not the most suitable for online mapping.

1.4.1.5 Time-Constraint, Cost-Constraint

Poola et. al [PGB+14] proposed a robust scheduling algorithm based on partial critical paths with deadline and budget constraints in clouds. The partial critical path (PCP) of a task is created by finding the unassigned critical parent of the node and repeating the same procedure for the critical parent recursively until there are no further unassigned parents. Then, the algorithm tries to find the best suitable VM type based on the allocation policy. They proposed three allocation policies for PCP tasks: (a) Robustness-Cost-Time (RCT) to maximize robustness and minimize cost and makespan; (b) Robustness-Time-Cost (RTC), similar to RCT, giving higher priority to robustness but followed by time and, finally, cost; and (c) Weighted policy, which allows users to define their own objective function using the three parameters (robustness, time and cost) by assigning weights to each of them. In each of these policies, solutions are first sorted by the first parameter, followed by the second and third parameters. The best solution from this sorted list is selected for all tasks in PCP, which are assigned to the corresponding VM type. In [GS13], a particle swarm optimization scheduling algorithm was proposed to minimize workflow time and cost simultaneously under the user's deadline and budget constraints. Prodan et al. [PW10] proposed a general bi-criteria scheduling heuristic called the Dynamic Constraint Algorithm (DCA), which is based on dynamic programming to optimize two independent generic criteria for workflows, e.g., execution time and cost. The DCA scheduling algorithm has two main phases: the first selects one criterion as the primary one and optimizes it; in the second phase, it optimizes the secondary criteria while keeping the primary criteria within a defined sliding constraint. In particular, DCA performs a full domain search in the second phase of the algorithm. Garg et al. [GS11] proposed a multi-objective non-dominated sort particle swarm optimization (NSPSO) approach to find schedule maps that minimize the makespan and total cost under the specified deadline and budget constraints. In [YB06b], Yu et al. proposed a genetic algorithm (GA) approach for scheduling workflow applications constrained to budget and deadline on heterogeneous environments. Two fitness functions are used to encourage the formation of individuals that satisfy the deadline and budget constraints.

1.4.2 Concurrent Workflow Scheduling

In contrast to single workflow scheduling, concurrent, or multiple, workflow scheduling has not received much attention. As single workflow strategies, multiple workflow scheduling algorithms can be divided into two main categories: static and dynamic strategies. In static strategies, workflows are available before the execution starts, that is, 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, for example, when a user has a set of nodes to run a set of workflows. This methodology is applied by most common resource management tools, where a user requests a set of nodes to execute his/her jobs exclusively. On the other hand, dynamic strategies exhibit online 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 a single workflow scheduling. However, in both cases, single or multiple QoS parameters can be defined as the scheduling objectives.

1.4.2.1 Static Concurrent workflow scheduling

Several algorithms have been proposed for static 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 [ZS06]. 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 the time when scheduled alone). They proposed two algorithms: one fairness policy based on finishing time and another fairness policy based on current time. Both algorithms first schedule each DAG on all processors with static scheduling (like HEFT [THW02] or Hybrid.BMCT [SZ04]) 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 the current time. In [BM10b], a path clustering heuristic was proposed that combines the clustering scheduling technique to generate groups (clusters) of tasks with the list scheduling technique to select tasks and processors. Based on this methodology, the authors propose and compare four algorithms: (i) sequential scheduling, where workflows are scheduled one after another; (ii) a gap search algorithm, which is similar to the former but searches for spaces between already-scheduled tasks; (iii) an interleave algorithm, where pieces of each workflow are scheduled in turns; and (iv) group workflows, where the workflows are joined to form a single workflow and are then scheduled. The evaluation was made in terms of schedule length and fairness, and it was concluded that interleaving the workflows leads to a 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 [CDS10], the algorithms for offline 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 [HCTY+12], two workflow scheduling algorithms were presented, namely, multiple workflow grid scheduling

MWGS4 and MWGS2, with four and two stages, respectively. The four-stage version comprises labeling, adaptive allocation, prioritization, and parallel machine scheduling. The two-stage version applies only adaptive allocation and parallel machine scheduling. Both algorithms, MWGS4 and MWGS2, are classified as offline 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 the mean critical path waiting time and critical path slowdown. Recently, Malawski et. al [MJDN15] proposed three scheduling algorithms, two dynamic and one static, for scientific workflow ensembles on clouds in order to complete workflows from an ensemble under a total budget and deadline constraints.

1.4.2.2 Online Concurrent workflow scheduling

Some algorithms have been proposed for online workflow scheduling. A planner-guided strategy, the RANK_HYBD algorithm, was proposed by Yu and Shi [YS08] 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 in each workflow application using rank upward priority measure. In each step, the algorithm reads all ready tasks from all workflows and selects the next task to schedule based on their rank. If the ready tasks belong to different workflows, the algorithm selects the task with lowest rank; if they belong to the same workflow, the task with the highest rank is selected. With this strategy, the RANK_HYBD algorithm allows the workflow with the lowest rank (lower makespan) to be scheduled first to reduce the waiting time of the workflow in the system. Hsu and Wang, in [HHW11], proposed Online Workflow Management (OWM) for scheduling multiple online workflows. Unlike RANK_HYBD, which puts all ready tasks from each workflow into the ready list, the OWM algorithm selects only a single ready task from each workflow with the highest rank into the ready list. Then, until there are unfinished workflows in the system, the OWM algorithm selects the task with the highest priority from the ready list. The earliest finish time (EFT) is then calculated for the selected task on each processor, and the processor with minimum earliest finish time is selected. If the processor is free at that time, the OWM algorithm assigns the selected task to the selected processor; otherwise, it keeps the selected task in the ready list to be scheduled later. In [LCJY09], 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 automatically adapting to changes in network transmission speed. 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. In [XCWB09], an algorithm was proposed for scheduling multiple workflows with multiple QoS constraints on the cloud. The resulting multiple QoS-constrained scheduling strategies of multiple workflows (MQMW) minimize the makespan and the cost of the resources and increase the scheduling success rate. The algorithm considers

two objectives, time and cost, that can be adapted to the user requirements. In [BM11a], 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 online scheduling but with the goal of minimizing a collective metric. This approach is different from the independent workflow execution we consider in this thesis. Zhou et al.[ZH14] proposed ToF, a general transformation-based optimization framework for optimizing the performance and cost of workflows in the cloud. Bochenina, in [Boc14], introduced a strategy for mapping the tasks of multiple workflows with different deadlines on the static set of resources. Jiang et al. in [JHC+11] proposed a method to minimize the total execution time of a scheduling solution for concurrent workflows in the HPC cloud. Their method tries to take advantage of any schedule gaps. First, a workflow is partitioned into several tasks, grouped by using a clustering-based PCH approach [BM10b, BM07]. Then, the proposed distributed gap search is applied to allocate these task groups to processors. The difference between the original gap search algorithm and proposed distributed gap search method is that by using the original gap search method, an entire task group is allocated to a single gap on a specific resource, but the proposed distributed gap approach allows for allocating the tasks of the same group to different gaps on different resources.

1.5 Main Contributions Achieved

In this section, I summarized the main contributions achieved in this thesis. This thesis focus on scheduling algorithms for workflow applications on heterogeneous computing systems based on user's QoS requirements. Two major QoS parameters considered in this thesis are cost and time. Therefore, the proposed algorithms are categorized attending to the number of simultaneous workflows, i.e. single and concurrent, and based on the QoS parameters, namely time and cost. In the following paragraphs, the proposed algorithms are briefly described and the full detail is shown on each related chapter.

Single workflow application, Time optimization

In our first work, chapter 2, we evaluated the performance of list-based scheduling algorithms. We compared their results with the solutions achieved by three meta-heuristic algorithms, namely, Tabu Search, Simulated Annealing, and Ant Colony System. The meta-heuristic algorithms, which feature a higher processing time, always achieved better solutions than the list scheduling heuristics with quadratic complexity. We then compared the best solutions for both types, step by step. We observed that the best meta-heuristic schedules could not be achieved if we followed the common strategy of selecting processors based only on current task execution time, because the best schedules consider not only the immediate gain in processing time but also the gain in a sequence of tasks. Most list-based scheduling heuristics with quadratic time complexity assign a task to a processor by evaluating only the current task. This methodology, although inexpensive, does not evaluate what is ahead of the current task, which leads to poor decisions in some cases. Algorithms that

analyze the impact on children nodes, such as Lookahead [BSM10] exist, but they increase the time complexity to the fourth order. The most powerful feature of the Lookahead algorithm, as the best algorithm with the lowest makespan, is its ability to forecast the impact of an assignment for all children of the current task. This feature permits better decisions to be made in selecting processors, but it increases the complexity significantly.

Therefore, in order to keep the ability to forecast while maintaining quadratic time complexity, In chapter 3, we propose the Predict Earliest Finish Time (PEFT) scheduling algorithm for heterogeneous computing systems. PEFT scheduling algorithm, unlike other state-of-the-art algorithms which are using the Earliest Finish Time (EFT) measure to select suitable processors for each task, introduced a look-ahead feature without increasing the time complexity associated with computation of an optimistic cost table (OCT). The OCT is a matrix in which the rows indicate the number of tasks and the columns indicate the number of processors, where each element $OCT(t_i, p_k)$ indicates the maximum of the shortest paths of t_i children's tasks to the exit node considering that processor p_k is selected for task t_i. The OCT value of task t_i on processor p_k is recursively defined by 7 by traversing the workflow from the exit task to the entry task:

$$OCT(t_i, p_k) = \max_{t_{child} \in succ(t_i)} \left[\min_{p_w \in P} \left\{ OCT(t_{child}, p_w) + ET(t_{child}, p_w) + \overline{C}_{(t_i \to t_{child})} \right\} \right] \tag{7}$$

where $\overline{C}_{(t_i \to t_{child})}$ is the average communication cost, which is zero if t_{child} is being evaluated for processor p_k, and $ET(t_{child}, p_w)$ is the execution time of task t_{child} on processor p_w. $OCT(t_i, p_k)$ represents the maximum optimistic processing time of the children of task t_i because it considers that children tasks are executed in the processor that minimizes processing time (communications and execution) independently of processor availability, as the OCT is computed before scheduling begins. Because it is defined recursively and the children already have the optimistic cost to the exit node, only the first level of children is considered. For the exit task, the $OCT(t_{eit}, p_k) = 0$ for all processors $p_k \in P$.

To select a processor for a task, we compute the Optimistic EFT (O_{EFT}) that sums to EFT the computation time of the longest path to the exit node. In this way, we are looking forward (forecasting) in the processor selection; perhaps we are not selecting the processor that achieves the earliest finishing time for the current task but we expect to achieve a shorter finishing time for the tasks in the next steps. The aim is to guarantee that the tasks ahead will finish earlier which is the purpose of the OCT table. O_{EFT} is defined by equation 8.

$$O_{EFT}(t_i, p_k) = EFT(t_i, p_k) + OCT(t_i, p_k) \tag{8}$$

The proposed PEFT algorithm is formalized in Algorithm 1.

Algorithm 1 The PEFT algorithm

1: Compute OCT table and $rank_{oct}$ for all tasks in a given workflow application
2: Create Empty list *ready-list* and put t_{entry} as initial task
3: **while** *ready-list* is NOT Empty **do**
4: $t_i \leftarrow$ the task with highest $rank_{oct}$ from *ready-list*
5: **for all** processor p_k in the processor-set P **do**
6: Compute $EFT(n_i, p_k)$ value using insertion-based scheduling policy
7: $O_{EFT}(t_i, p_k) = EFT(t_i, p_k) + OCT(t_i, p_k)$
8: **end for**
9: Assign task t_i to the processor p_k that minimize O_{EFT} of task n_i
10: Update *ready-list*
11: **end while**

In terms of time complexity the PEFT requires the computation of OCT table that is $O(p(e+v))$ and to assign the tasks to processors it is of the order $O(v^2.p)$. The total time is $O(p(e+v)+v^2.p)$. For dense DAGs e becomes v^2 being the total algorithm complexity of the order $O(v^2.p)$. That is, the time complexity of the PEFT is of the same order as the HEFT algorithm.

The results show that the PEFT scheduling algorithm outperforms the state-of-the-art list-based algorithms for heterogeneous systems in terms of schedule length ratio, efficiency, and frequency of best results while maintaining quadratic time complexity. Please note that, the PEFT scheduling algorithm is designed and proposed for the inconsistent model of execution time and, as it is shown in the paper, for the consistent model it has similar results to HEFT.

Multiple workflow applications, Time optimization

When we consider multiple and concurrent workflow application submitted at different moments in time, to achieve an efficient execution of a workflow, an effective scheduling strategy that decides when and which resource must execute the tasks of the workflow is necessary. In this case, the common definition of the 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 QoS experienced by the users, which is related to the response time of each user application. Usually, dynamic (on-line) strategies for scheduling multiple workflow applications needs to contain three main policies: filling the *ready tasks* pool, task selection policy to select a task among all available ready tasks to be executed and processor selection policy which finds the best suitable processor for selected tasks by considering QoS objectives of the task's workflow application. In section 1.4.2, we presented some proposed strategies in this area. Two well-known dynamic scheduling strategies for multiple workflows are RANK_HYBD [YS08] and OWM [HHW11] algorithms. As explained before, because of the strategy for filling ready tasks pool and task selection phases, RANK_HYBD 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. Unlike the RANK_HYBD algorithm, OWM selects only a single ready task from each workflow application so that it gives all workflow applications the chance to be selected in the current time for scheduling. Also, for selecting a task among all ready tasks, the OWM algorithm selects and schedules tasks from the longer workflows first. Then, like RANK_HYBD, all processors (both free and busy processors) are tested and the one that has the lowest FET is candidate for task assignment. If candidate processor was not free at that time, it keeps the selected task in the ready list in order to be scheduled later. Since the system is dynamic, it is possible that at any time a new application may arrive and the postponed task may have lower priority than the new ones and therefore it is postponed again. This may lead to an excessive completion time for smaller workflows. Both algorithms, RANK_HYBD and OWM, present results in terms of average makespan. This metric combines long and short workflows and does not allow to infer the average waiting time spent by the workflows individually.

In chapter 4, we propose the Fairness Dynamic Workflow Scheduling (FDWS) for scheduling dynamically workflow applications in a heterogeneous system. 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, for example, the turnaround time, including execution time and waiting time. To fill the ready tasks list, considering all ready tasks from each workflow leads to an unbiased preference for longer workflows and the consequent postponing of smaller ones resulting in higher turnaround time and unfair processor sharing. Therefore, only a single ready task with highest priority from each workflow is added to the ready tasks pool. For selecting a task among all ready tasks, both RANK_HYBD and OWM algorithms used upward rank to select a task from the pool of ready tasks. In FDWS algorithm, we proposed $rank_r$ metric to assign a second priority to each task in ready tasks pool.

$$rank_r(t_{i,j}) = \frac{1}{PRT(DAG_j)} \times \frac{1}{|CP(DAG_j)|} \tag{9}$$

where $t_{i,j}$ is the ith task belonging to workflow (DAG) j, PRT is defined as the Percentage of Remaining Task number for workflow j and CP is Critical Path length of workflow j. The $rank_r$ gives more priority to workflows that are almost completed and only have few tasks to execute (lower PRT value). However, the PRT does not consider the width of the workflow. A wider workflow has a shorter $|CP|$ than other workflows with the same number of tasks; it also has a lower expected finishing time. Therefore, in this case, FDWS would give higher priority to workflows with smaller $|CP|$ values. In both RANK_HYBD and OWM, only the individual upward rank 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. For the processor selection phase, only the free processors are considered in FDWS algorithm and the processor with the lowest finishing time for the current task is selected for task assignment.

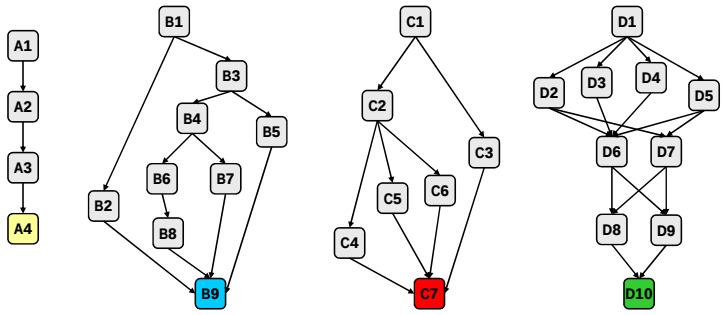
The proposed FDWS (Fairness Dynamic Workflow Scheduling) algorithm is formalized in Algorithm 2.

Algorithm 2 The FDWS algorithm

```
while Workflow Pool is NOT Empty do
    if new workflow has arrived then
        calculate rank_u for all tasks of the new Workflow
        Insert the Workflow into Workflow Pool
    end if
    Ready_Pool ← ready tasks (one task with highest rank_u from each DAG)
    calculate rank_r(t_{i,j}) for each task t_i belonging to DAG_j in Ready_Pool
    while Ready_Pool ≠ ϕ AND CPUs_free ≠ 0 do
        T_sel ← the task with highest rank_r from Ready_Pool
        P_sel ← the processor with lowest EFT for task T_sel among all available and free
        processors
        Assign Task T_sel to processor P_sel
        remove Task T_sel from Ready_Pool
    end while
end while
```

In order to have better demonstration of effectiveness of the proposed scheduling algorithm, in Figure 3, we compare proposed FDWS algorithm with RANK_HYBD and OWM algorithms for 4 workflow applications with entrance time to system equals to 0 (all DAGs submitted at the same time).

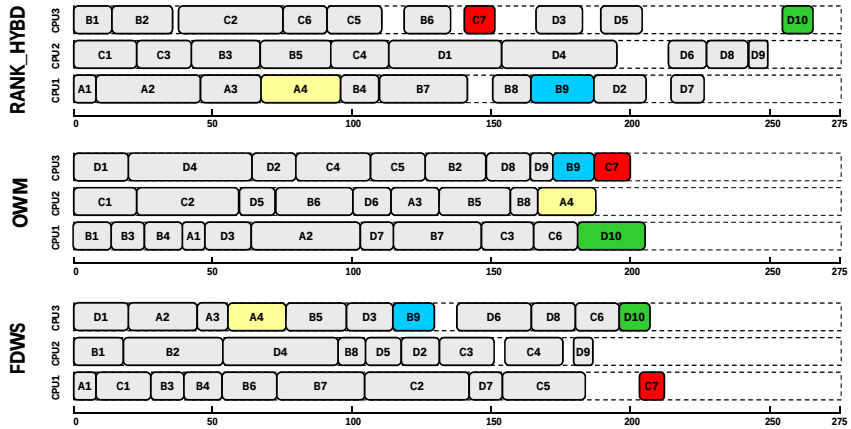
As Figure3(c) shows, RANK_HYBD algorithm tries to finish workflows with lower rank first and then schedules the higher ones. In RANK_HYBD algorithm, workflow A finished at 97, followed by workflows B, C and D with finishing time 189, 151 and 266, respectively. The OWM algorithm tries to finish workflows with higher ranks first. However, scheduling the highest rank workflows seems to be good idea in order to decrease the turnaround time for these type of workflows, but as a workflow has few tasks to be executed, the rank value becomes lower, and those tasks remaining will be postponed due to tasks from other workflows with higher ranks. This situation causes all workflows to finish more and less at the same time; in our sample, the finishing times are 189 , 189 , 200 and 205 for workflow A, B, C and D, respectively. In the FDWS algorithm, the history execution of each workflow application is taken into account by calculating the percentage of reminding tasks. With this strategy, if we have a workflow with higher rank but lower remaining number of tasks, the scheduler may selects this workflow to decrease its turnaround time instead of start to schedule new workflows with lower rank. In our sample, the finish time for workflows A, B, C and D with FDWS algorithm will be 77, 131, 212 and 209 respectively. As shown in



(a) Sample Workflows

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
DAG A	103	86	53	24	-	-	-	-	-	-
DAG B	160	51	142	108	49	74	53	37	18	-
DAG C	151	112	45	62	61	30	11	-	-	-
DAG D	172	93	101	112	98	71	66	42	30	18

(b) upward rank values for each sample workflows



(c) Task assignment with three scheduling algorithms

Figure 3: Scheduling of 4 sample sample DAGs

Figure3(c), the main advantage of FDWS is to consider the turnaround time for each individual workflow application instead of decreasing the overall execution time of all submitted workflows.

The results shown in chapter 4 are achieved for inconsistent model of execution time. In chapter 5, we proposed an extended version of FDWS and we present results based on a consistent model of execution time and using realistic parameters for simulation platform such as bounded multi-port model. In bounded multi-port 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[VL09].

Single workflow applications, Budget Constrained-Time optimization Utility computing is a service provisioning model that provides computing resources and infrastructure management to customers as they need them, as well as a payment model that charges for usage (pay-as-you-go model). Cost and time are two of the most important user concerns. Thus, the cost/time trade-off problem for scheduling workflow applications has become challenging. In chapter 6, we propose the Heterogeneous Budget Constrained Scheduling (HBCS) algorithm, which minimizes execution time while constrained to a user-defined budget. Like most list-based algorithms, HBCS consists of two phases, namely a task selection phase and a processor selection phase. For the task selection phase, HBCS uses the upward rank for assigning a priority to each task in a given workflow. First, all tasks are ordered by their upward rank value and, in each step and then among all ready tasks, the task with highest priority is selected as current task (t_{curr}). To find the best suitable processor (p_{sel}) for t_{curr} in the processor selection phase, in addition to the time metric which leads to a lower makespan for a given workflow, the cost metric should be considered in order to meet the cost constraint value defined as a QoS parameter of the scheduling problem. The following attributes are used in the processor selection phase of the HBCS scheduling algorithm:

- $Cheapest_{Cost}$: denotes the total execution cost of a given workflow on the cheapest processors.
- RCB: denotes the Remaining Cheapest Budget for unscheduled tasks in a given workflow and calculated as sum of minimum cost for each unscheduled tasks excluding the current task. The initial value is $RCB = Cheapest_{Cost}$ and each step, before executing the processor selection phase for the current task is updated by : $RCB = RCB - Cost_{lowest}(t_{curr})$ where $Cost_{lowest}$ is the lowest assignment cost for the current task (t_{curr}) among all available processors.
- RB: denotes the actual Remaining Budget and represents the budget available for the remaining unscheduled tasks. The initial value for the Remaining Budget is the user budget, $RB = BUDGET_{user}$. and it is updated after the processor selection phase for the current task (t_{curr}) by $RB = RB - Cost(t_{curr}, p_{sel})$ where p_{sel} is the selected processor for the current task assignment.

The current task has a different value of execution cost and time on each processor and to select a suitable processor, we need a trade-off between cost and time factors. Two relative quantities, namely Time rate ($Time_r$) and Cost rate ($Cost_r$) are defined for the current task on each tested processor $p_j \in P$ by equations 10 and 11, respectively.

$$Time_r(t_{curr}, p_j) = \frac{\max\limits_{\bar{p} \in P} \{EFT(t_{curr}, \bar{p})\} - EFT(t_{curr}, p_j)}{\max\limits_{\bar{p} \in P} \{EFT(t_{curr}, \bar{p})\} - \min\limits_{\bar{p} \in P} \{EFT(t_{curr}, \bar{p})\}} \tag{10}$$

$$Cost_r(t_{curr}, p_j) = \frac{Cost(t_{curr}, p_{best}) - Cost(t_{curr}, p_j)}{\max\limits_{\bar{p} \in P} \{Cost(t_{curr}, \bar{p})\} - \min\limits_{\bar{p} \in P} \{Cost(t_{curr}, \bar{p})\}} \tag{11}$$

where, p_{best} is the processor with lowest earliest finish time for the current task. $Time_r$ measures how much the finishing time of the current task on processor p_j is shorter than the highest finish time. Similarly, $Cost_r$ measures how much less the actual cost on p_j is than the cost on the processor that results in the earliest finish time. Both variables are normalized to their highest ranges.

Finally, to select the processor for the current task t_{curr}, the worthiness value for each processor $p_j \in P$ is computed as:

$$worthiness(t_{curr}, p_j) = \begin{cases} -\infty & \text{if } Cost(t_{curr}, p_j) > Cost(t_{curr}, p_{best}) \\ -\infty & \text{if } Cost(t_{curr}, p_j) > RB - RCB \\ Cost_r(t_{curr}, p_j) \times Cost_{Coeff} & \\ +Time_r(t_{curr}, p_j) & \text{otherwise} \end{cases} \tag{12}$$

where $Cost_{Coeff}$ is the quantity Cost Coefficient and defined as the ratio between remaining cheapest budget (RCB) and the actual Remaining Budget (RB).

$$Cost_{Coeff} = \frac{RCB}{RB} \tag{13}$$

The quantity Cost Coefficient provides a measurement of the least expensive assignment cost relative to the remaining budget available. If $Cost_{Coeff}$ is near one, it means that the available budget allows for selecting only the cheapest processors.

In the worthiness value equation, the first two statements guarantee that if the cost of the current task t_{curr} on processor p_j is higher than the cost on the processor that gives the minimum finishing time and if that cost is higher than the available budget for task t_{curr}, then processor p_j cannot be selected. With these statements, the resulting schedule does not exceed the user budget and is guaranteed to be valid. In the third statement, the worthiness value depends on the available budget and on the time during which a processor can finish the task. If the remaining budget (RB) is high, then $Time_r$ has more influence, and a processor with the greater difference in finishing time is compared to the worst processor, will have higher worthiness. In contrast, if the remaining budget is smaller, then the cost factor will increase the worthiness of the processors with lower cost to run task t_{curr}. After testing all of the processors, the one with highest worthiness value is selected (p_{sel}), and the remaining budget (RB) is updated.

The proposed HBCS (Heterogeneous Budget Constrained Scheduling) algorithm is formalized in Algorithm 3.

Algorithm 3 HBCS algorithm

Require: DAG and user defined BUDGET
1: Schedule DAG with HEFT and Cheapest algorithm
2: Set task priorities with $rank_u$
3: **if** $HEFT_{cost} < BUDGET$ **then**
4: **return** Schedule Map assignment by HEFT
5: **end if**
6: $RB = BUDGET$ and $RCB = Cheapest_{Cost}$
7: **while** there is an unscheduled task **do**
8: n_i = the next ready task with highest $rank_u$ value
9: Update the Remaining Cheapest Budget (RCB)
10: **for all** Processor $p_i \in P$ **do**
11: calculate $FT(n_i, p_j)$ and $Cost(n_i, p_j)$
12: **end for**
13: Compute $Cost_{Coeff}$ as defined in Eq.13
14: **for all** Processor $p_i \in P$ **do**
15: calculate $worthiness(n_i, p_i)$ as defined in Eq.12
16: **end for**
17: P_{sel} = Processor p_i with highest $worthiness$ value
18: Assign Task n_i to Processor P_{sel}
19: Update the Remaining Budget (RB)
20: **end while**
21: **return** Schedule Map

In terms of time complexity, HBCS algorithm has a time complexity of the order $O(v^2.p)$.

The HBCS algorithm was compared with three well-known scheduling algorithms, namely LOSS1 [SZTD07], GreedyTimeCD and BHEFT [ZS12, ZS13]. The LOSS scheduling algorithm is a search-based strategy which has two main phases, the first makes an initial assignment by using a heuristic algorithm, to meet the time optimization objective, and the second phase is a reassignment strategy to reduce costs and meet the second objective, the user's budget. Our proposed scheduling algorithm differs from LOSS algorithm because HBCS does not have any initial assignments and in contrast to these search-based strategies, the HBCS is not iterative. So, the time to produce a schedule is constant for a given workflow and not uncertain. Also, we have made some modifications in LOSS's original algorithm. The original implementation assumed that all of the processors had different costs, and therefore, there was no conflict in selecting a processor based on the cost parameter. In our heterogeneous computing environment, each cluster is homogeneous (section 1.3.3), so there could be more than one processor candidate. In this case, we tested all of the possible processors and select the one which achieves the smallest makespan. The same procedure was applied to the least expensive scheduling strategy, which attempts to schedule each task on the service with the lower execution cost. The BHEFT algorithm is a heuristic strategy designed to minimize the makespan and cover user-defined budget value. Our proposed HBCS algorithm differs from BHEFT in two important aspects: first, we allow more processors to be considered as affordable and, therefore, selected; and second, we

do not necessarily select the processor that guarantees the earliest finishing time, as BHEFT does. Instead, we compute a worthiness value, proposed in this paper, which combines the time and cost factors to decide on the processor for the current task. The GreedyTimeCD algorithm is also heuristic with the same objective, time optimization-cost constraint.

The HBCS algorithm was shown to achieve lower makespans for all of the budget factors; that is, HBCS can produce shorter makespans for the same budget rather than state-of-the-art algorithms. A reduction of up to 30% in execution time was achieved while maintaining the same budget level. The results were obtained in a simulation with a realistic model of the computing platform and with shared links, as occurs in a common grid infrastructure.

Multiple workflow applications, Budget Constrained-Time optimization This is an extension of the Multiple workflow applications, a time optimization problem where budget constraint is considered. Users submit jobs at any moment in time and therefore a dynamic behavior is required. When scheduling multiple independent workflows that represent user jobs and are thus submitted at different moments in time, a dynamic behaviour is required to redistribute the workload. Most concurrent workflow scheduling algorithms proposed are for the static case. However, there are some methods which address the problem of scheduling on-line multiple workflows, namely OWM [HHW11], RANK_HYBD [YS08] and FDWS, whose target is to minimize the average relative waiting time of the workflows. In previous sections, all these three algorithms are described. But none of these approaches consider cost as a QoS parameter in their scheduling strategies.

As we explained in the details of the FDWS scheduling algorithm, on-line scheduling strategies contain three main policies: filling the *ready tasks* pool, *task selection* policy to select a task among all available ready tasks to be executed, and *processor selection* policy, which finds the best suitable processor for a selected task by considering QoS objectives of task's workflow application. In chapter 7, we proposed two generic strategies for both task and processor selection phase for on-line scheduling of concurrent workflows with budget constraints defined by users for each workflow. In the following paragraphs, each proposed strategy is described in detail.

Proposed priority strategy for ready tasks in *ready tasks* pool

There is a *ready tasks* pool which is filled by the ready tasks belonging to each submitted and unfinished workflow at each scheduling round. In general, two methods are used to fill the *ready tasks pool*, first, like FDWS algorithm, gather only a single ready task with highest priority ($rank_u$) from each workflow, or insert all ready tasks belonging to each unfinished workflow application into *ready tasks* pool such as RANK_HYBD. But the important key is how to order these ready tasks, i.e., how to assign a priority to each ready task based on our QoS parameters to have higher quality solutions and system performance. To select a task from the *ready tasks* pool to be scheduled on the resources, we define a new strategy to assign a secondary priority

to each task of the *ready tasks* pool. Because our goal is to execute applications in the lowest turnaround time with its limited budget, the cost factor should be taken into account.

To assign the secondary priority to each task in *ready tasks* pool, we propose $rank_B$ for each task t_i in the pool, that belongs to workflow j, defined by equation 14. The task with the highest $rank_B$ is selected to be scheduled in the next phase, i.e processor selection phase.

$$rank_B(t_{i,j}) = \frac{1}{TP_j} \times \frac{1}{BP_j} \tag{14}$$

Here, TP_j defined as the Task Proportion of workflow j and it is calculated by the ratio of the number of unscheduled tasks to the total number of tasks in the workflow j. BP_j is the Budget Proportion and equals to the ratio of the Remaining Cheapest Budget (RCB_j) to the Remain Budget (RB_j).

$$BP_j = \frac{RCB_j}{RB_j} \tag{15}$$

Here, RCB is updated in each step *after* making the processor selection for the selected task belonging to workflow j, unlike the RCB definition in HBCS scheduling algorithm in the previous section which updated *before* executing the processor selection phase for the current task.

The $rank_B$ value is the product of two factors: (a) the first one is the inverse of the fraction of the workflow j that is remaining in the system; and (b) the ratio of the budget value over the remaining cheapest budget. This priority factor gives higher priority to the workflows that have a lower percentage of tasks unscheduled and to workflows that have higher budgets when compared to the cheapest budget for the workflow. The rational for the first factor is to give higher priority to workflows that were submitted earlier, so that a longer workflow with several tasks already executed may have priority over a short and recent workflow. And the rational of the budget factor is that the scheduler will consider first tasks that can spend more budget and therefore they will select more expensive and faster processors, resulting in a lower turnaround time for the workflow.

Proposed Processor Selection strategy

The Task scheduler has responsibility for selecting affordable resources for the current selected task. For each step of the processor selection phase, we select the task with highest $rank_B$ from *ready tasks* pool as the current task ($t_{curr,j}$) for scheduling. The processor to be selected to execute the current task is guided by the following strategy related to time and cost. To achieve minimum execution time under limited budget, we used two relative quantities defined previously in equations 10 and 11, namely $Time_r$ and $Cost_r$. In addition to these variables that give time and cost relative processor performance, there is a limitation on cost consumption. This constraint is represented

by the spare budget, that is defined by the difference between the remaining budget available (RB) and the remaining cheapest assignment (RCB). Once RCB includes the minimum cost of the current task (t_{curr}), this quantity is added to the spare budget available, as expressed by:

$$Cost_{lim}(t_{curr,j}) = Cost_{lowest}(t_{curr,j}) + (RB_j - RCB_j) \tag{16}$$

To select an affordable processor, the $Cost_{lim}$ value is used in order to cover the available budget for the current task. Additionally, selecting the processor with higher cost than the processor that gives the minimum finish time (p_{best}) is not logic. Therefore, P_v is defined as the set of reasonable and valid processors.

$$P_v = \{p_i \in P | Cost(t_{curr}, p_i) \leq Cost_{lim}(t_{curr,j})\} \tag{17}$$

Finally, to select the processor for the current task $t_{curr,j}$ belonging to workflow j, it is computed the Quality value (Q) for each available and free processor $p_i \in P_v$ as:

$$Q(t_{curr,j}, p_i) = \begin{cases} -\infty & Cost(t_{curr,j}, p_i) \\ & > Cost(t_{curr}, p_{best}) \\ Time_r(t_{curr,j}, p_i) + Cost_r(t_{curr,j}, p_i) \times BP_j & \text{otherwise} \end{cases} \tag{18}$$

The Budget Proportion (BP) is the ratio between RCB and RB, and gives a measure of how far the cheapest assignment is from the remaining budget available. If BP is near to one, it means that the available budget only allows to select the cheapest assignment. In addition, the $Cost_{lim}$ controls the processor's decision to avoid cost consumption higher than the user-defined budget.

In the Quality value (Q) equation, the selection of a processor with higher cost than the processor that gives the minimum finish time (p_{best}) is avoided by the first statement that guaranties that these type of processors cannot be selected. Otherwise, the processor is evaluated considering the time and cost quantities and the processor with higher Quality value will be selected for current task assignment.

Algorithm 4 shows the general algorithm for on-line scheduling for multiple workflow applications by optimizing execution time constrained to the user budget. We implemented modified versions of RANK_HYBD and FDWS, called Budget RANK_HYBD (B-RANK_HYBD) and Budget FDWS (B-FDWS) to consider budget limitation imposed by users. In the processor selection phase of these two algorithms, cost is not taken into account and there is a possibility to have higher cost than the limited budget defined by the user. So, in the modified version, instead of considering all processors to compute the finishing time of the current task, processors are filtered based on the cost limitation value

defined by p_v. To evaluate the influence of the new strategy proposed in this study, $rank_B$ for selecting tasks and the quality measure Q for selecting processors, we consider several versions of the scheduling algorithms as described in the table 2. For all algorithms, only processors that are free on current time (no reservation policy) are selected. Table 2 shows the characteristics of the algorithms, i.e. the strategies to select ready tasks from each workflow, the priority assigned to each ready task in the ready tasks pool and the processor selection policy. These policies are parameters of the general algorithm.

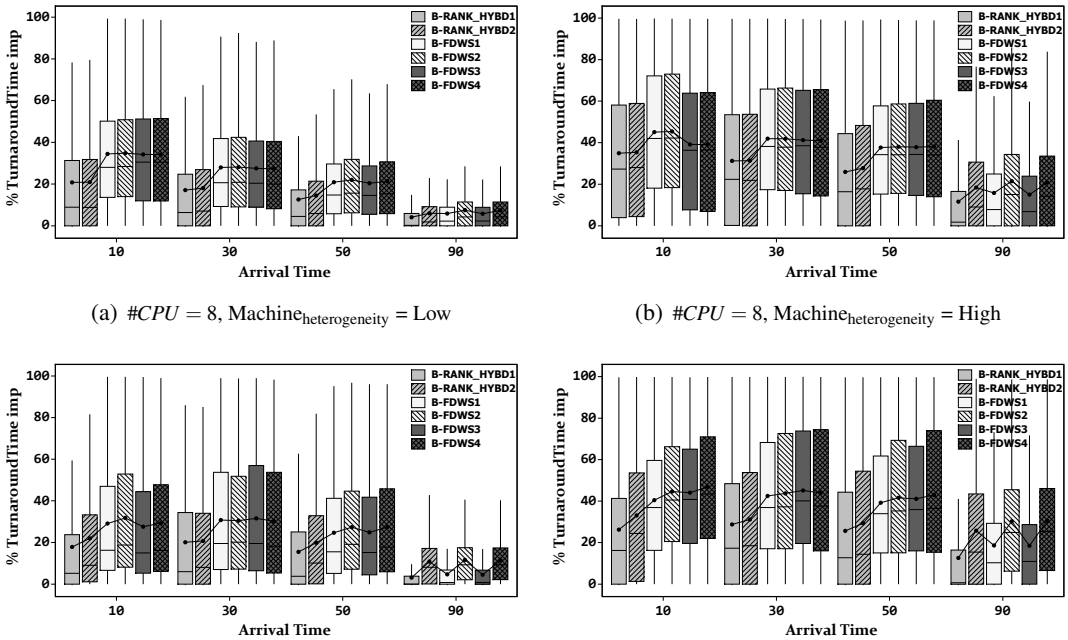
Algorithm 4 The General Budget Constrained Scheduling Strategies for On-Line Workflow Applications

1: **while** Application DB $\neq \phi$ **do**
2: Fill *Ready Tasks* pool based on the input filling strategy
3: **for all** $t_i \in$ *Ready Tasks* pool **do**
4: Assign a rank value for t_i according to the input priority strategy for ready tasks
5: **end for**
6: **while** *Ready Tasks* $\neq \phi$ **do**
7: Select current task $t_{curr,j}$ with highest priority from *Ready Tasks* pool
8: based on the input processor selection strategy, Select best suitable processor (p_{sel})
9: Assign current task $t_{curr,j}$ to selected Processor (p_{sel})
10: Update the Remain Budget (*RB*) and the Remain Cheapest Budget (RCB)
11: Remove current task $t_{curr,j}$ from *Ready Tasks* pool
12: **end while**
13: **end while**

Algorithm Name	Strategies		
	Filling Ready Pool	Selecting task to schedule	Processor Selection
B-RANK_HYBD1	**for each** *workflow* **Insert** *all ready tasks*	**if all** $t_i \in$ *ready pool* belong to *same workfolow* **then select** t_i with highest $rank_u$, **else select** t_i with lowest $rank_u$	$P_{sel} = \left\{ p_j \mid EFT(t_{curr}, p_j) = \min_{p' \in P_v} \{EFT(t_{curr}, p')\} \right\}$
B-RANK_HYBD2			$P_{sel} = \left\{ p_j \mid Q(t_{curr}, p_j) = \max_{p' \in P_v} \{Q(t_{curr}, p')\} \right\}$
B-FDWS1	**for each** *workflow* **Insert** *Single ready task* **with** highest $rank_u$	**select** $t_i \in$ *ready pool* **with** highest $rank_r$	$P_{sel} = \left\{ p_j \mid EFT(t_{curr}, p_j) = \min_{p' \in P_v} \{EFT(t_{curr}, p')\} \right\}$
B-FDWS2			$P_{sel} = \left\{ p_j \mid Q(t_{curr}, p_j) = \max_{p' \in P_v} \{Q(t_{curr}, p')\} \right\}$
B-FDWS3		**select** $t_i \in$ *ready pool* with highest $rank_B$	$P_{sel} = \left\{ p_j \mid EFT(t_{curr}, p_j) = \min_{p' \in P_v} \{EFT(t_{curr}, p')\} \right\}$
B-FDWS4			$P_{sel} = \left\{ p_j \mid Q(t_{curr}, p_j) = \max_{p' \in P_v} \{Q(t_{curr}, p')\} \right\}$

Table 2: Description of the modified algorithms for on-line budget constrained scheduling

To evaluate the algorithms we consider the relative improvement on Turnaround Time achieved with our strategy, for a given workflow, when compared to the maximum Turnaround Time achieved for that workflow among all strategies. The Turnaround time is the difference between submission and final completion of an application. The $TurnaroundTime_{imp}$ is obtained by the ratio of the difference of turnaround time for a given workflow obtained by an algorithm, and the maximum turnaround time



(a) $\#CPU = 8$, $\text{Machine}_{\text{heterogeneity}}$ = Low

(b) $\#CPU = 8$, $\text{Machine}_{\text{heterogeneity}}$ = High

(c) $\#CPU = 32$, $\text{Machine}_{\text{heterogeneity}}$ = Low

(d) $\#CPU = 32$, $\text{Machine}_{\text{heterogeneity}}$ = High

Figure 4: Turnaround Time improvement values for 8 and 32 processors and, for low and high machine heterogeneity

among all algorithms. Figure 4 shows the Turnaround Time percentage improvement ($TurnaroundTime_{imp}$) achieved for the 6 algorithm's versions.

For a low number of CPUs, as we can see in Fig 4(a) and Fig 4(b), the filling policy for adding ready tasks from workflow applications into ready tasks pool, is the strategy that differentiates the two algorithms RANK_HYBD and FDWS. The FDWS filling strategy, which selects a single task from each workflow, leads to higher fairness in scheduling and avoids the postponing of larger workflows as it happens with RANK_HYBD, contributing to a better relative turnaround time. The improvements are more significant when we have higher concurrency in the system, i.e. low arrival time, starting on 67% improvement for an arrival time interval of 10%, and 30% improvement for an arrival time interval of 90%.

As we move to higher arrival time intervals, when comparing the algorithms versions that use the quality measure Q, with the ones that do not use it, we conclude that Q improves the algorithms performance. For instance, B-RANK_HYBD2 has 46% improvement over B-RANK_HYBD1, as well as 27% improvement for B-FDWS2 over B-FDWS1 and 28% for B-FDWS4 over B-FDWS3.

For higher arrival time intervals, which means lower concurrency, using the quality measure Q achieves higher turnaround time percentage improvement on platforms with larger values of machine heterogeneity. We obtained improvements of 58%, 36% and 39% of turnaround time, with arrival time of 90% for B-RANK_HYBD2, B-FDWS2 and B-FDWS4 over B-RANK_HYBD1, B-FDWS1 and B-FDWS3, respectively.

On the other hand, for a higher number of CPUs, in addition to filling ready task policy, two other strategies, $rank_B$ and quality measure Q, proposed here, have higher influence in the improvements obtained with both algorithms. Fig. 4(c) and Fig. 4(d) show that, besides the filling ready task policy used by FDWS which improves algorithm performance over RANK_HYBD, quality measure Q always improves the algorithm's performance. The improvements of B-RANK_HYBD2, B-FDWS2 and B-FDWS4 over B-RANK_HYBD1, B-FDWS1 and B-FDWS3, respectively, start at 55% for an arrival time interval of 10% and increase to 240% for an arrival time interval of 90%.

Comparing results of B-FDWS1 to B-FDWS3 or B-FDWS2 to B-FDWS4, we can conclude that $rank_B$, as the policy for selecting the task from ready tasks pool to be schedule, improves the algorithm performances slightly, in comparison to $rank_r$. The highest improvement observed is 9%.

Single workflow applications, Budget-Deadline Constrained In chapter 8, a low-time complexity heuristic, named Deadline–Budget Constrained Scheduling (DBCS), is proposed to schedule workflow applications on computational heterogeneous infrastructures constrained to two QoS parameters, namely time and cost. The objective of the proposed DBCS algorithm is to find a feasible schedule map that satisfies the user defined deadline and budget constraint values. To fulfil this objective, DBCS implements a mechanism to control the time and cost consumption by each task when producing a schedule solution. To the best of our knowledge, the algorithm proposed here is the first low-time complexity heuristic for a bounded number of heterogeneous resources addressing two QoS parameters that obtains similar performances to higher-time complexity scheduling algorithms in a small fraction of the scheduling time.

The DBCS algorithm is a heuristic strategy that in a single step obtains a schedule that always accomplishes the budget constraint and that may or may not accomplish the deadline constraint. If the time constraint is met, we have a successful schedule; otherwise, we have a failure, and no schedule is produced. The algorithm is evaluated based on the success rate. As most heuristic scheduling algorithms, DBCS contains two main phases, *task selection* and *processor selection* phases. For *task selection* phase, tasks are prioritized by upward rank values and in each step of scheduling, the task with the highest priority is selected as current task (t_{curr}) to be scheduled in the next phase. In *processor selection* phase, the processor to be selected to execute the current task is guided by the following quantities related to cost and time. To control the consumed cost and time, a limit value for each factor is needed. We define two variables, CL and DL as limits for cost and time. To select the best suitable processor, a trade-off between these two variables is evaluated. In the following paragraphs, we describe these two variables in detail.

- $CL(t_{curr})$ is the maximum available budget for the current task t_{curr} that can be consumed by its assignment, and it is defined as the minimum cost for t_{curr} plus the spare

budget available:

$$CL(t_{curr}) = Cost_{min}(t_{curr}) + \Delta_{Cost} \tag{19}$$

where Δ_{Cost} represents the spare budget defined as the difference between unconsumed budget and cheapest cost assignment for unscheduled tasks. The initial value is $\Delta_{Cost} = BUDGET_{user} - Cost_{cheapest}$ where $BUDGET_{user}$ is the user defined budget as maximum allowed cost and $Cost_{cheapest}$, defined as $Cost_{cheapest} = \sum_{t_i \in T} Cost_{min}(t_i)$, is the cost of the cheapest assignment and represents the cost lower bound for executing the application. Δ_{Cost} is updated at each step *after* selecting the processor for the current task t_{curr}:

$$\Delta_{Cost} = \Delta_{Cost} - \Big[AC(t_{curr}) - Cost_{min}(t_{curr})\Big] \tag{20}$$

- $DL(t_{curr})$ is defined as the sub-DeadLine that is assigned to each task based on the total application deadline. There are some studies that proposed different strategies to distribute workflow deadlines among tasks. In [YBT05a], tasks are grouped in different levels based on their depth in the graph, and then the final deadline is divided into levels in such a way that all tasks belonging to the same level have the same sub-deadline. In [YBT+05b], first the original workflow is partitioned into sub-workflows, and then the total deadline is divided among partitions. In this paper, we apply the common and direct project planning sub-deadline distribution strategy. The sub-deadline value for each task t_i is computed recursively by traversing the task graph upwards, starting from the exit task. Due to heterogeneity, sub-Deadline can be defined in several different forms. Here, we consider the minimum execution time of the current task, defined as:

$$DL(t_{curr}) = \min_{t_{child} \in succ(t_{curr})} \Big[DL(t_{child}) - \overline{C}_{(t_{curr} \to t_{child})} - ET_{min}(t_{child})\Big] \tag{21}$$

  where ET_{min} is defined as the minimum execution time of task t_{curr} among available processors. For the exit task, the sub-deadline is equal to the user defined deadline, $DL(t_{exit}) = DEADLINE_{user}$.

  Unlike the cost limit, the sub-Deadline is a soft limit as in most deadline distribution strategies on grid platforms with a fixed number of available resources [YRB09]; if the scheduler cannot find a processor that satisfies the sub-deadline for the current task, the processor that can finish the current task at the earliest time is selected.

All available processors are filtered by $CL(t_{curr})$ to guarantee that the application can be executed without exceeding the budget constraint. We defined this filtered processor set as admissible processors, $P_{admissible}$.

$$P_{admissible} = \Big\{p_j \in P | Cost(t_{curr}, p_j) \leq CL(t_{curr})\Big\} \tag{22}$$

In the most restricted case, only the cheapest processors are considered. Otherwise, no feasible schedule exists under the user defined budget.

The processor selection phase is based on the combination of the two QoS factors, time and cost, to obtain the best balance between time and cost minimum values. We define two relative quantities, namely, Time Quality ($Time_Q$) and Cost Quality ($Cost_Q$), for current task t_{curr} on each admissible processor $p_j \in P_{admissible}$, shown in (23) and (24), respectively. Both quantities are normalized by their maximum values.

$$Time_Q(t_{curr}, p_j) = \frac{\Omega \times DL(t_{curr}) - FT(t_{curr}, p_j)}{FT_{max}(t_{curr}) - FT_{min}(t_{curr})} \tag{23}$$

$$Cost_Q(t_{curr}, p_j) = \frac{Cost_{best}(t_{curr}) - Cost(t_{curr}, p_j)}{Cost_{max}(t_{curr}) - Cost_{min}(t_{curr})} \times \Omega \tag{24}$$

where

$$\Omega = \begin{cases} 1 & \text{if} \quad FT(t_{curr}, p_j) < DL(t_{curr}) \\ 0 & otherwise \end{cases} \tag{25}$$

$Time_Q$ measures how much closer to the task sub-deadline (DL) the finish time of the current task on processor p_j is. Processors with higher $Time_Q$ values have a greater possibility of being selected. If the current task has a higher finish time on processor p_j than its sub-deadline, $Time_Q$ assumes a negative value for p_j, reducing the possibility of this processor being selected. Similarly, $Cost_Q$ measures how much less the actual cost on p_j is than the cost on the processor that results in the earliest finish time ($Cost_{best}$). Although CL is the maximum allowed cost for the current task, here, $Cost_{best}$ is used to avoid selecting a processor that performs worse and costs more than the processor that guarantees the earliest finish time.

In the case where none of the processors from $P_{admissible}$ can guarantee t_{curr} sub-deadline, $Cost_Q$ is zero for all of them, and $Time_Q$ for each processor p_j is a negative value that represents the relative finish time obtained with p_j. The processor from $P_{admissible}$ with higher $Time_Q$, i.e., closer to zero, would be selected. Note that, in any case, cost will be lower than CL, the maximum available budget for the current task.

Finally, to select the most suitable processor for the current task, the Quality measure (Q) for each processor $p_j \in P_{admissible}$ is computed:

$$Q(t_{curr}, p_j) = Time_Q(t_{curr}, p_j) \; + Cost_Q(t_{curr}, p_j) \times \frac{Cost_{Cheapest}}{Budget_{Unconsumed}} \tag{26}$$

Here the cost quality factor is weighted by the ratio of the cheapest cost execution for unscheduled tasks over the unconsumed budget, so that the effectiveness of the cost quality

factor can be controlled. A higher value of the fraction means that the unconsumed budget is close to the cheapest cost execution for unscheduled tasks, so that the cost factor is more predominant in the processor Quality measure. In the same way, a lower value means a higher difference between unconsumed budget and cheapest cost execution for unscheduled tasks, so that the cost factor is less influential, allowing the selection of more expensive processors that guarantee a lower processing time for t_{curr}.

The DBCS algorithm is shown in Algorithm 5. First, the possibility of finding a schedule map under a user defined budget is checked in lines 1-3. After some initializations in lines 4-5, the algorithm starts to map all tasks of the application (while looping in lines 6-14). At each step, on line 7, among all ready tasks, the task with highest priority ($rank_u$) is selected as the current task (t_{curr}). Then, in lines 8-10, the Quality measure for assigning t_{curr} to processor p_j ($Q(t_{curr}, p_j)$) is calculated. Note that, first, the finish time (FT) and execution cost of the current task is calculated and then the quality measure for all *admissible* processors is calculated. Next, the processor with the highest quality measure among all processors is selected (line 11-12). Finally, after assigning the processor to the current task, the Δ_{Cost} variable is updated using Eq.20 (line 13).

Algorithm 5 DBCS algorithm

Require: a DAG and user's QoS Parameters values for time ($DEADLINE_{user}$) and cost ($BUDGET_{user}$)

1: **if** $BUDGET_{user} < Cost_{cheapest}$ **then**
2: **return** no possible schedule map
3: **end if**
4: Initialize $\Delta_{Cost} = BUDGET_{user} - Cost_{cheapest}$
5: Compute the upward rank ($rank_u$) and sub-DeadLine value (DL) for each task
6: **while** there is an unscheduled task **do**
7: t_{curr} = the next ready task with highest $rank_u$ value
8: **for all** $p_j \in P_{admissible}$ **do**
9: calculate Quality measure $Q(t_{curr}, p_j)$ using Eq.26
10: **end for**
11: P_{sel} = Processor p_j with highest Quality measure (Q)
12: Assign current task t_{curr} to Processor P_{sel}
13: Update Δ_{Cost} using Eq.(20)
14: **end while**
15: **return** Schedule Map

In terms of time complexity, DBCS requires the computation of the upward rank ($rank_u$) and sub-DeadLines (DL) for each task that has complexity $O(n.p)$, where p is the number of available resources and n is the number of tasks in the workflow application. In the processor selection phase, to find and assign a suitable processor for the current task, the complexity is $O(n.p)$ for calculating FT and $Cost$ for the current task among all processors, plus $O(p)$ for calculating the Quality measure. The total time is $O(n.p + n(n.p + p))$, where the total algorithm complexity is of the order $O(n^2.p)$.

For comparison of the DBCS algorithm, we select four scheduling algorithms with the same objectives. Two algorithms, DCA[PW10] and GA[YB06b], are search-based strategies with high time complexity. The DCA[PW10] algorithm is based on dynamic programming to optimize two independent generic criteria for workflows, e.g., execution time and cost. The DCA scheduling algorithm has two main phases: the first selects one criterion as primary and optimizes it and, in the second phase, optimizes the secondary criteria while keeping the primary criteria within a defined sliding constraint. In particular, DCA performs a full domain search in the second phase of the algorithm. In GA[YB06b] scheduling algorithm, by using a genetic algorithm approach, two fitness functions are used to encourage the formation of individuals who satisfy the deadline and budget constraints. The other two algorithms, BHEFT[ZS12, ZS13] and LOSS1[SZTD07] are heuristic based approaches with low time complexity. The BHEFT algorithm optimizes the execution time of a workflow application constrained to a constraint budget. The BHEFT algorithm is not designed to cover the time constraint parameter as its objective, so in this case, if the makespan of the solution does not meet the user deadline, there is a failure. In LOSS algorithm, after initial assignment, the reassignment strategy is continued until the cost constraint is met, but at the same time we check the makespan in order to not exceed the deadline value.

To evaluate and compare our algorithm with other approaches, we consider the Planning Successful Rate (PSR) which is the percentage of successful schedules obtained in a given experiment. The main result is that the DBCS algorithm obtains similar performance to other state-of-the-art search-based algorithms with higher time complexity for the range of budget and deadline values considered here. As a heuristic algorithm, the main advantage of the DBCS consists in having an execution time in the range of the heuristic algorithms, such as BHEFT, but a planning success rate similar to the higher-time complexity search-based algorithms.

Concurrent workflow applications, Budget-Deadline Constrained

In chapter 9, a Multi-Workflow Deadline-Budget scheduling algorithm (MW-DBS) is proposed, to schedule multiple and concurrent workflow applications that may be submitted at different moments in time and with individual user's budget and deadline constraints. Workflow applications described by directed acyclic graphs present intrinsic parallelism among tasks so that their processing time can be optimized by a parallel task approach. However, the optimization process has limitations due to task dependencies. As common resource managers allocate a set of resources to execute a single workflow, in a user-centric approach, those resources cannot be fully utilized by a single job, incurring higher costs to the user without obtaining any improvement in the job processing time. In chapter 9, it is proposed a framework that executes simultaneously several workflow applications, where resources are shared among tasks. A user, when submitting a task, specifies a budget and a deadline for the task, in a range of values specified by the framework, and that can be accomplish with

the available infrastructure. The framework is dynamic (on-line), so that it can receive tasks at any moment in time.

The MW-DBS algorithm is a heuristic strategy and like other online schedulers contains two main steps: first, *task selection* phase that selects a task from each workflow and assigns a priority to each task based on the remaining time to the application deadline; and second, *processor selection* phase, where for the task with the highest priority, finds a suitable resource based on a quality measure computed to each resource. The common scheduling objective of concurrent applications is to increase the number of successful applications, but in addition to this objective, the proposed framework tries to increase the revenue of the provider by giving higher priority to jobs with higher budgets.

Generally, in most on-line scheduling systems, without an advance reservation, the scheduler is called when an executing task finishes and there is at least one free available processor to execute new tasks.

Task selection strategy

In general, for online concurrent workflow scheduling, in each scheduling step, there are many ready tasks from different submitted and unfinished workflow applications. So, the MW-DBS algorithm adds a single ready task from each unfinished workflow into the *ready tasks* pool and then select a suitable task to be executed among all tasks from this *ready tasks* pool. Another strategy to fill *ready tasks* pool is to insert all ready tasks belonging to each unfinished workflow application. Adding all ready tasks from each available workflow leads to an unfair strategy because the high number of ready tasks may cause that some workflow applications may not participate in the current scheduling round.

After filling the *ready task* pool, one is selected to schedule based on the QoS parameters defined to each application. The key point in the task selection phase is which task should be selected for scheduling among all ready tasks. In MW-DBS algorithm it is proposed a new strategy ($rank_D$) to assign a secondary priority to each task t_i belonging to workflow j in the *ready tasks* pool. Since we are dealing with both time and cost factors as our QoS parameters, the new priority assignment strategy considers both measures.

$$rank_D(t_{i,j}) = Cost^R_{t_{i,j}} \times \frac{1}{Time^R_{t_{i,j}} \times PRT_j} \tag{27}$$

where $Cost^R_{t_{i,j}}$ is the relative Cost Ratio of task t_i from workflow j and it is calculated as:

$$Cost^R_{t_{i,j}} = \frac{B_j}{CA_j} \tag{28}$$

where B_j is the cost constraint value (user's BUDGET) and CA_j is defined as Cheapest Assignment, i.e. all tasks from application j scheduled to cheapest processors.

$Time^R_{t_{i,j}}$ is the relative Time Ratio of task t_i from workflow j and it is calculated as:

$$Time^R_{t_{i,j}} = \frac{D_j - SD(t_i)}{D_j} \tag{29}$$

where D_j is the time constraint value (user's DEADLINE) and $SD(t_i)$ is defined as Sub-Deadline assigned to task t_i. We applied the common and direct project planning sub-deadline distribution strategy. $SD(t_i)$ is computed recursively by traversing the task graph upwards, starting from the exit task. Due to heterogeneity, sub-Deadline can be defined in several different forms. Here, we consider the average execution time of the current task, as shown by Eq(30):

$$SD(t_i) = \min_{t_{child} \in succ(t_i)} \left[SD(t_{child}) - \overline{TR}(t_{child}) \right] \tag{30}$$

where $\overline{TR}$ is defined as the average time reservation of task t_i among available processors. For the exit task, the sub-deadline is equal to the user defined deadline ($SD(t_{exit}) = D_j$). PRT_j is the Percentage Remaining Tasks of workflow j and calculated as:

$$PRT_j = \frac{Unscheduled\ tasks\ of\ workflow\ j}{Total\ tasks\ of\ workflow\ j} \tag{31}$$

The $rank_D$ priority value contains two major factors: a) the cost parameter which gives higher priority to the submitted and unfinished workflow applications that have higher budget ratio in order to maximize the provider profit; and b) the time parameter which contains two time measures, $Time^R$ and PRT. The first has the responsibility of assigning higher priority to workflows which have lower sub-deadlines. The second ensures that a workflow with few unscheduled tasks has higher priority. Finally, the task with highest $rank_D$ in *ready tasks* pool is selected to be schedule in the next phase.

Processor selection strategy

The processor selection phase has the responsibility of selecting an affordable resource for the current task (t_{curr}) and it is repeated until there is no more tasks left in *ready tasks* pool. A new strategy for processor selection phase based on QoS requirements is proposed. In order to control the consumed cost and time, a bound value for each factor is needed. Next, it is described bound values for cost and time, and then it is presented a new strategy for processor selection.

The Cost bound value ($Cost_{Bound}$) is a limitation on budget consumption by each task based on used budget, by previously scheduled tasks, and available budget for the current task that can be consumed by its assignment:

$$Cost_{Bound}(t_{curr}) = Cost_{min}(t_{curr}) + \Delta_j^{Cost} \tag{32}$$

where $Cost_{min}$ denotes the minimum execution cost of the current task among all processors and $\Delta_j^{Cost} = RB_j - RCA_j$ represents the spare budget defined as the difference between unconsumed budget and cheapest cost assignment for unscheduled tasks for workflow j which task t_{curr} belongs to. The remaining unconsumed Budget of workflow j (RB_j) has an initial value equal to the available user budget (B_j) and is updated at each step after selecting the processor for t_{curr} as shown in Eq(33), where $AC(t_{curr})$ is the Assigned Cost. Similarly, RCA_j is defined as Remaining Cheapest Assignment of workflow j with initial value equal to Cheapest Assignment (CA_j) and updated by Eq(34).

$$RB_j = RB_j - AC(t_{curr}) \tag{33}$$

$$RCA_j = RCA_j - Cost_{min}(t_{curr}) \tag{34}$$

All free available processors are filtered by $Cost_{Bound}(t_{curr})$ in order to guarantee that the application can be executed without exceeding the budget constraint. For the current assignment, we defined this set of acceptable processors as $P_{admissible}$. In the most restricted case, only the cheapest processors are considered. Otherwise, no feasible schedule exists under the user defined budget.

For the Time bound value it is used the sub-Deadline (SD), introduced in *task selection* phase, and it is a soft limitation as in most deadline distribution strategies for a fixed number of available resources [YRB09]; if the scheduler cannot find a processor that satisfies the sub-deadline for the current task, the processor that can finish the current task at the earliest time is selected.

The processor selection phase is based on a quality measure assigned to each processor that combines the QoS factors. Once there is no optimization step, each resource is evaluated in terms of the processing time and cost for the current task t_{curr}. Two quantities are defined, namely, Time Quality ($Time_Q$) and Cost Quality ($Cost_Q$), on each admissible processor $\acute{p} \in P_{admissible}$, shown in (35) and (36), respectively. Both quantities are normalized by their maximum values.

$$Time_Q(t_{curr}, \acute{p}) = \begin{cases} 1 - \dfrac{FT(t_{curr}, \acute{p})}{SD(t_{curr})} & \text{if } FT(t_{curr}, \acute{p}) < SD(t_{curr}) \\ 1 - \dfrac{FT(t_{curr}, \acute{p})}{FT_{min}(t_{curr})} & \text{otherwise} \end{cases} \tag{35}$$

$$Cost_Q(t_{curr}, \acute{p}) = \begin{cases} 1 - \dfrac{Cost(t_{curr}, \acute{p})}{Cost_{max}(t_{curr})} & \text{if } FT(t_{curr}, \acute{p}) < SD(t_{curr}) \\ 1 & \text{otherwise} \end{cases} \tag{36}$$

Where $FT_{min}(t_{curr})$ and $Cost_{max}(t_{curr})$ denote the minimum finishing time and the maximum execution cost of current task among all available processors.

Finally, to select the most suitable processor for t_{curr}, the Quality measure (Q) for each processor $\acute{p} \in P_{admissible}$ is computed as shown in Eq(37) and the processor with highest Q is selected.

$$Q(t_{curr}, \acute{p}) = Time_Q(t_{curr}, \acute{p}) \times Cost_Q(t_{curr}, \acute{p}) \tag{37}$$

Resources for which the finishing time is lower than the deadline ($FT(t_{curr}, \acute{p}) < SD(t_{curr})$), $Time_Q$ measures how much the finishing time of the current task on a processor is closer to the task sub-deadline. The processor with higher $Time_Q$ has higher possibility to be selected. Otherwise, $Time_Q$ assumes a negative or zero value, reducing the processor quality value. Processors with a lower cost to execute t_{curr} have a higher $Cost_Q$, increasing their quality measure. However, for the processors that do not cover the sub-deadline, the processor with the lowest finishing time should be selected regardless of what cost it has. In this case, $Cost_Q$ is set to 1, being the processors quality only influenced by $Time_Q$. It should be noted that in both cases, the tested processors are selected from admissible processor list so that it can be guaranteed that the application can be executed without exceeding its budget constraint.

The MW-DBS algorithm is shown in Algorithm 6. First, all ready tasks in the *ready tasks* pool are ranked by $rank_D$ priority value (Eq.27). To fill the *ready tasks* pool, the framework collects a single ready-to-execute task with the highest primary rank value $rank_u$ [THW02] from each submitted and unfinished workflow application. Until there is at least one ready and unscheduled task in the *ready tasks* pool and free available processors, the current task t_{curr} is selected and its quality measure Q (Eq.37) is calculated among all admissible processors ($P_{admissible} \subset P_{free}$). Then, the current task t_{curr} is assigned to processor P_{sel} that has the highest quality measure. Then, the Remaining unconsumed Budget (RB) and Remaining Cheapest Assignment (RCA) are updated for workflow j where task t_{curr} belongs to.

Algorithm 6 MW-DBS algorithm

1: **for all** $t_{i,j} \in Ready\ Tasks$ pool **do**
2: Assign a priority rank $rank_D(t_{i,j})$
3: **end for**
4: $P_{free} \leftarrow$ free processors $\acute{p} \in P$
5: **while** $(Ready\ Tasks \neq \phi\ \&\ P_{free} \neq \phi)$ **do**
6: $t_{curr} \leftarrow$ task with highest $rank_D$
7: **for all** $\acute{p} \in P_{admissible}$ **do**
8: calculate Quality measure $Q(t_{curr}, \acute{p})$
9: **end for**
10: $P_{sel} \leftarrow$ Processor $\acute{p}$ with highest Q
11: Assign current task t_{curr} to Processor P_{sel}
12: Update RB_j and RCA_J
13: $P_{free} \leftarrow P_{free} - P_{sel}$
14: Remove Task t_{curr} from $Ready\ Tasks$ pool
15: **end while**

In terms of time complexity, MW-DBS requires the computation of the upward rank ($rank_u$) and Sub-DeadLines (SD) for each task that have complexity $O(n.p)$, where p is the number of available resources and n is the number of tasks in the workflow application. In the processor selection phase, to find and assign a suitable processor for the current task, the complexity is $O(n.p)$ for calculating FT and $Cost$ for current tasks among all processors, plus $O(p)$ for calculating the Quality measure. The total time is $O(n.p+n(n.p+p))$, where the total algorithm complexity is of the order $O(n^2.p)$.

For the comparison of the MW-DBS algorithm, we select three algorithms, FDWS2[AB14b] and the modified versions of MIN-MIN and MAX-MIN, called MIN-MIN* and MAX-MIN*. We consider a low number of processors compared to the number of DAGs to analyze the behavior of the algorithms in a higher concurrent environment. The maximum load configuration is observed for 8 processors and 50 DAGs.

The results show that the MW-DBS algorithm obtains performances in terms of Planning successful rate (PSR) among all compared algorithms. The main advantage of the MW-DBS algorithm occurred for low time intervals that shows significant performance improvement unlike the other algorithms. Increasing the time intervals, between the DAGs arrival times, reduces the concurrency, and thus, the improvements are less significant.

From the service provider's viewpoint, the major key is how much revenue is made. And we compare results based on Profit metric which is defined as the ratio of the total cost achieved by the algorithm and the maximum total cost among all algorithms. In this context, the MW-DBS algorithm shows better profit value over other compared algorithms.

1.6 Conclusion and Future Work Perspectives

Heterogeneous computing systems provide a global infrastructure for solving large-scale problems in science and business. Heterogeneous computing systems enable the sharing of geographically distributed heterogeneous resources. Resource management for executing workflow applications has been recognized as an important component for heterogeneous computing systems. Resource management is a way to execute and monitor workflow applications in grid infrastructures in order to improve the performance of the system. However, the users' requirements for their submitted applications should be considered in the resource manager. Therefore, deploying a workflow scheduling to meet users' QoS requirements on heterogeneous resources is a challenging task. The scheduling problem was identified as NP-complete, and there is no approach that could give us the optimal solution in a bounded time period. Generally, for NP problems, to achieve a good solution near the optimal one, the search-based or meta-heuristic approaches are used. However, in these types of approaches, i.e., search-based and meta-heuristic algorithms, by increasing the number of iterations or by increasing the searching domain, a better solution may be found. However, this usually results in a higher time complexity, making these methods less useful in real infrastructures. On the other hand, using heuristic methods with low time complexity gives us an affordable solution that may not be as good as those achieved by search-based approaches. The main objective of this thesis is to develop and propose scheduling techniques with low time complexity but that achieve good performances by obtaining schedule results comparable to those of search-based and meta-heuristic algorithms.

This thesis began by introducing a generic workflow scheduling system model and characterizing several components of the system such as a workflow application model, a platform and system model and scheduling problems. We considered two major classes of scheduling problems, namely, single and multiple workflow resource management, and taxonomies for each of these classes were provided. The quality of a workflow scheduling algorithm is measured by two main aspects: a) the quality of solutions, which indicates the user's QoS satisfaction and the system performance, and b) the ability to use the proposed approaches in a real platform that is dependent on its time complexity to produce a solution. Unlike most existing workflow scheduling approaches, which only target one aspect of the scheduling problem in their development, in this thesis, for all proposed approaches, both target aspects are considered by proposing scheduling approaches that can achieve good solutions with low time complexity.

The main contribution of the proposed algorithms in this thesis is the development of approaches that achieve performances comparable to those of previous approaches in the state-of-the-art with low time complexity. We mainly classified scheduling approaches in two main categories: Single and Multiple workflow application scheduling algorithms. Each category is classified into subcategories based on two conflicting QoS requirements: time and cost. Also, time and cost can be optimized or considered as a problem constraint.

In the first step, in chapter 2, a comparison is made of scheduling algorithms. We compared heuristic list-based scheduling approaches with search-based and meta-heuristic approaches,

namely, Tabu Search, Simulated Annealing, and the Ant Colony System. The results showed that the meta-heuristic scheduling algorithms, which feature a higher processing time, always achieved better solutions than the list scheduling heuristics with quadratic complexity. After deep consideration of their processes and comparing the results for schedule maps side by side with the results achieved by heuristic methods, we determined that for the inconsistent model of task execution time for workflow application, the processor selection phase should consider the forecasting of the finishing time of children for the current task, such as the Lookahead approach, in order to improve the total execution time of a given workflow application. However, this feature significantly increases the complexity. Therefore, to keep the ability to forecast while maintaining quadratic time complexity, in chapter 3, we proposed the Predict Earliest Finish Time (PEFT) scheduling algorithm for heterogeneous computing systems. The PEFT scheduling algorithm, unlike other state-of-the-art algorithms that use the Earliest Finish Time (EFT) measure to select a suitable processor for each task, introduced a look-ahead feature without increasing the time complexity associated with the computation of an optimistic cost table (OCT). The results show that the PEFT scheduling algorithm outperforms the state-of-the-art list-based algorithms for heterogeneous systems in terms of schedule length ratio, efficiency, and frequency of the best results while maintaining quadratic time complexity. Please note that the PEFT scheduling algorithm is designed and proposed for inconsistent models of execution time, as shown in the paper. To consider multiple workflow application scheduling, in chapters 4 and 5, the Fairness Dynamic Workflow Scheduling (FDWS) for scheduling multiple and concurrent workflow applications was proposed. FDWS is an on-line scheduling approach that dynamically schedules workflow applications submitted at different moments in time in order to improve the turnaround time for each individual application instead of decreasing the overall execution time of all submitted workflows. To include more QoS parameters as objectives of the scheduling approach, in this thesis, two conflicting QoS objectives, time and cost, are considered. In chapter 6, the Heterogeneous Budget Constrained Scheduling (HBCS) algorithm, which minimizes execution time while constrained to a user-defined budget, is proposed. In the HBCS algorithm, for the processor selection phase, a new strategy for selecting suitable resources to execute tasks is proposed. The HBCS algorithm is a heuristic strategy and was shown to achieve lower makespans for all of the budget factors. Moreover, a reduction of up to 30% in execution time was achieved while maintaining the same budget level compared to other approaches. For multiple and concurrent workflow scheduling with the same target, i.e., time-optimization and budget-constraint, in chapter 7, we proposed two generic strategies for both the task and processor selection phases for on-line scheduling of concurrent workflows with budget constraints defined by users for each workflow. In the next step, we considered time as a QoS constraint parameter for scheduling objectives. In chapter 8, a low-time complexity heuristic named Deadline–Budget Constrained Scheduling (DBCS) was proposed to schedule workflow applications on computational heterogeneous infrastructures constrained to time and cost as two conflicting QoS parameters. To find a feasible schedule map that satisfies the user-defined deadline and budget constraint values, DBCS implements a mechanism to control the time and cost consumption by each task when producing a schedule solution. The

DBCS algorithm is a heuristic strategy, and compared with other state-of-the-art search-based algorithms with higher time complexity, it obtains similar performance for the range of budget and deadline values. As a heuristic algorithm, the main advantage of the DBCS consists in having an execution time within the range of the heuristic algorithms but a planning success rate similar to the higher time complexity search-based or meta-heuristic scheduling algorithms. In chapter 9, a Multi-Workflow Deadline-Budget scheduling algorithm (MW-DBS) was proposed to schedule multiple and concurrent workflow applications that may be submitted at different moments in time and with individual user's budget and deadline constraints. In the MW-DBS algorithm, in addition to increasing the number of successful applications that meet users' QoS parameter constraint values, it aims to increase the obtained revenue for providers. The MW-DBS consists of two steps: first, it selects a task from each ready workflow and assigns a priority to each task based on the remaining time for the application deadline and budget value; second, for the high priority task, it selects a suitable resource based on a quality measure computed to each resource based on the job QoS parameters and provider profit.

1.6.1 Future Directions

Resource management in heterogeneous computing systems has advanced significantly in recent years. This thesis has contributed with substantial improvements in this field by proposing heuristic strategies for QoS-based workflow application scheduling problems. All proposed strategies in this thesis were designed with a low time complexity approach and show a similar or better improvement in the quality of results compared with other approaches with the same objective. Nevertheless, there are a number of open research challenges that need to be addressed and that can serve as a starting point for future research:

- supporting multiple QoS objectives for scheduling: the proposed scheduling strategies in this thesis mainly focus on time and cost as QoS parameters for workflow applications. However, other parameters such as reliability, security, availability and accessibility may also be required by many applications. For example, a service with high reliability may incur higher execution costs but may reduce the risk of execution failure during the application execution process, and therefore, it should be considered in the scheduling approach.

- return a set of solutions instead of a single one: increasing the number of QoS parameters causes difficulty in finding a single solution. Usually, in this context, it would be better if the workflow planner advises the user by recommending a set of solutions based on his/her QoS requirements. In this scenario, there are several algorithms that provide a set of solutions as pareto-front to give more options to the user to select his/her desirable solution for the application. One challenge in this field is to design and propose strategies that could cover more separated points in the solution space according to user QoS parameters.

- dynamic model of resources: this thesis has presented the scheduling approach on heterogeneous computing systems. The proposed approaches can produce promising optimized

task-to-resources maps according to the user's QoS demands. However, one of our assumptions for the platform was availability of resources without any failure during application execution. Once this assumption has been violated, the scheduler needs to modify and adopt the resource assignment for the remaining unexecuted tasks in order to meet the application's QoS requirements. This monitoring should be done during workflow execution. When the execution of one task is jeopardized due to resource failure, it reschedules and adopts resource assignment for the rest of the tasks in the workflow and finds an alternative resource to execute the task.

- scheduling non-DAG workflow applications: in this thesis, the considered workflow application model is Directed Acyclic Graphs (DAGs), which is based on advance information of tasks and their dependencies. Also, in this model, the workflow application does not contain any loop or conditional branch. This type of assumption is applied for many scientific applications [JCD+13]. However, many other applications contain loops and condition checking, which is required for their run-time scheduling monitoring and decision-making.

- supporting cloud platforms with a different pricing model: the work of scheduling problems in cloud computing environments is also of interest to us. The main difference between grid and cloud infrastructures comes from the virtualization of computing and storage resources. In a grid platform, all available resources have the same operating system, and similar software environments are installed. However, this uniform configuration brings limitations for both users and providers. The interesting feature of resource virtualization in cloud infrastructures is that each virtual machine can run different customized system images, i.e., operating system and software. Furthermore, cloud computing can create, pause and stop VMs in dynamic ways to be adapted to the workload. The users will be charged for what they use, but with a different billing model. One of the major concerns when moving to clouds is related to the billing model, in which users are charged based on hourly usage of requested VMs. Additionally, in cloud platforms, Service Level Agreements (SLAs), which include QoS requirements, are set up between customers and cloud providers. SLA describes the characteristics of cloud services and QoS requirements such as pricing model, usage model, billing and monitoring that are agreed upon between the customers and the cloud providers. When a service provider is unable to meet the terms stated in the SLA or when a QoS parameter defined by a customer is not satisfied, the SLA is violated. In this context, studying how to make scheduling decisions such that cloud service providers can guarantee QoS satisfaction and prevent SLA violations could be an interesting subject. Recently, federated clouds have brought new opportunities when using commercial clouds, as different providers may offer resources with different performances and pricing models. In such situations, low-priority applications could be executed on the slow resources offered by cheap providers, and high-priority applications could be executed on the fast resources offered by expensive providers. Thus, scheduling on this type of infrastructure become more challenging when additional limitations—such as the different geographical

locations of providers, which affects data-intensive applications—posed by federated clouds are considered.

Chapter 2

Performance Evaluation of List Based Scheduling on Heterogeneous Systems

Hamid Arabnejad and Jorge G. Barbosa

Euro-Par 2011: Parallel Processing Workshops (HeteroPar),

volume 7155 of Springer LNCS, pages 440–449, 2012,

DOI: 10.1007/978-3-642-29737-3_49

abstract

This paper addresses the of evaluating the schedule produced by list based scheduling algorithms, with metaheuristic algorithms. Task scheduling in heterogeneous systems is a NP-problem, therefore several heuristic approaches were proposed to solve it. These heuristics are categorized into several classes, such as list based, clustering and task duplication scheduling. Here we consider the list scheduling approach. The objective of this study is to assess the solutions obtained by list based algorithms to verify the space of improvement that new heuristics can have considering the solutions obtained with metaheuritcs that are higher time complexity approaches. We concluded that for low Communication to computation rate (CCR) of 0.1, the schedules given by the list scheduling approach is in average close to metaheuristic solutions. And for CCRs up to 1 the solutions are below 11% worse than the metaheuristic solutions, showing that it may not be worth to use higher complexity approaches and that the space to improve is narrow.

2.1 Introduction

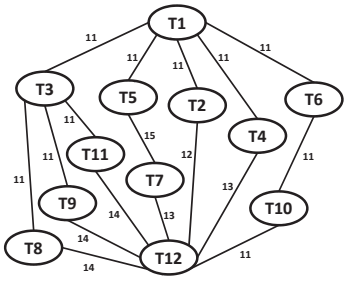
The problems of task matching and scheduling, in general, are to resolve a composite parallel program into several tasks and assign these tasks to a set of processor elements (PEs) to execute. These tasks have restriction of priority order to execute with each other due to its characteristic of data dependencies. The relationship among the tasks can be represented by a weighted Direct Acyclic Graph (DAG). Also, the processing elements are connected by a high speed communication network. Task matching is to assign a specific task to a suitable processing element to execute; and scheduling is to determine execution priority of each task among the composite parallel program. The general form of the problem has already been proved to be $NP-complete$ [CB76, KS74, LP+97, PY79]. Although it is possible to formulate and search for the optimal solution, the feasible solution space quickly becomes intractable for larger problem instance. To overcome the exponential time complexity, heuristic based scheduling algorithms of been proposed that found a sub-optimal solution in polynomial time. These heuristics are categorized into several classes, mainly list based, clustering and task duplication scheduling. Among these, list scheduling algorithms are generally regarded as having a good cost performance trade-off because of their low cost and acceptable results. In list scheduling, tasks are sorted by their priorities and scheduled accordingly [DAYA02, ERL90, THW02, IÖF95, KY94, KA96, LPX05, PK01, SS04]. Although these algorithms can find a feasible solution in polynomial time they are not able to guarantee to find a suitable solution when size of the problem becomes large.

In this paper we evaluate the quality of the solutions obtained by two best list scheduling algorithms, namely HEFT and CPOP [THW02], for heterogeneous systems by comparing with the solutions obtained by metaheuristic algorithms. Once these last algorithms do not guarantee the optimal solution, we obtain for each scheduling the best solution and measure the distance to the list scheduling solution. The metaheuristic algorithms considered in this study are Ant Colony System (ACS), Simulated annealing (SA) and Tabu Search (TA).
At first, we introduce the DAG scheduling problem,then describe two static list scheduling algorithms, HEFT and CPOP [THW02]. Followed by an introduction to the Ant Colony System, Simulated Annealing and Tabu Search. Further, we describe the design and the implementation on these algorithms with a discussion about the results achieved.

2.2 DAG Scheduling

A scheduling system model represented by a direct acyclic graph (DAG), $G = (V, E, P, W, data, rate)$, where V is set of v tasks, E is the set of e edges between tasks, and P is the set of processors available in the system. Each $edge(i, j) \in E$ represents the task-dependency constraint such that task n_i should complete its execution before task n_j can be started. A task with no predecessors is called an $entry$ task, n_{entry}, and n_{exit} is one with no successors. W is a $v \times p$ computation cost matrix, where v is the number of tasks and p is the number of processors in the system. Figure 1 shows an example of a DAG comprising 12 tasks to

illustrate these definitions graphically. It can be seen that the immediate successors of t_3 are t_8, t_9 and t_{11}; the immediate predecessors of t_{10} is t_6. Furthermore, t_1 is an entry task and t_{12} represents a pseudo exit-task.



	P1	P2	P3	P4
T1	7	13	11	23
T2	17	23	10	9
T3	7	20	14	16
T4	24	8	21	13
T5	24	14	21	17
T6	21	11	8	13
T7	23	9	10	24
T8	7	18	19	24
T9	14	9	24	24
T10	19	10	19	8
T11	21	15	7	25
T12	6	13	20	10

Figure 1: Example of a DAG and its computation costs matrix [CCR=0.8]

Each $w_{i,j}$ gives the estimated execution time to complete task n_i on processor p_j. The average of execution cost of a node n_i is defined $\overline{w_i} = (\sum_{j \in P} w_{i,j})/p$. The $data$ parameter is a $v \times v$ matrix of communication data, where $data(i,j)$ is the amount of data required to be transmitted from task n_i to task n_j. The $rate$ parameter is a $p \times p$ matrix and represent the data transfer rate between processors. The communication cost of $edge(i,j)$, which is for data transfer from task n_i (scheduled on processor p_m) to task n_j (scheduled on processor p_n), is defined by $c_{i,j} = data(n_i, n_j)/rate(p_m, p_n)$. When both n_i and n_j are scheduled on the same processor ($p_m = p_n$), then $c_{i,j}$ becomes zero. The average communication cost of an edge is defined by $\overline{c_{i,j}} = data(n_i, n_j)/\overline{rate}$, where $\overline{rate}$ is the average transfer rate between the processors in the domain.

The $EST(n_i, p_j)$ and $EFT(n_i, p_j)$ are the *Earliest Execution Start* time and the *Earliest Execution Finish* time of node n_i on processor p_j. For the entry task $EST(n_{entry}, p_j) = 0$. For other tasks, the EST and EFT values are computed recursively, starting from the entry task as shown by

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

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

where $pred(n_i)$ is the set of immediate predecessor tasks of task n_i and $T_{Available}(p_j)$ is the earliest time at which processor p_j is available for task execution. The inner *max* block in the EST equation returns the *ready time*, i.e, the time when all data needed by n_i has arrived at the processor p_j.

The *objective function* of the scheduling problem is to determine the assignment of task of a given application to processors such that the schedule length or *makespan* is minimized. After a task n_i is scheduled on processor p_j, the Actual Start Time of node n_i ($AST(n_i)$) is equal to $EST(n_i)$ and the Actual Finish Time of node n_i ($AFT(n_i)$) is equal to $EFT(n_i)$. After all nodes in the DAG are scheduled, the schedule length will be $makespan = max[\,AFT(n_{exit})]$, i.e. the Actual Finish Time of exit task.

The *Critial Path* (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 cost 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.

2.3 List scheduling Algorithms

The list scheduling technique [KA99] has the following steps: a) determine the available tasks to schedule, b) assign a priorities to them and c) until all tasks are scheduled, select the task with the highest priority and assign it to the processor that allows the earliest start time.

Two attributes frequently used to define the tasks priorities are the *upward* and the *downward* ranks. The *upward rank* of a node n_i ($rank_u$) is defined as the length of the longest path from an entry node to n_i (excluding n_i). The *downward rank* of a node n_i ($rank_d$) is the length of the longest path from n_i to an exit node. The nodes of the DAG with higher $rank_d$ values belong to the critical path.

2.3.1 HEFT Algorithm

The HEFT (Heterogeneous Earliest Finish Time) algorithm [THW02] is highly competitive in that it generates a comparable schedule length to other scheduling algorithms, with a low time complexity. The HEFT algorithm is an application scheduling algorithm for a bounded number of heterogeneous processors, which has two major phases: a *task prioritizing* phase for computing the priorities of all tasks and a *processor selection* phase for selecting the tasks in the order of their priorities and scheduling each selected task on its best processor, which minimizes the task's finish time. In HEFT algorithm, tasks are ordered by their scheduling priorities that are based on upward ranking ($rank_u$).

Algorithm 1 The HEFT algorithm

```
Compute rank_u(n_i) for all n_i ∈ V
ReadyTaskList ← Start Node
while ReadyTaskList ≠ Empty do
  n_i ← node with the maximum rank_u in ReadyTaskList
  for all p_j ∈ P do
    Compute EST(n_i, p_j)
    EFT(n_i, p_j) ← w_{i,j} + EST(n_i, p_j)
  end for
  Map node n_i on processor p_j which provides its least EFT
  Update T_Available(p_j) and ReadyTaskList
end while
```

2.3.2 CPOP Algorithm

The critical path (*CP*)is the longest path in a DAG. The Critical Path on Processor (CPOP) algorithm is a variant of the HEFT algorithm [THW02]. CPOP adopts a different mapping strategy for the critical path nodes and the non-critical path nodes. A *CP* processor is defined as the processor that minimizes the overall execution time of the critical path assuming all the critical path nodes are mapped onto it. If the selected node is a critical path node, it is mapped onto the *CP* processor. Otherwise, it is mapped onto a processor that minimizes its EFT (like in the HEFT algorithm).

Algorithm 2 The CPOP algorithm

```
Compute rank_u(n_i) and rank_d(n_i) for all n_i ∈ V
Identify the Critical Paths and mark the Critical Path Nodes
priority(n_i) ← rank_u(n_i) + rank_d(n_i)
ReadyTaskList ← Start Node
while ReadyTaskList ≠ Empty do
  n_i ← node with the maximum rank_u in ReadyTaskList
  if n_i ∈ Critical Path then
    Map n_i on the CP Processor
  else
    for all p_j in P do
      Compute EST(n_i, p_j)
      EFT(n_i, p_j) ← w_{i,j} + EST(n_i, p_j)
    end for
    Map node n_i on processor p_j which provides its least EFT
  end if
  Update T_Available(p_j)
  Update ReadyTaskList
end while
```

2.4 Metaheuristic Algorithms

2.4.1 Ant Colony System

Ant colony system (ACS) is a metaheuristic that was first proposed by Dorigo and Gambardella [DG97], it is one of the most popular swarm inspired methods in computational intelligence areas. And latter adapted to discrete optimization problems [DCG99]. The basic idea is to imitate the cooperative behaviour of real ants, to solve optimization problems. At first, ants have no clue about which way belongs to the shortest path to nest, so they choose randomly. Once the ants discover a paths from nest to food, they changed pheromone on the path. So another ants can follow the trails to find the food source. The ants that found the shortest path will come back to nest sooner, than ants via longer paths, and that path will have higher traffic. As this process continuous, the shortest paths have a huge amount of pheromone and most of ants tend to choose these paths. ACS includes five steps: (1) ants initialization to positioning (2) for each ant applied a state transition

rule to incrementally build a solution and a local pheromone updating rule (3) Global pheromone updating (4) ending test to evaluate the best solution that if it is not acceptable go to step 1.

To apply the ACS meta-heuristic to the task scheduling problem, we need to translate this problem into the structure of ACS so that ants can find solutions. For this purpose, we considered a Graph with two subgraphs G_1 and G_2, where the first represents the set of tasks to schedule and the second denotes the set of processors available. At each iteration, each ant selects a source node and a suitable processor based on a selection rule. Then we add tasks that are ready to schedule, i.e. tasks where their predecessors have been scheduled, and this procedure continues until all task are scheduled.

In ACS (Ant Colony System) the state transition rule provides a direct way to balance between exploration of new edges and exploitation of a priori and accumulated knowledge about the problem. It is defined as follows: an ant positioned on task i chooses the processor u to move to by applying the rule given by

$$Prob(i,p) = \begin{cases} \max\left[\tau(i,p) \times [\eta(i,p)]^{\beta}\right] & \text{if } q_0 < q(\text{exploitation}) \\ \dfrac{\tau(i,p) \times [\eta(i,p)]^{\beta}}{\sum_{q \in P} (\tau(i,q) \times [\eta(i,q)]^{\beta})} & \text{otherwise (biased exploration)} \end{cases}$$

where q is a random number uniformly distributed in $[0..1]$ and q_0 is a parameter $(0 \leq q_0 \leq 1)$. Tuning the parameter q_0 allows modulation of the degree of exploration and the choice of whether to concentrate the search of the system around the best-so-far solution or to explore other tours, here $q_0 = 0.7$. And $\eta(n,p) = 1/AFT(n_{i,p})$ is the heuristic function and $\beta = 2$ is a parameter which determine the relative influence of the heuristic information.

The *Global Pheromone Update* Rule is performed only by the best ants that have the shortest path from source to sink. This rule besides the use of the pseudo-random-proportional rule, cause to encourage the ants in next iterations to search in a neighbourhood of the best path found up to current iteration. After all ants finished their tour, we can perform global updating for current iteration. The pheromone level is updated by applying the global updating rule $\tau(i,p) = (1-\rho) \cdot \tau(i,p) + \rho \cdot \Delta\tau(i,p)$ where $\Delta\tau(i,p)$ for global best tour is $\Delta\tau(i,p) = 1/[AFT_{best\ ant}(n_{exit})]$ and for other nodes is $\Delta\tau(i,p) = 0$. Also, $0 < \rho < 1$ is the pheromone decay parameter and here is $\rho = 0.1$. In addition to the global pheromone trail updating rule, in ACS the ants use a *Local Pheromone Update* rule in each iteration since each ant by choosing a processor p for task i, is applied by $\tau(i,p) = (1-\xi) \cdot \tau(i,p) + \xi \cdot \tau_0$ where $0 < \xi < 1$ denotes the pheromone decay parameter and $\tau_0 = \frac{1}{|V|}$ is the initial value of pheromone on all edges. Experimentally, a good value for ξ was found to be $\xi = 0.1$.

2.4.2 Simulated annealing

Simulated Annealing (SA) is a generic probabilistic meta-algorithm proposed by Kirpatrick, Gelett and Vecchi [KGV+83] and Cerny [Čer85] used to find an approximate solution to global optimization problems. It is inspired by annealing in metallurgy which is a technique of controlled cooling

of material to reduce defects. In simulated annealing, a cost function to be minimized is defined in terms of the parameters of the problem at hand. The cost minimization process is governed by a cooling temperature which varies from a given high value to a low value slowly. At every temperature, we generate a fixed number of scheduling and calculate cost function(makespan) for each of them. If the cost function is less than the previous cost, the new configuration is accepted. If the cost is more than the previous one, the new configuration is chosen with a probability $r \leq exp(-\Delta C/T_k)$ where $r \in [0,1]$. Probabilistic acceptance of costlier solutions is behind the success of the simulated annealing process. Actually, when $\Delta C \leq 0$, we have a *downhill* step, that means a search for a new solution around a best solution. But if this condition is not satisfied, we can use the new solution instead of the best solution, with higher cost (*uphill* step) and helps the solution process overcome the possibility of getting trapped in a local minimum and move toward the global minimum. The three most important parts are: (1)*Cost Function* that is the schedule length of the solution; (2) *Generating mechanism* to randomly generate a scheduling of a set of tasks; and, (3) *Cooling mechanism* that initializes the temperature to a value T_0, and in each step, it decreases by $T_{k+1} = \alpha \times T_k$ and $\alpha = 0.1$. if you chose higher, you will have less exploration of the search space and move faster to final temperature. In our implementation the length of Markov chain is $|V|$, final temperature is 0.01, initial temperature is $\left[best_{makespan}(S_i) - worst_{makespan}(S_i)\right] / \log(0.9)$, where S_i is the initial solution.

Algorithm 3 The Simulated annealing algorithm

```
Create an initial(feasible) solution s;
Set an initial temperature T_0 (with k ← 0);
Set number of trials at each temperature level (level-length) α
while termination criterion not satisfied do
  for i = 1 → length_{Markov chain} do
    Create new neighbor s' by applying a random move to s;
    Calculate cost difference ΔC between s' and s : ΔC = C(s') − C(s);
    if ΔC ≤ 0 then
      Switch over to solution s' (current solution s is replaced by s');
    else
      Create random number r ∈ [0,1];
      if r ≤ exp(−ΔC/T_k) then
        Switch over to solution s' (current solution s is replaced by s');
      end if
    end if
  end for
  Update best found solution (if necessary);
  Set k ← k + 1 and Set / Update temperature value T_k for next level k;
end while
return Best found solution
```

2.4.3 Tabu Search

Tabu search (TS) is one the a heuristic methods proposed by Glover [Glo89] [Glo90]. Unlike other meta-heuristics, in TS, we have an intelligent search to perform a systematic exploration of the solution space. The main idea in TS is to use the information about search history to guide local search approaches to overcome local optimality. In general we examine a path sequence of solutions and moves to the best neighbour of the current solution and, to avoid cycling, solutions that were recently examined are forbidden or tabu. Elements of Tabu Search: 1) *Tabu List* (short term memory): to record solutions to prevent revisiting a visited solution; 2) *Tabu tenure*: number of iterations a tabu move is considered to remain tabu; 3) *Aspiration criteria*: accepting an improved solution even if generated by a tabu move 4) *Long term memory*: to record attributes of elite solutions to be used in: a) Intensification(giving priority to attributes of a set of elite solutions) b) Diversification(Discouraging attributes of elite solutions in selection functions in order to diversify the search to other areas of solution space)

Algorithm 4 Pseudocode for Tabu Search

```
S ← random valuation of variables;
iter ← 0;
initialize randomly the tabu_list
while (eval(S)> 0) and (iter < Maxiter) do
    choose a move < V, v' > with the best performance among the non-tabu moves and the moves
    satisfying the aspiration criteria;
    introduce < V, v > in the tabu_list, where v is the current value of V
    remove the oldest move from the tabu_list
    assign v' to V;
    iter ← iter + 1;
end while
return S
```

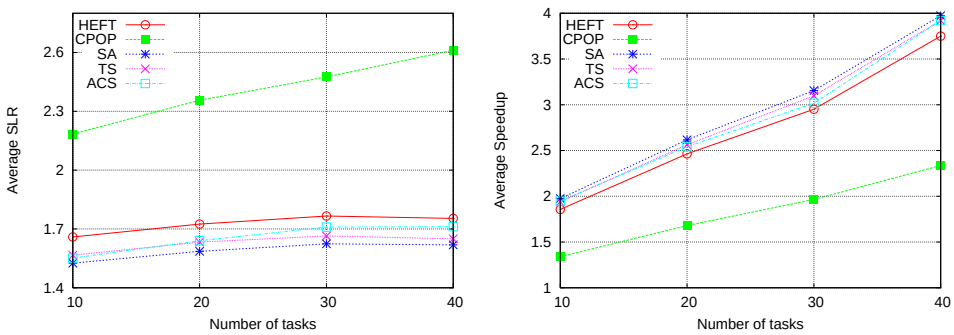
2.5 Result and conclusions

In this section, we evaluate and compare the solution performance of the HEFT and CPOP with metaheuristic algorithms for single DAG scheduling using an extensive simulation setup. The metrics used for comparison are the **SLR (schedule length ratio)** and the **Speedup** (used in [THW02]). In fact, the *SLR* metric make a normalization on the schedule length to a lower bound.

$$SLR = \frac{makespane(solution)}{\sum_{n_i \in CP_{MIN}} \min_{p_j \in P} (w_{(i,j)})} \qquad Speedup = \frac{\min_{p_j \in P} \left[\sum_{n_i \in V} w_{(i,j)} \right]}{makespane(solution)}$$

The denominator in *SLR* is the minimum computation of tasks on critical path. With any algorithm, there is no makespane less than the denominator of *SLR* equation. Therefore, the algorithm with lower *SLR* is the best algorithm. Average *SLR* values over several task graphs are used in our results. In *Speedup*, the sequential time is obtained by the sum of the processing time on the

processor that minimizes the total computation cost [THW02]. The DAGs used in this simulation setup were randomly generated using the program[1] which consider the following parameters: *width* as the number of tasks on the largest level; *regularity* is the uniformity of the number of tasks in each level; *density* is the number of edges between two levels of the DAG. These parameters may vary between 0 and 1. An additional parameter, *jump*, indicates that an edge can go from level l to level $l + jump$. In this paper, we consider DAGs with 10, 20, 30 and 40 tasks; the number of processors equal to 4, 8, 16, and 32; CCR of 0.1, 0.5, 0.8 and 1; width equal to 0.1, 0.2, 0.8; density equal to 0.2, 0.8; and jumps of 1, 2, and 4. These combinations give 1152 different DAG types. Since 5 random DAGs were generate for each combination, the total number of DAGs used in our experiment was 5760. We do not considers CCR above 1 because for a high speed network it would not be a realistic value.



(a) SLR (b) Speedup

Figure 2: The SLR and Speedup average values for each size graph and CCR=[0.1 0.5 0.8 1.0]

	N=10			N=20			N=30			N=40		
CCR	SA	TS	ACS	SA	TS	ACS	SA	TS	ACS	SA	TS	ACS
0.1	0.80%	0.60%	0.53%	1.64%	1.38%	0.62%	3.16%	2.94%	1.35%	4.07%	3.90%	1.79%
0.5	7.03%	5.70%	5.74%	7.09%	5.66%	4.04%	7.97%	6.50%	2.84%	7.93%	6.74%	2.88%
0.8	9.96%	6.91%	8.27%	10.0%	6.80%	6.45%	9.93%	6.49%	4.09%	9.48%	6.61%	1.87%
1.0	10.0%	6.23%	7.74%	10.2%	5.65%	6.14%	9.45%	5.85%	2.98%	10.9%	6.98%	2.51%

Table 3: SLR improvement observed with metaheuristic algorithms compared to HEFT

Figure 2 shows the results of SLR and Speedup for list scheduling algorithms (HEFT and CPOP) and metaheuristic algorithms (ACS, TS and SA). It can be observed that in average there is a consistent gap between the two types of algorithms, being the best solutions obtained by the Simulated Annealing metaheuristic. Also HEFT has always better performance than CPOP, as shown in [THW02]. Considering the results shown on table 3, it can be concluded that for low CCR (0.1) the HEFT gives near results comparing to metaheuristic approaches. This means that the effort of using a higher time complexity approach may not be worth. For higher CCRs up to

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

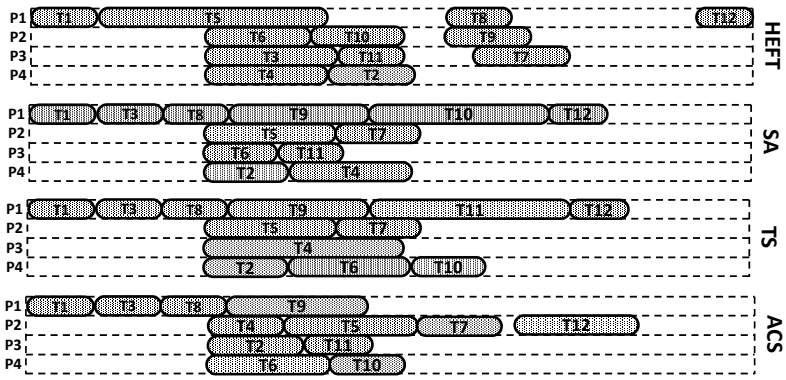


Figure 3: Scheduling of task graph with HEFT, SA, TS, ACS

1.0 the improvement is always below 11%, which means that the improvement is not very high in order to compensate the usage of metaheuristic algorithms. For illustrative purpose, in figure 3 we can see an example of the makespan obtained by HEFT, AS, TS and ACS algorithms, for the DAG represented in Figure 1.

In conclusion, we can say that for low CCR (0.1) HEFT produces schedules competitive with metaheuristic approaches, with a lower time complexity. For higher CCRs up to 1, the improvement achieved with SA is below 11%, being also competitive the schedules produced by HEFT. These results show also that new heuristic base algorithms have a narrow space of improvement over HEFT. Regarding the metaheuristic algorithms, SA showed to achieve consistently better scheduling solutions for DAG scheduling in heterogeneous systems.

Chapter 3

List Scheduling Algorithm for Heterogeneous Systems by an Optimistic Cost Table

Hamid Arabnejad and Jorge G. Barbosa

IEEE Transactions on Parallel and Distributed Systems (TPDS),

volume 25, number 3, pages 682-694, March 2014,

DOI: 10.1109/TPDS.2013.57,

abstract

Efficient application scheduling algorithms are important for obtaining high performance in heterogeneous computing systems. In this paper, we present a novel list-based scheduling algorithm called Predict Earliest Finish Time (PEFT) for heterogeneous computing systems. The algorithm has the same time complexity as the state-of-the-art algorithm for the same purpose, that is, $O(v^2.p)$ for v tasks and p processors, but offers significant makespan improvements by introducing a look-ahead feature without increasing the time complexity associated with computation of an Optimistic Cost Table (OCT). The calculated value is an optimistic cost because processor availability is not considered in the computation. Our algorithm is only based on an OCT table that is used to rank tasks and for processor selection. The analysis and experiments based on randomly generated graphs with various characteristics and graphs of real-world applications show that the PEFT algorithm outperforms the state-of-the-art list-based algorithms for heterogeneous systems in terms of schedule length ratio, efficiency and frequency of best results.

3.1 Introduction

A heterogeneous system can be defined as a range of different system resources, which can be local or geographically distributed, that are utilized to execute computationally intensive applications. The efficiency of executing parallel applications on heterogeneous systems critically depends on the methods used to schedule the tasks of a parallel application. The objective is to minimize the overall completion time or *makespan*. The task scheduling problem for heterogeneous systems is more complex than that for homogeneous computing systems because of the different execution rates among processors and possibly different communication rates among different processors. A popular representation of an application is the Directed Acyclic Graph (DAG), which includes the characteristics of an application program such as the execution time of tasks, the data size to communicate between tasks and task dependencies. The DAG scheduling problem has been shown to be NP-complete [CB76, GJ79, Ull75], even for the homogeneous case; therefore, research efforts in this field have been mainly focused on obtaining low-complexity heuristics that produce good schedules. In the literature, one of the best list-based heuristics is the Heterogeneous Earliest-Finish-Time (HEFT) [THW02]. In [CJSZ08], the authors compared 20 scheduling heuristics and concluded that on average, for random graphs, HEFT is the best one in terms of robustness and schedule length.

The task scheduling problem is broadly classified into two major categories, namely Static Scheduling and Dynamic Scheduling. In the Static category, all information about tasks such as execution and communication costs for each task and the relationship with other tasks is known beforehand; in the dynamic category, such information is not available and decisions are made at run-time. Moreover, Static scheduling is an example of compile-time scheduling, whereas Dynamic scheduling is representative of run-time scheduling. Static scheduling algorithms are universally classified into two major groups, namely Heuristic-based and Guided Random Search-based algorithms. Heuristic-based algorithms allow approximate solutions, often good solutions, with polynomial time complexity. Guided Random Search-based algorithms also give approximate solutions, but the solution quality can be improved by including more iterations, which therefore makes them more expensive than the Heuristic-based approach [THW02]. The Heuristic-based group is composed of three subcategories: list, clustering and duplication scheduling. Clustering heuristics are mainly proposed for homogeneous systems to form clusters of tasks that are then assigned to processors. For heterogeneous systems, CHP algorithms [BR$^+$04] and Triplet [CJ01] were proposed, but they have limitations in higher-heterogeneity systems. The duplication heuristics produce the shortest *makespans*, but they have two disadvantages: a higher time complexity, i.e., cubic, in relation to the number of tasks, and the duplication of the execution of tasks, which results in more processor power used. This is an important characteristic not only because of the associated energy cost but also because, in a shared resource, fewer processors are available to run other concurrent applications. List scheduling heuristics, on the other hand, produce the most efficient schedules, without compromising the *makespan* and with a complexity that is generally quadratic in relation to the number of tasks. HEFT has a complexity of $O(v^2.p)$, where v is the

number of tasks and p is the number of processors.

In this paper, we present a new list scheduling algorithm for a bounded number of fully connected heterogeneous processors called Predict Earliest Finish Time (PEFT) that outperforms state-of-the-art algorithms such as HEFT in terms of *makespan* and Efficiency. The time complexity is $O(v^2.p)$, as in HEFT. To our knowledge, this is the first algorithm to outperform HEFT while having the same time complexity. We also introduce one innovation, a look ahead feature, without increasing the time complexity. Other algorithms such as LDCP [DK08] and Lookahead [BSM10] have this feature but with a cubic and quartic time complexity, respectively. We present results for randomly generated DAGs[1] and DAGs from well-known applications used in related papers, such as Gaussian Elimination, Fast Fourier Transform, Montage and Epigenomic Workflows.

This paper is organized as follows: in Section 2, we introduce the task scheduling problem; in Section 3, we present related work in scheduling DAGs on heterogeneous systems, and we present in detail the scheduling algorithms that are used for the comparison with PEFT, which is the list scheduling heuristic proposed here; in Section 4, we present PEFT; in Section 5 we present the results and, finally, we present the conclusions in Section 6.

3.2 Scheduling problem formulation

The problem addressed in this paper is the static scheduling of a single application in a heterogeneous system with a set P of processors. As mentioned above, task scheduling can be divided into Static and Dynamic approaches. Dynamic scheduling is adequate for situations where the system and task parameters are not known at compile time, which requires decisions to be made at runtime but with additional overhead. A sample environment is a system where users submit jobs, at any time, to a shared computing resource [MAS+99]. A dynamic algorithm is required because the workload is only known at runtime, as is the status of each processor when new tasks arrive. Consequently, a dynamic algorithm does not have all work requirements available during scheduling and cannot optimize based on the entire workload. By contrast, a static approach can maximize a schedule by considering all tasks independently of execution order or time because the schedule is generated before execution begins and introduces no overhead at runtime. In this paper, we present an algorithm that minimizes the execution time of a single job on a set of P processors. We consider that P processors are available for the job and that they are not shared during the job execution. Therefore, with the system and job parameters known at compile time, a static scheduling approach has no overhead at runtime and is more appropriate.

An application can be represented by a *Directed Acyclic Graph* (DAG), $G = (V, E)$, as shown in Figure 1, where V is the set of v nodes and each node $v_i \in V$ represents an application task, which includes instructions that must be executed on the same machine. E is the set of e communication edges between tasks; each $e(i, j) \in E$ represents the task-dependency constraint such that task n_i should complete its execution before task n_j can be started. The DAG is complemented by a matrix

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

W that is a $v \times p$ computation cost matrix, where v is the number of tasks and p is the number of processors in the system. $w_{i,j}$ gives the estimated time to execute task v_i on machine p_j. The mean execution time of task v_i is calculated by equation 1.

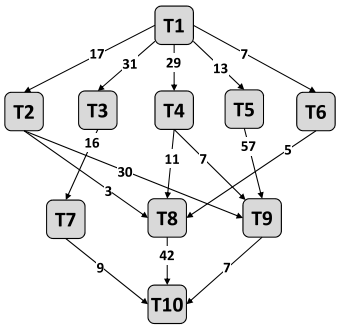
$$\overline{w_i} = (\sum_{j \in P} w_{i,j})/p \tag{1}$$

The average execution time $\overline{w_i}$ is commonly used to compute a priority rank for the tasks. The algorithm proposed in this paper uses the $w_{i,j}$ rather than $\overline{w_i}$ as explained in section 3.4.

Each edge $e(i,j) \in E$ is associated with a non-negative weight $c_{i,j}$ representing the communication cost between the tasks v_i and v_j. Because this value can be computed only after defining where tasks v_i and v_j will be executed, it is common to compute the average communication costs to label the edges [THW02]. The average communication cost $\overline{c_{i,j}}$ of an edge $e(i,j)$ is calculated by equation 2.

$$\overline{c_{i,j}} = \overline{L} + \frac{data_{i,j}}{\overline{B}} \tag{2}$$

where $\overline{L}$ is the average latency time of all processors and $\overline{B}$ is the average bandwidth of all links connecting the set of P processors. $data_{i,j}$ is the amount of data elements that task v_i needs to send to task v_j. Note that when tasks v_i and v_j are assigned to the same processor, the real communication cost is considered to be zero because it is negligible compared with interprocessor communication costs.



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

Figure 1: Application DAG and computation time matrix of the tasks in each processor for a three processor machine

Additionally, in our model, we consider processors that are connected in a fully connected topology. The execution of tasks and communications with other processors can be achieved for each processor simultaneously and without contention. Additionally, the execution of any task is considered nonpreemptive. These model simplifications are common in this scheduling problem [DK08, HJ05, THW02], and we consider them to permit a fair comparison with state-of-the-art algorithms and because these simplifications correspond to real systems. Our target system is a single site infrastructure that can be as simple as a set of devices (e.g., CPUs and GPUs) connected by a switched network that guarantees parallel communications between different pairs of devices.

The machine is heterogeneous because CPUs can be from different generations and also because other very different devices such as GPUs can be included. Another common machine is the one that results from selecting processors from several clusters from the same site. Although a cluster is homogeneous, the set of processors selected to execute a given DAG forms a heterogeneous machine. Because the clusters are connected by high-speed networks, with redundant links, the simplification is still reasonable. The processor latency can differ in a heterogeneous machine, but such differences are negligible. For low communication-to-computation ratios (CCRs), the communications are negligible; for higher CCRs, the predominant factor is the network bandwidth, and we consider that the bandwidth is the same throughout the entire network.

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

Definition (1) $pred(n_i)$: 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 can be added to the graph.

Definition (2) $succ(n_i)$: denotes the set of immediate successors of task n_i. A task with no successors is called an *exit* task, n_{exit}. Similar to 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 can be added to the graph.

Definition (3) *makespan* or *schedule length*: denotes the finish time of the last task in the scheduled DAG and is defined by

$$makespan = max\{AFT(n_{exit})\} \tag{3}$$

where $AFT(n_{exit})$ denotes the *Actual Finish Time* of the exit node. In the case where there is more than one exit node and no redundant node is added, the *makespan* is the maximum actual finish time of all exit tasks.

Definition (4) $level(n_i)$: the level of task n_i is an integer value representing the maximum number of edges of 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

$$level(n_i) = \max_{q \in pred(n_i)} \{level(q)\} + 1 \tag{4}$$

Definition (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 lower bound of a schedule length is the minimum critical path length (CP_{MIN}), which is computed by considering the minimum computational costs of each node in the critical path.

Definition (6) $EST(n_i, p_j)$: denotes the *Earliest Start Time* of a node n_i on a processor p_j and is defined as

$$EST(n_i, p_j) = \max\left\{T_{Available}(p_j), \max_{n_m \in pred(n_i)} \left\{AFT(n_m) + c_{m,i}\right\}\right\} \tag{5}$$

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 arrive at the processor p_j. The communication cost $c_{m,i}$ is zero if the predecessor node n_m is assigned to processor p_j. For the entry task, $EST\left(n_{entry}, p_j\right) = 0$.

Definition (7) $EFT(n_i, p_j)$: denotes the *Earliest Finish Time* of a node n_i on a processor p_j and is defined as

$$EFT(n_i, p_j) = EST(n_i, p_j) + w_{i,j} \tag{6}$$

which is the *Earliest Start Time* of a node n_i on a processor p_j plus the computational cost of n_i on a processor p_j.

The *objective* of the scheduling problem is to determine an assignment of tasks of a given DAG to processors such that the *Schedule Length* is minimized. After all nodes in the DAG are scheduled, the schedule length will be the *Actual Finish Time* of the exit task, as expressed by equation 1.

3.3 Related work

In this section, we present a brief survey of task scheduling algorithms, specifically list-based heuristics. We present their time complexity and their comparative performance.

Over the past few years, research on static DAG scheduling has focused on finding suboptimal solutions to obtain a good solution in an acceptably short time. List scheduling heuristics usually generate high-quality schedules at a reasonable cost. In comparison with clustering algorithms, they have lower time complexity, and in comparison with task duplication strategies, their solutions represent a lower processor workload.

3.3.1 List-based algorithms

Many list scheduling algorithms have been developed by researchers. This type of scheduling algorithm has two phases: the *prioritizing* phase for giving a priority to each task and a *processor selection* phase for selecting a suitable processor that minimizes the heuristic cost function. If two or more tasks have equal priority, then the tie is resolved by selecting a task randomly. The last phase is repeated until all tasks are scheduled to suitable processors. Table 4 presents some list scheduling algorithms including some of the most cited, along with their time complexity.

The MH and DLS algorithms consider a link contentions model, but their versions for a fully connected network were used for comparison purposes by other authors and are therefore described here. The schedules generated by the Mapping Heuristic (MH) algorithm [ERL90] are generally longer than recently developed heuristics because MH only considers a processor ready when it finishes the last task assigned to it. Therefore, MH ignores the possibility that a processor that is busy when a task is being scheduled can complete the task in a shorter time, which thus results in poorer scheduling. The time complexity of MH without contention is $O(v^2.p)$ and $O(v^2.p^3)$ otherwise. Dynamic Level Scheduling (DLS) [SL+93] is one of the first algorithms that

Algorithm	Reference	Complexity
MH	[ERL90]	$O(v^2.p)$
DLS	[SL+93]	$O(v^3.p)$
LMT	[IÖF95]	$O(v^2.p^2)$
BIL	[OH96]	$O(v^2.p.\log p)$
FLB	[RVG00]	$O(v.(\log p + \log v) + v^2)$
CPOP	[THW99, THW02]	$O(v^2.p)$
HEFT	[THW99, THW02]	$O(v^2.p)$
HCPT	[HJ03]	$O(v^2.p)$
HPS	[ITM05]	$O(v^2(p.\log v))$
PETS	[IT07]	$O(v^2(p.\log v))$
LDCP	[DK08]	$O(v^3.p)$
Lookahead	[BSM10]	$O(v^4.p^3)$

Table 4: List-based scheduling algorithms for heterogeneous systems

computed an estimate of the availability of each processor and thus allowed a task to be scheduled to a currently busy processor. Consequently, DLS yields better results than MH. The DLS authors proposed a version for a heterogeneous machine, but the processor selection was based on the *Earliest Start Time* (EST) as the homogeneous version, which is one of the drawbacks of the algorithm because the EST does not guarantee the minimum completion time for a task. Additionally, DLS does not try to fill scheduling holes in a processor schedule (idle time slots that are between two tasks already scheduled on the same processor), in contrast to other more recent algorithms. The time complexity of DLS for a fully connected network is $O(v^3.p)$, where the routing complexity is equal to 1. The Levelized min-Time Algorithm (LMT) [IÖF95] is a very simple heuristic that assigns priorities to tasks based on their precedence constraints, which are called levels. The time complexity is squared in relation to the number of processors and tasks. The schedules produced are significantly worse than those produced by other more recently developed heuristics, such as CPOP [THW99, THW02]. The Best Imaginary Level (BIL) [OH96] defines a static level for DAG nodes, called BIL, that incorporates the effect of interprocessor communication overhead and processor heterogeneity. The BIL heuristic provides an optimal schedule for linear DAGs. Fast Load Balancing (FLB) [RVG00] was proposed with the aim of reducing the time complexity relative to that of HEFT [THW99, THW02]. FLB generates schedules comparable to those of HEFT, but it generates poor schedules for irregular task graphs and for higher processor heterogeneities. The Critical Path On a Processor (CPOP) [THW99, THW02] achieves better schedules than LMT and MH as well as schedules that are comparable to those of DLS, with a lower time complexity. The main feature of CPOP is the assignment of all the tasks that belong to the critical path to a single processor. Heterogenous Critical Parent Trees (HCPT) [HJ03] yield better scheduling results than CPOP, FLB and DLS. The Heterogeneous Earliest-Finish-Time (HEFT) [THW99, THW02]

is one of the best list scheduling heuristics, as it has quadratic time complexity. In [CJSZ08], the authors compared 20 heuristics and concluded that HEFT produces the shortest schedule lengths for random graphs.

HPS[ITM05] and PETS[IT07] are other list scheduling heuristics reported to achieve better results than HEFT. HPS, PETS, HEFT, HCPT and Lookahead are explained in more detail in section 3.3.2, and they are used in this paper for comparison with our proposed list scheduling algorithm.

For the Longest Dynamic Critical Path (LDCP) [DK08], the authors reported better scheduling results than HEFT, although these results were not significant when considering the associated increase in complexity. For random graphs, the improvements in the schedule length ratio over HEFT were less than 3.1%. The algorithm builds for each processor a DAG, called a DAGP, that consists of the initial DAG with the computation costs of the processors. The complexity is higher (cubic) because the algorithm needs to update all DAGPs after scheduling a task to a processor.

Another recent algorithm that reported an average improvement in *makespan* over HEFT is the Lookahead approach [BSM10]. This algorithm has quartic complexity for the one step Lookahead. We consider the Lookahead for comparison with our algorithm because it has achieved the best results reported thus far in the literature. Because it has a higher complexity, it serves as a reference and an upper bound for our algorithm.

3.3.2 Selected list scheduling heuristics

Here, we describe the list scheduling heuristics for scheduling tasks on a bounded number of heterogeneous processors selected for comparison with our proposed algorithm, namely HEFT, HCPT, HPS, PETS and Lookahead.

3.3.2.1 Heterogeneous Earliest Finish Time (HEFT)

The HEFT algorithm [THW02] is highly competitive in that it generates a schedule length comparable to the schedule lengths of other scheduling algorithms with a lower time complexity. The HEFT algorithm has two phases: a *task prioritizing* and a *processor selection* phase. In the first phase task, priorities are defined as $rank_u$. $rank_u$ represents the length of the longest path from task n_i to the exit node, including the computational cost of n_i, and is given by $rank_u(n_i) = \overline{w_i} + \max_{n_j \in succ(n_i)}\{\overline{c_{i,j}} + rank_u(n_j)\}$. For the exit task, $rank_u(n_{exit}) = \overline{w_{exit}}$. The task list is ordered by decreasing value of $rank_u$. In the *processor selection phase*, the task on top of the task list is assigned to the processor p_j that allows for the EFT (Earliest Finish Time) of task n_i. However, the HEFT algorithm uses an insertion policy that tries to insert a task in at the earliest idle time between two already scheduled tasks on a processor, if the slot is large enough to accommodate the task.

3.3.2.2 Heterogeneous Critical Parent Trees (HCPT)

The HCPT algorithm [HJ03] uses a new mechanism to construct the scheduling list L, rather than assigning priorities to the application tasks. HCPT divides the task graph into a set of unlisted-parent trees. The root of each unlisted-parent tree is a critical path node (CN). A CN is defined as the node that has zero difference between its *AEST* and *ALST*. The *AEST* is the *Average Earliest Start Time* of node n_i and is equivalent to $rank_d$, as indicated by $AEST(n_i) = \max_{n_j \in pred(n_i)}\{AEST(n_j) + \overline{w_j} + \overline{c_{j,i}}\}; AEST(n_{entry}) = 0$.

The *Average Latest Start Time* (*ALST*) of node n_i can be computed recursively by traversing the DAG upward starting from the exit node and is given by $ALST(n_i) = \min_{n_j \in succ(n_i)}\{ALST(n_j) - \overline{c_{i,j}}\} - \overline{w_i}; ALST(n_{exit}) = AEST(n_{exit})$.

The algorithm also has two phases, namely *listing tasks* and *processor assignment*. In the first phase, the algorithm starts with an empty queue L and an auxiliary stack S that contains the CNs pushed in decreasing order of their ALSTs, i.e., the entry node is on top of S. Consequently, $top(S)$ is examined. If $top(S)$ has an unlisted parent (i.e., has a parent not in L), then this parent is pushed on the stack S. Otherwise, $top(S)$ is removed and enqueued into L. In the processor assignment phase, the algorithm tries to assign each task $n_i \in L$ to a processor p_j that allows the task to be completed as early as possible.

3.3.2.3 High Performance Task Scheduling (HPS)

The HPS [ITM05] algorithm has three phases, namely a *level sorting*, *task prioritization* and *processor selection* phase. In the level sorting phase, the given DAG is traversed in a top-down fashion to sort tasks at each level to group the tasks that are independent of each other. As a result, tasks in the same level can be executed in parallel. In the task prioritization phase, priority is computed and assigned to each task using the attributes *Down Link Cost* (DLC), *Up Link Cost* (ULC) and *Link Cost* (LC) of the task. The DLC of a task is the maximum communication cost among all the immediate predecessors of the task. The DLC for all tasks at level 0 is 0. The ULC of a task is the maximum communication cost among all the immediate successors of the task. The ULC for an exit task is 0. The LC of a task is the sum of DLC, ULC and maximum LC for all its immediate predecessor tasks.

At each level, based on the LC values, the task with the highest LC value receives the highest priority, followed by the task with the next highest LC value and so on in the same level. In the processor selection phase, the processor that gives the minimum *EFT* for a task is selected to execute that task. HPS has an insertion-based policy, which considers the insertion of a task in the earliest idle time slot between two already-scheduled tasks on a processor.

3.3.2.4 Performance Effective Task Scheduling (PETS)

The PETS algorithm [IT07] has the same three phases as HPS. In the level sorting phase, similar to HPS, tasks are categorized in levels such that in each level, the tasks are independent. In the task prioritization phase, priority is computed and assigned to each task using the attributes *Average*

Computation Cost (ACC), *Data Transfer Cost* (DTC) and the *Rank of Predecessor Task* (RPT). The ACC of a task is the average computation cost for all p processors, which we referred to before as $\overline{w_i}$. The DTC of a task n_i is the communication cost incurred when transferring the data from task n_i to all its immediate successor tasks; for an exit node, $DTC(n_{exit}) = 0$. The RPT of a task n_i is the highest rank of all its immediate predecessor tasks; for an entry node, $RPT(n_{entry}) = 0$. The rank is computed for each task n_i based on the task's ACC, DTC and RPT values and is given by $rank(n_i) = round\{ACC(n_i) + DTC(n_i) + RPT(n_i)\}$.

At each level, the task with the highest rank value receives the highest priority, followed by the task with next highest rank value and so on. A tie is broken by selecting the task with the lower ACC value. As in some of the other task scheduling algorithms, in the processor selection phase, this algorithm selects the processor that gives the minimum *EFT* value for executing the task. It also uses an insertion-based policy for scheduling a task in an idle slot between two previously scheduled tasks on a given processor.

3.3.2.5 Lookahead Algorithm

The Lookahead scheduling algorithm [BSM10] is based on the HEFT algorithm, whose main feature is its processor selection policy. To select a processor for the current task t, it iterates over all available processors and computes the EFT for the child tasks on all processors. The processor selected for task t is the one that minimizes the maximum EFT from all children of t on all resources where t is tried. This procedure can be repeated for each child of t by increasing the number of levels analyzed. In HEFT, the complexity of v tasks is $O(e.p)$, where EFT is computed v times. In the worst case, by replacing e by v^2, we obtain $O(v^2.p)$. The Lookahead algorithm has the same structure as HEFT but computes EFT for each child of the current task. The number of EFT calls (graph vertices) is equal to $v + p.e$ for a single level of forecasting. By replacing this number of vertices in $O(v^2.p)$, in the worst case the total time complexity of Lookahead is $O(v^4.p^3)$. The authors reported that additional levels of forecasting do not result in significant improvements in the makespan. Here, we only consider the one-level Lookahead.

3.4 The proposed algorithm PEFT

In this section, we introduce a new list-based scheduling algorithm for a bounded number of heterogeneous processors, called PEFT. The algorithm has two major phases: a *task prioritizing* phase for computing task priorities, and a *processor selection* phase for selecting the best processor for executing the current task.

In our previous work [AB12b], we evaluated the performance of list-based scheduling algorithms. We compared their results with the solutions achieved by three meta-heuristic algorithms, namely Tabu Search, Simulated Annealing and Ant Colony System. The meta-heuristic algorithms, which feature a higher processing time, always achieved better solutions than the list scheduling heuristics with quadratic complexity. We then compared the best solutions for both types, step by step. We observed that the best meta-heuristic schedules could not be achieved if

we followed the common strategy of selecting processors based only on current task execution time, because the best schedules consider not only the immediate gain in processing time but also the gain in a sequence of tasks. Most list-based scheduling heuristics with quadratic time complexity assign a task to a processor by evaluating only the current task. This methodology, although inexpensive, does not evaluate what is ahead of the current task, which leads to poor decisions in some cases. Algorithms that analyze the impact on children nodes, such as Lookahead [BSM10], exist, but they increase the time complexity to the 4th order.

The most powerful feature of the Lookahead algorithm, as the best algorithm with the lowest makespan, is its ability to forecast the impact of an assignment for all children of the current task. This feature permits better decisions to made in selecting processors, but it increases the complexity significantly. Therefore, the novelty of the proposed algorithm is its ability to forecast by computing an *Optimistic Cost Table* while maintaining quadratic time complexity, as explained in the following section.

3.4.1 Optimistic cost table (OCT)

Our algorithm is based on the computation of a cost table on which task priority and processor selection are based. The OCT is a matrix in which the rows indicate the number of tasks and the columns indicate the number of processors, where each element $OCT(t_i, p_k)$ indicates the maximum of the shortest paths of t_i children's tasks to the exit node considering that processor p_k is selected for task t_i. The OCT value of task t_i on processor p_k is recursively defined by Equation 7 by traversing the DAG from the exit task to the entry task.

$$OCT(t_i, p_k) = \max_{t_j \in succ(t_i)} \left[\min_{p_w \in P} \left\{ OCT(t_j, p_w) + w(t_j, p_w) + \overline{c_{i,j}} \right\} \right], \quad \overline{c_{i,j}} = 0 \; if \; p_w = p_k. \tag{7}$$

where $\overline{c_{i,j}}$ is the average communication cost, which is zero if t_j is being evaluated for processor p_k, and $w(t_j, p_w)$ is the execution time of task t_j on processor p_w. As explained before, we use the average communication cost and the execution cost for each processor. $OCT(t_i, p_k)$ represents the maximum optimistic processing time of the children of task t_i because it considers that children tasks are executed in the processor that minimizes processing time (communications and execution) independently of processor availability, as the OCT is computed before scheduling begins. Because it is defined recursively and the children already have the optimistic cost to the exit node, only the first level of children is considered. For the exit task, the $OCT(n_{exit}, p_k) = 0$ for all processors $p_k \in P$.

3.4.2 Task prioritizing phase

To define task priority, we compute the average OCT for each task that is defined by equation 8.

$$rank_{oct}(t_i) = \frac{\sum_{k=1}^{P} OCT(t_i, p_k)}{P} \tag{8}$$

Table 5 shows the values of OCT for the DAG sample of Figure 1. When the new rank $rank_{oct}$ is compared with $rank_u$, the former shows slight differences in the order of the tasks based on these two priority strategies. For instance, $T5$ has a lower $rank_{oct}$ value than $T4$, where $T4$ is selected first for scheduling. With $rank_u$, the opposite is true. The main feature of our algorithm is the cost table that reflects for each task and processor the cost to execute descendant tasks until the exit node. This information permits an informed decision to be made in assigning a processor to a task. Task ranking is a less relevant issue because few tasks in each step are ordered by priority and the influence on performance is less relevant. By comparing $rank_u$ and $rank_{oct}$, we can see that $rank_u$ uses the average computing cost for each task and also accumulates the maximum descendant costs of descendant tasks to the exit node. In contrast, $rank_{oct}$ is an average over a set of values that were computed with the cost of each task on each processor. Therefore, the ranks are computed using a similar procedure, and significant differences in performance are not expected when using either system.

For the tests with random graphs reported in the results section, when using $rank_u$ the performance is on average better in approximately 0.5%. The OCT exerts a greater influence in the processor selection phase, and using $rank_u$ would require additional computations without providing a significant advantage.

Task	$P1$	$P2$	$P3$	$rank_{oct}$	$rank_u$
T1	64	68	86	72.7	169
T2	42	39	42	41	114.3
T3	27	41	43	37	102.7
T4	42	39	50	43.7	110
T5	28	37	28	31	129.7
T6	42	39	44	41.7	119.3
T7	13	16	22	17	52.7
T8	13	16	33	20.7	92
T9	13	16	20	16.3	42.3
T10	0	0	0	0	20.7

Table 5: Optimistic Cost Table for the DAG of Figure 1; $rank_{oct}$ and $rank_u$ are also shown for comparison purposes

3.4.3 Processor selection phase

To select a processor for a task, we compute the Optimistic EFT (O_{EFT}), which sums to EFT the computation time of the longest path to the exit node. In this way, we are looking forward (forecasting) in the processor selection; perhaps we are not selecting the processor that achieves the earliest finish time for the current task, but we expect to achieve a shorter finish time for the tasks in the next steps. The aim is to guarantee that the tasks ahead will finish earlier, which is the purpose of the OCT table. O_{EFT} is defined by equation 9.

$$O_{EFT}(t_i, p_j) = EFT(t_i, p_j) + OCT(t_i, p_j) \tag{9}$$

3.4.4 Detailed description of PEFT algorithm

In this section, we present the algorithm PEFT, supported by an example, to detail the description of each step. The proposed PEFT algorithm is formalized in Algorithm 1.

Algorithm 1 The PEFT algorithm

1: Compute OCT table and $rank_{oct}$ for all tasks
2: Create Empty list *ready-list* and put n_{entry} as initial task
3: **while** *ready-list* is NOT Empty **do**
4: $n_i \leftarrow$ the task with highest $rank_{oct}$ from *ready-list*
5: **for all** processor p_j in the processor-set P **do**
6: Compute $EFT(n_i, p_j)$ value using insertion-based scheduling policy
7: $O_{EFT}(n_i, p_j) = EFT(n_i, p_j) + OCT(n_i, p_j)$
8: **end for**
9: Assign task n_i to the processor p_j that minimize O_{EFT} of task n_i
10: Update *ready-list*
11: **end while**

The algorithm starts by computing the OCT table and $rank_{oct}$ at line 1. It then creates an empty $ready-list$ and places the entry task on top of the list. In the while loop, from line 4 to 10, in each iteration, the algorithm will schedule the task with a higher value of $rank_{oct}$. After selecting the task for scheduling, the PEFT algorithm calculates the O_{EFT} values for the task on all processors. In the processor selection phase, the aim is to guarantee that the tasks ahead will finish earlier, but rather than analyzing all tasks until the end, to reduce complexity we use the OCT table, which incorporates that information. In line 9, the processor p_j that achieves the minimum $O_{EFT}(n_i, p_j)$ is selected to execute task n_i.

Table 6 shows an example that demonstrates the PEFT for the DAG of Figure 1.

Figure 2 shows the scheduling results for the sample DAG with the algorithms PEFT, Lookahead, HEFT, HCPT, HPS and PETS. By comparing the schedules of PEFT and HEFT, we can see that T1 is assigned to P1 although it does not guarantee the earliest finish time for T1, but P1 minimizes the expected EFT of all DAGs. This is only one example used to illustrate the algorithm, but as shown in the results section, PEFT produces, on average, better schedules than the state-of-the-art algorithms.

Step	Ready List	Task selected	EFT			O_{EFT}			CPU Selected
			P1	P2	P3	P1	P2	P3	
1	T1	T1	22	**21**	36	**86**	89	122	P1
2	T4,T6,T2,T3,T5	T4	**29**	61	55	**71**	100	105	P1
3	T6,T2,T3,T5	T6	55	**46**	53	97	**85**	97	P2
4	T2,T3,T5	T2	**51**	64	57	**93**	103	99	P1
5	T3,T5,T8	T3	83	**80**	96	**110**	121	139	P1
6	T5,T8,T7	T5	112	73	**70**	140	110	**98**	P3
7	T8,T7,T9	T8	112	**77**	106	125	**93**	139	P2
8	T7,T9	T7	**97**	124	129	**110**	140	151	P1
9	T9	T9	142	148	**89**	155	164	**109**	P3
10	T10	T10	132	**122**	152	132	**122**	152	P2

Table 6: Schedule produced by the PEFT algorithm in each iteration

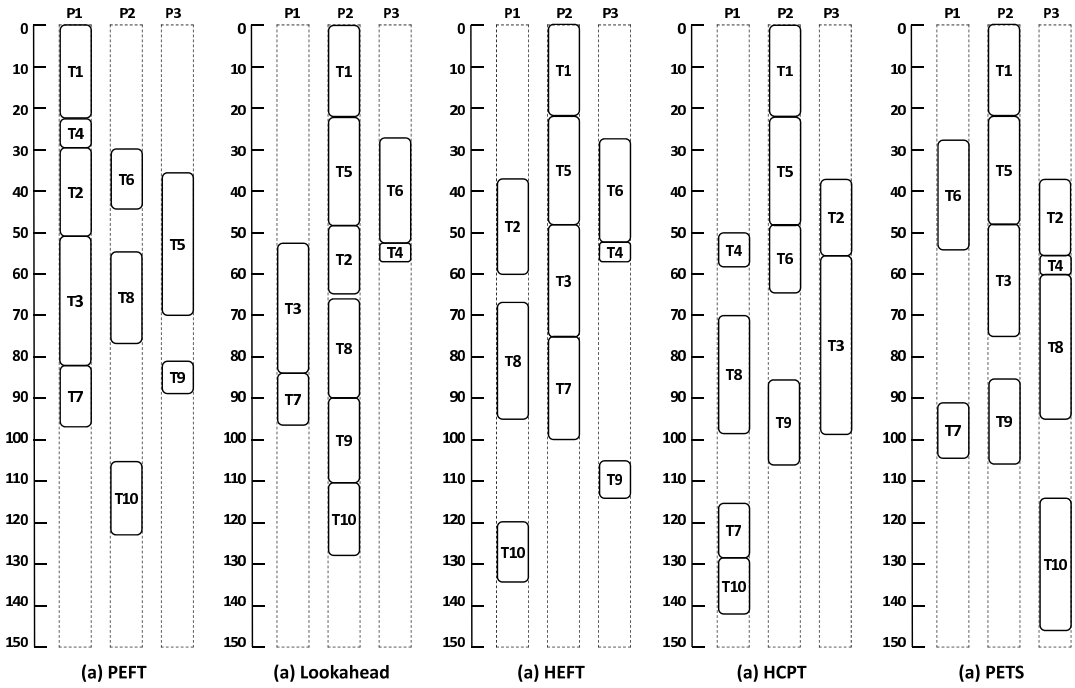


Figure 2: Schedules of the sample task graph in Figure 1 with **(a)** PEFT (*makespan=122*), **(b)** Lookahead (*makespan=127*), **(c)** HEFT (*makespan=133*), **(d)** HCPT (*makespan=142*), **(e)** PETS (*makespan=147*)

In terms of time complexity, PEFT requires the computation of an OCT table that is $O(p(e+v))$, and to assign the tasks to processors the time complexity is of the order $O(v^2.p)$. The total

time is $O(p(e+v)+v^2.p)$. For dense DAGs, e becomes v^2, where the total algorithm complexity is of the order $O(v^2.p)$. That is, the time complexity of PEFT is of the same order as the HEFT algorithm.

3.5 Experimental Results and Discussion

This section compares the performance of the PEFT algorithm with that of the algorithms presented above. For this purpose, we consider two sets of graphs as the workload: randomly generated application graphs and graphs that represent some real-world applications. We first present the comparison metrics used for the performance evaluation.

3.5.1 Comparison Metrics

The comparison metrics are Scheduling Length Ratio, Efficiency, pair-wise comparison of the number of occurrences of better solutions and Slack.

Scheduling Length Ratio (SLR)

The metric most commonly used to evaluate a schedule for a single DAG is the *makespan*, as defined by equation 1. Here, we want to use a metric that compares DAGs with very different topologies; the metric most commonly used to do so is the Normalized Schedule Length (NSL) [DK08], which is also called the Scheduling Length Ratio (SLR) [THW02]. For a given DAG, both represent the *makespan* normalized to the lower bound. SLR is defined by equation 10.

$$SLR = \frac{makespan(solution)}{\sum_{n_i \in CP_{MIN}} \min_{p_j \in P} (w_{(i,j)})} \tag{10}$$

The denominator in *SLR* is the minimum computation cost of the critical path tasks (CP_{MIN}). There is no *makespan* less than the denominator of the *SLR* equation. Therefore, the algorithm with the lowest *SLR* is the best algorithm.

Efficiency

In the general case, Efficiency is defined as the Speedup divided by the number of processors used in each run, and Speedup is defined as the ratio of the sequential execution time to the parallel execution time (i.e., the *makespan*). The sequential execution time is computed by assigning all tasks to a single processor that minimizes the total computation cost of the task graph, as shown by equation 11.

$$Speedup = \frac{\min_{p_j \in P} \left[\sum_{n_i \in V} w_{(i,j)} \right]}{makespan(solution)} \tag{11}$$

Number of occurrences of better solutions

This comparison is presented as a pair-wise table, where the percentage of better, equal and worse solutions produced by PEFT is compared to that of the other algorithms.

Slack

The *slack* metric [BM02, SJD06] is a measure of the robustness of the schedules produced by an algorithm to uncertainty in the tasks' processing time, and it represents the capacity of the schedule to absorb delays in task execution. The *slack* is defined for a given schedule and a given set of processors. The *slack* of a task represents the time window within which the task can be delayed without extending the makespan. *Slack* and makespan are two conflicting metrics; lower makespans produce small *slack*. For deterministic schedules, the *slack* is defined by Equation12.

$$Slack = \left[\sum_{t_i \in V} M - b_{level}(t_i) - t_{level}(t_i) \right] / n \tag{12}$$

where M is the makespan of the DAG, n is the number of tasks, b_{level} is the length of the longest path to the exit node and t_{level} is the length of the longest path from the entry node. These values are referred to a given schedule, and therefore the processing time used for each task is the processing time on the processor that it was assigned. The aim of using this metric is to evaluate whether the proposed algorithm has an equivalent *slack* to HEFT, which is the reference quadratic algorithm.

3.5.2 Random Graph Generator

To evaluate the relative performance of the heuristics, we first considered randomly generated application graphs. For this purpose, we used a synthetic DAG generation program[2] with an adaptation to the fat parameter, as explained next. Five parameters define the DAG shape:

- ***n***: number of computation nodes in the DAG (i.e., application tasks);
- ***fat***: this parameter affects the height and the width of the DAG. The width in each level is defined by a uniform distribution with a mean equal to $fat.\sqrt{n}$. The height, or the number of levels, is created until n tasks are defined in the DAG. The width of the DAG is the maximum number of tasks that can be executed concurrently. A small value will lead to a thin DAG (e.g., chain) with low task parallelism, whereas a large value induces a fat DAG (e.g., fork-join) with a high degree of parallelism;
- ***density***: determines the number of edges between two levels of the DAG, with a low value leading to few edges and a large value leading to many edges;
- ***regularity***: the regularity determines the uniformity of the number of tasks in each level. A low value indicates that levels contain dissimilar numbers of tasks, whereas a high value indicates that all levels contain similar numbers of tasks;

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

- ***jump***: indicates that an edge can go from level l to level $l + jump$. A jump of 1 is an ordinary connection between two consecutive levels.

In the present study, we used this synthetic DAG generator to create the DAG structure, which includes the specific number of nodes and their dependencies. To obtain computation and communication costs, the following parameters are used:

- ***CCR*** (Communication to Computation Ratio): ratio of the sum of the edge weights to the sum of the node weights in a DAG;
- β (Range percentage of computation costs on processors): the *heterogeneity factor* for processor speeds. A high β value implies higher heterogeneity and different computation costs among processors, and a low value implies that the computation costs for a given task are nearly equal among processors [THW02]. The average computation cost of a task n_i in a given graph $\overline{w_i}$ is selected randomly from a uniform distribution with range $\left[0, 2 \times \overline{w_{DAG}}\right]$, where $\overline{w_{DAG}}$ is the average computation cost of a given graph that is obtained randomly. The computation cost of each task n_i on each processor p_j is randomly set from the range of equation 3.

$$\overline{w_i} \times \left(1 - \frac{\beta}{2}\right) \leq w_{i,j} \leq \overline{w_i} \times \left(1 + \frac{\beta}{2}\right) \tag{13}$$

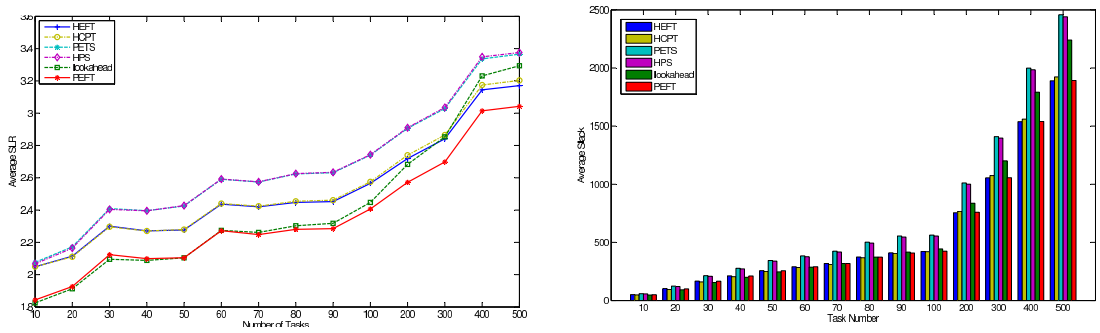
In our experiment, for random DAG generation, we considered the following parameters:

- $n = \left[10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300, 400, 500\right]$
- $CCR = \left[0.1, 0.5, 0.8, 1, 2, 5, 10\right]$
- $\beta = \left[0.1, 0.2, 0.5, 1, 2\right]$
- $jump = \left[1, 2, 4\right]$
- $regularity = \left[0.2, 0.8\right]$
- $fat = \left[0.1, 0.4, 0.8\right]$
- $density = \left[0.2, 0.8\right]$
- $Processors = \left[4, 8, 16, 32\right]$

These combinations produce 70,560 different DAGs. For each DAG, 10 different random graphs were generated with the same structure but with different edge and node weights. Thus, 705,600 random DAGs were used in the study.

Figure 3a shows the average SLR and Figure 3b shows the average *slack* for all algorithms as a function of the DAG size. For DAGs featuring up to 100 tasks, Lookahead and PEFT present similar results. For larger DAGs, PEFT is the best algorithm, as it outperforms the Lookahead

algorithm. All quadratic algorithms maintain a certain level of performance, in contrast to the Lookahead algorithm, which after 100 tasks suffers a substantial decrease in performance. The Lookahead algorithm bases its decisions on the analysis of the children nodes for the current task, expecting that those nodes will be scheduled shortly. However, if there are too many concurrent tasks to schedule, as observed for DAGs with more than 100 tasks, the processor load is substantially changed by the concurrent tasks to be scheduled after the current task. This finding implies that when the children tasks are scheduled, the conditions are different and the optimization decision made by the parent task is not valid, which results in poorer performance. Compared with HEFT, our PEFT algorithm improves by 10% for 10 task DAGs. This improvement gradually decreases to 6.2% for 100 task DAGs and to 4% for 500 task DAGs. Despite this significant improvement, we can observe that PEFT maintains the same level of *Slack* as HEFT. Thus, the schedules produced, although shorter, have the same robustness to uncertainty as those produced by HEFT.



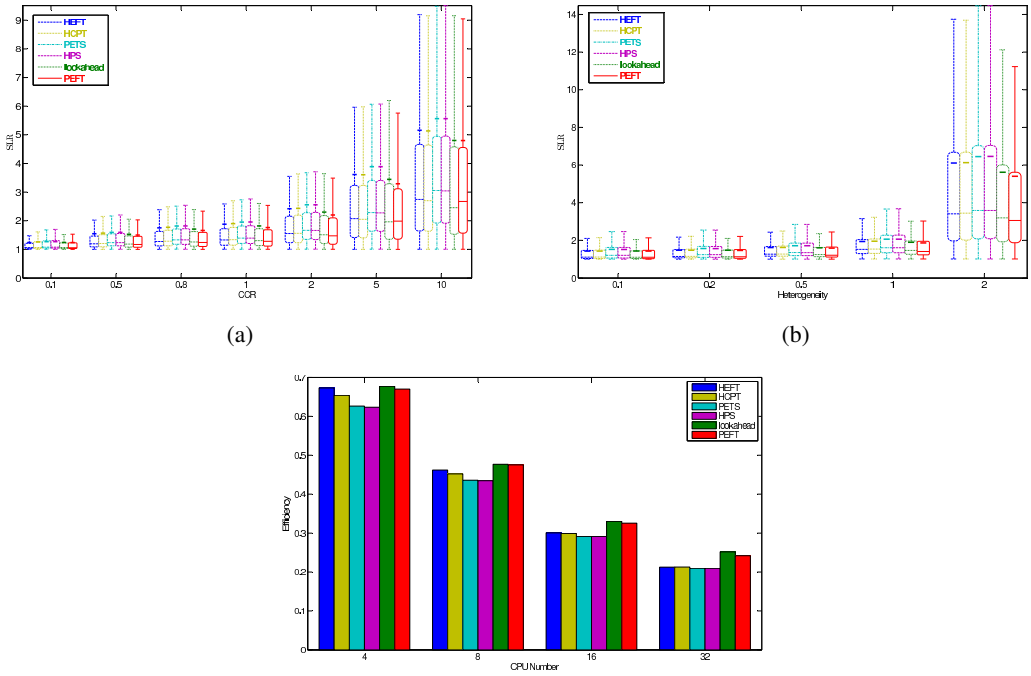
(a) (b)

Figure 3: **(a)** Average SLR and **(b)** slack for random graphs as a function of DAG size

To illustrate the results statistically, boxplots are presented, where the minimum, 25%, mean and 75% values of the algorithm results are represented. The maximum is not shown because there is a broad distribution in the results; therefore, we only show values up to the Upper Inner Fence. We also show the average values, which are indicated by an individual line in each boxplot.

The SLRs obtained for the PEFT, Lookahead, HEFT, HCPT, HPS and PETS algorithms as a function of CCR and heterogeneity are shown in Figure 5a and 5b, respectively. We can see that PEFT has the lowest average SLR and a smaller dispersion in the distribution of the results. The second best algorithm in terms of SLR is Lookahead. In terms of Efficiency, Figure 5c, Lookahead is the best algorithm, with a performance very similar to that of PEFT. This is an important finding because with the proposed algorithm, we improved the SLR and also achieved high values of Efficiency that are only comparable with those of a higher-complexity algorithm. PEFT was the best quadratic algorithm in our simulation.

Table 7 shows the percentage of better, equal and worse results for PEFT when compared with the remaining algorithms, based on *makespan*. Compared with HEFT, PEFT achieves better scheduling in 72% of runs, equivalent schedules in 3% of runs and worse schedules in 25% of



(a) (b) (c)

Figure 4: (a) Boxplot of SLR as a function of CCR; (b) boxplot of SLR as a function of heterogeneity and (c) efficiency for random graphs

runs.

		PEFT	Lookahead	HEFT	HCPT	PETS	HPS
PEFT	better worse equal	*	66% 28% 6%	72% 25% 3%	79% 18% 3%	91% 9% 0%	90% 8% 2%
Look-ahead	better worse equal	28% 66% 6%	*	64% 31% 5%	70% 26% 4%	86% 14% 0%	84% 14% 2%
HEFT	better worse equal	25% 72% 3%	31% 64% 5%	*	29% 20% 51%	91% 8% 0%	72% 9% 18%
HCPT	better worse equal	18% 79% 3%	26% 70% 4%	20% 29% 51%	*	86% 14% 0%	68% 13% 18%
PETS	better worse equal	9% 91% 0%	14% 86% 0%	8% 91% 0%	14% 86% 0%	*	39% 60% 1%
HPS	better worse equal	8% 90% 2%	14% 84% 2%	9% 72% 18%	13% 68% 13%	60% 39% 1%	*

Table 7: Pair-wise schedule length comparison of the scheduling algorithms

3.5.3 Real-world application graphs

In addition to the random graphs, we evaluated the performance of the algorithms with respect to real-world applications, namely Gaussian Elimination [ABK98], Fast Fourier Transform [CR92, THW02], Laplace Equation [WG90], Montage [3][BGL+04, DSS+05] and Epigenomics[4] [BBD+07]. All of these applications are well known and used in real-world problems. Because of the known structure of these applications, we simply used different values for CCR, heterogeneity and CPU number. The range of values that we used in our simulation was $[0.1, 0.5, 0.8, 1, 2, 5, 10]$ for CCR, $[0.1, 0.2, 0.5, 1, 2]$ for heterogeneity and $[2, 4, 8, 16, 32]$ for CPU number. For Montage and Epigenomics, we also considered 64 CPUs. The range of parameters considered here represents typical values within the context of this work [DK08, THW02]. The CCR and heterogeneity represent a wide range of machines, from high-speed networks (CCR=0.1) to slower ones (CCR=10) and from nearly homogeneous systems (heterogeneity equal to 0.1) to highly heterogeneous machines (heterogeneity equal to 2). We also considered a wide range of CPU numbers to simulate higher concurrent environments with 2 processors, as well as low concurrent situations where the total number of processors, in most of the cases, is higher than the maximum number of concurrent tasks ready to be scheduled at any given instant.

3.5.3.1 Gaussian Elimination

For Gaussian Elimination, a new parameter, matrix size m, was used to determine the number of tasks. The total number of tasks in a Gaussian Elimination graph is equal to $\frac{m^2+m-2}{2}$. The values considered for m were $\left[5, 10, 15, 20, 30, 50, 100\right]$. The boxplot of SLR as a function of the matrix size is shown in Figure 5.

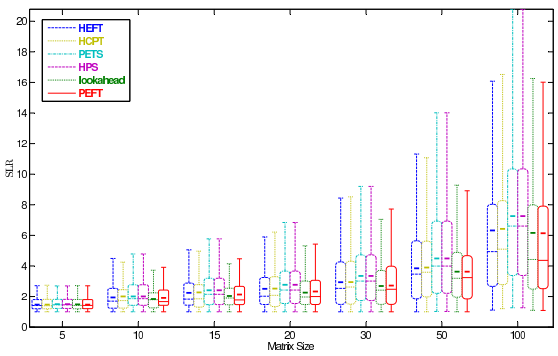


Figure 5: Boxplot of SLR for the Gaussian Elimination graph as a function of matrix size

For all matrix sizes, PEFT produced shorter schedules than all other algorithms with quadratic complexity and almost the same results as the Lookahead algorithm, which has quartic complexity. Figure 6 shows the SLR boxplot as a function of the CCR parameter. For low CCRs, PEFT yielded results equivalent to those of HEFT, and for higher CCRs, PEFT performed significantly better,

[3]Montage, An Astronomical Image Mosaic Engine, http://montage.ipac.caltech.edu/
[4]USC epigenome center, http://epigenome.usc.edu/

obtaining an average improvement of 2%, 9% and 16% over HEFT for CCR values of 2, 5 and 10, respectively. From the boxplots, we can see that statistically, PEFT produced schedules with lower SLR dispersion than the other quadratic algorithms.

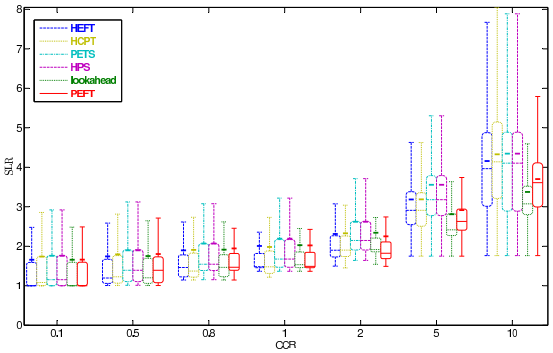


Figure 6: Boxplot of SLR for the Gaussian Elimination graph as a function of the CCR

Concerning heterogeneity, PEFT outperformed HEFT in average SLR by 2%, 3%, 4%, 6% and 7% for heterogeneity values of 0.1, 0.2, 0.5, 1 and 2, respectively. Concerning the number of CPUs, the improvement over HEFT was 2%, 6%, 12% and 12% for sets of 4, 8, 16 and 32 CPUs, respectively. Graphical representation is not provided for SLR as a function of heterogeneity and CPU numbers.

3.5.3.2 Fast Fourier Transform

The second real application was the Fast Fourier Transform (FFT). As mentioned in [THW02], we can separate the FFT algorithm into two parts: recursive calls and the butterfly operation. The number of tasks depends on the number of FFT points (n), where there are $2 \times (n-1)+1$ recursive call tasks and $n \log_2 n$ butterfly operation tasks. Also, in this application, because the structure is known, we simply change CCR, β and the CPU number. Figure 7 shows the SLR for different FFT sizes.

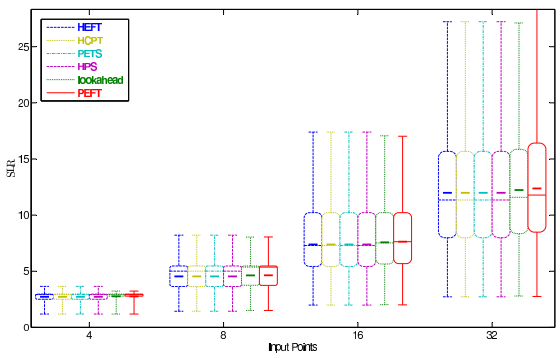
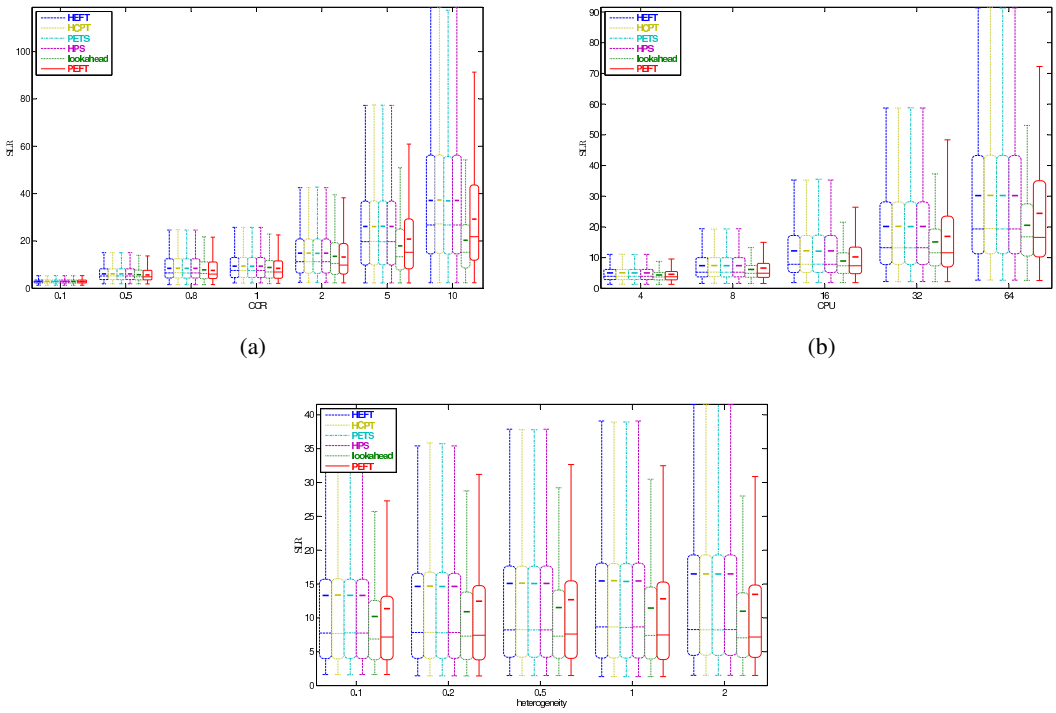


Figure 7: Boxplot of SLR for the Fast Fourier Transform graph as a function of the input points



(a) (b)

(c)

Figure 8: Boxplot of SLR for Montage with respect to (a) CCR, (b) number of CPUs and (c)heterogeneity factor

In this type of application, PEFT and Lookahead yielded the worst results, although the results were similar to those obtained using HEFT. This example allowed us to conclude, with additional experiments, that PEFT does not perform better than HEFT when all tasks belong to a critical path, i.e., when all paths are critical.

3.5.3.3 Montage Workflow

Montage is an application for constructing custom astronomical image mosaics of the sky.

We considered Montage graphs with 25 and 50 tasks, and as in the other real applications, because the graph structure was defined, we simply considered different values of *CCR*, β and CPU number. Figure 8 shows the boxplots of SLR as a function of different hardware parameters.

All algorithms except for PEFT and Lookahead exhibited the same performance for this application. The average SLR improvement for PEFT over HEFT for different values of CCR (Figure 8a) started at 0.8% for a low CCR value (equal to 0.1) and increased to 22% at a CCR value equal to 10. Concerning the number of CPUs, the improvement was 10% with 4 CPUs and increased to 19% for 64 CPUs, as shown in Figure 8b. The improvement for different heterogeneities, Figure 8c, started at 15% for low heterogeneity ($\beta = 0.1$) and increased to 18% for a heterogeneity of 2 ($\beta = 2$).

3.5.3.4 Epigenomic Workflow

The Epigenomic workflow is used to map the epigenetic state of human cells on a genome-wide scale. As was the case for the other real application graphs, the structure of this application is known; therefore, we simply considered different values of *CCR*, β and CPU number. In our experiment, we used graphs with 24 and 46 tasks.

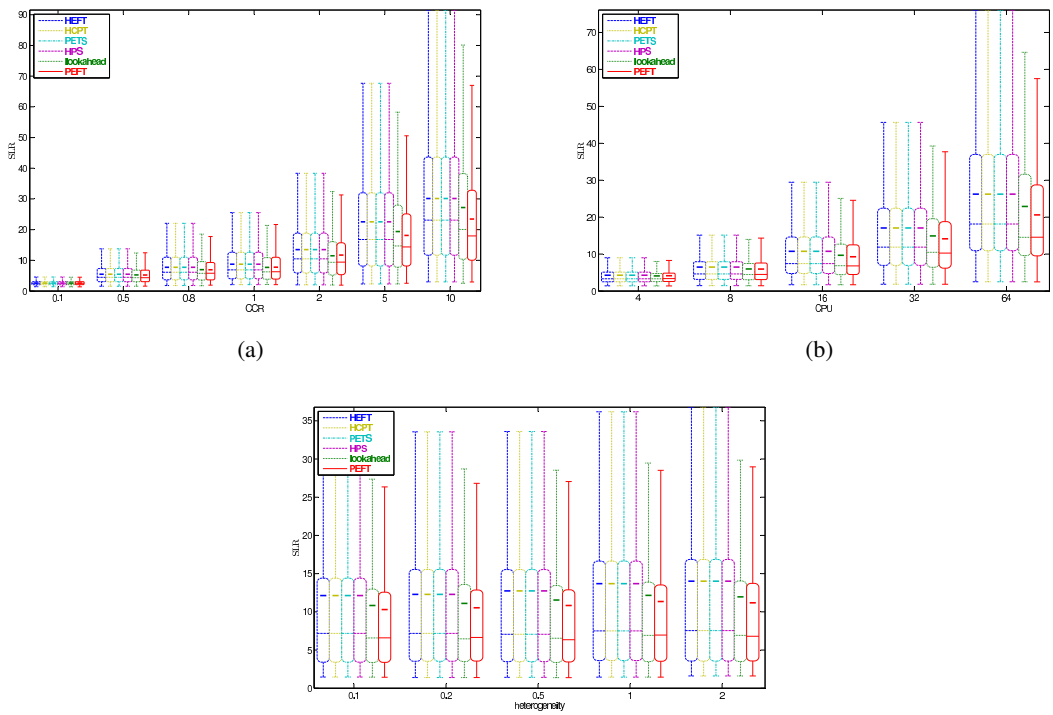


Figure 9: Boxplot of SLR for Epigenomic Workflow as a function of (a) CCR, (b) CPU number and (c) heterogeneity factor

Figure 9 shows the boxplot for SLR as a function of the hardware parameters. Also, for this application, PEFT always outperformed the other algorithms, including the Lookahead algorithm. The average SLR improvement of PEFT over HEFT for a low CCR value of 0.1 was 0.1% and increased to 22% for a CCR value of 10 (Figure 9a). Similarly, PEFT showed (Figure 9b) a range of SLR improvement for different CPU numbers: 3% for 4 CPUs and 21% for 64 CPUs. In addition, we observed an average SLR improvement of 15% to 21% for low heterogeneity ($\beta = 0.1$) to high heterogeneity ($\beta = 2$) (Figure 9c).

3.6 Conclusions

In this paper, we proposed a new list scheduling algorithm with quadratic complexity for heterogeneous systems called PEFT. This algorithm improves the scheduling provided by state-of-the-art quadratic algorithms such as HEFT [THW02]. To our knowledge, PEFT is the first algorithm to

outperform HEFT while maintaining the same time complexity of $O(v^2.p)$. The algorithm is based on an Optimistic Cost Table (OCT) that is computed before scheduling. This cost table represents for each pair (task, processor) the minimum processing time of the longest path from the current task to the exit node by assigning the best processors to each of those tasks. The table is optimistic because it does not consider processor availability at a given time. The values stored in the cost table are used in the processor selection phase. Rather than considering only the Earliest Finish Time (EFT) for the task that is being scheduled, PEFT adds to EFT the processing time stored in the table for the pair (task, processor). All processors are tested, and the one that gives the minimum value is selected. Thus, we introduce the look ahead feature while maintaining quadratic complexity. This feature has been proposed in other algorithms, but all such algorithm increase the complexity to cubic or higher orders.

To prioritize the tasks, we also use the OCT table to define a new rank that is given by the average of the costs for a given task over all processors. Although other ranks could be used, such as $rank_u$ [THW02], we concluded that with the new rank, similar performance is obtained. Therefore, the use of $rank_u$ would require additional computations without resulting in a significant advantage.

In terms of Scheduling Length Ratio (SLR), the PEFT algorithm outperformed all other quadratic algorithms considered in this work for random graph sizes of 10 to 500. Statistically, PEFT had the lowest average SLR and a lower dispersion in the distribution of the results.

We also compared the algorithms in terms of robustness to uncertainty in the task processing time, given by the *Slack* function, and we obtained the same level of robustness for PEFT and HEFT, which is an important characteristic of the proposed algorithm.

The simulations performed for real-world applications also verified that PEFT performed better then the remaining quadratic algorithms. These tests also revealed an exceptional case (the FFT transform) in which PEFT did not perform better. We concluded that this lack of improvement by PEFT occurs for graphs with the same characteristics as FFT and that are characterized by having all tasks belong to a critical path, i.e., having all paths be critical.

From the results, we can conclude that among the static scheduling algorithms studied in this paper, PEFT exhibits the best performance for the static scheduling of DAGs in heterogeneous platforms with quadratic time complexity and the lowest quadratic time complexity.

Chapter 4

Fairness resource sharing for dynamic workflow scheduling on Heterogeneous Systems

Hamid Arabnejad and Jorge G. Barbosa

10th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA),

pages 633-639, 2012,

DOI: 10.1109/ISPA.2012.94,

abstract

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

4.1 Introduction

Heterogeneous Computing Systems (HCS) are characterized by having a variety of different types of computational units and are widely used for executing parallel applications, especially scientific workflows. The workflow consist of many tasks with logical or data dependencies that can be dispatched to different computation nodes in the HCS. To achieve efficient execution of a workflow and minimize the turnaround time, we need an effective scheduling strategy that decides when and which resource must execute the tasks of the workflow. When scheduling multiple independent workflows that represent user jobs and, consequently, are submitted at different instants of time, the common definition of makespan needs to be extended in order to account the waiting time as well as the execution time of a given workflow, contrary to the makespan definition for single workflow scheduling [KA99]. The metric to evaluate a dynamic scheduler of independent workflows has to represent the individual makespan instead of a global measure for the set of workflows, in order to measure the Quality of Service experienced by the users which is related to the finish time of each user application.

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

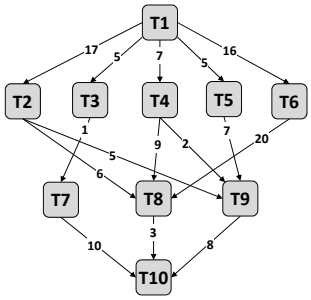
The DAG scheduling problem has been shown to be NP-complete [CB76]. DAG scheduling is mainly divided in two major categories, namely Static Scheduling and Dynamic Scheduling. Most of the scheduling algorithms in static category are restricted to single DAG scheduling. In [CJSZ08] the authors compared 20 scheduling heuristics for single DAG scheduling. For static scheduling of multiple DAGs there were proposed some algorithms such as in [HS06] for homogeneous non-parallel task graphs with the aim of increasing system efficiency, [ZS06] for heterogeneous clusters and non-parallel task graphs and [NS09] for multi-clusters and parallel task graphs. In [ZS06] the authors proposed an algorithm for scheduling several DAGs at the same time where the aim was to achieve fairness in the resource sharing, defined by the slowdown each DAG experiences as a results of competing for resources. In [NS09] the authors proposed several strategies of sharing the resources based on the proportional share. They defined a proportional share based on critical path, width and work of each DAG. They also proposed a weighted proportional share that represent a better tradeoff between fairness resource sharing and makespan reduction of the DAGs. Both works differ from what is proposed here due to their static approach and due to the objective functions considered. Here we consider a dynamic scheduling and we focus on the response time the system gives to each application.

For dynamic scheduling multiple parallel task graphs on a heterogeneous system, it was proposed an algorithm in [BM11a] that minimizes the overall makespan, that is the finish time of all DAGs. In [HHW11] and [YS08] there were proposed two algorithms namely OWM and RANK_HYBD for dynamic scheduling of multiple workflows. Both algorithms were proposed for the same context of the algorithm proposed here.

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

4.2 Workflow Representation

A workflow application can be represented by a Directed Acyclic Graph (DAG), $G(V,E,P,W)$ as shown in Figure 1, where V is the set of v tasks and E is the set of e communication edges between



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

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

tasks. Each $e(i,j) \in E$ represents the task-dependency constraint such that task n_i should complete its execution before task n_j can be started. P is the set of p heterogeneous processors available in the system. W is a $v \times p$ computation cost matrix, where v is the number of tasks and p is the number of processors in the system. $w_{i,j}$ gives the estimate time to execute task v_i on machine p_j. The mean execution time of task n_i can be calculated by $\overline{w_i} = (\sum_{j \in P} w_{i,j})/p$. Each edge $e(i,j) \in E$ is associated with a non-negative weight $c_{i,j}$ representing the communication cost between the tasks n_i and n_j. Once this value could be computed only after defining where tasks i and j will be executed, it is common to compute the average communication costs to label the edges [THW02]. The average communication cost $\overline{c_{i,j}}$ of an edge $e(i,j)$ can be calculated by $\overline{c_{i,j}} = \overline{L} + \frac{data_{i,j}}{\overline{B}}$ where $\overline{L}$ is the average latency time of all processors and $\overline{B}$ is the average bandwidth of all links connecting the set of P processors. $data_{i,j}$ is the amount of data elements that task i needs to send to task j. Note that, when task i and j are assigned to the same processor, the real communication cost is considered to be zero because it is negligible compared to interprocessor communication costs.

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

- **pred**$(\mathbf{n_i})$: denotes the set of immediate predecessors of task n_i in a given DAG. A task with no predecessors is called an *entry* task, n_{entry}. If a DAG has multiple entry tasks, a dummy entry task with zero weight and zero communication edges is added to the graph.

- **succ**$(\mathbf{n_i})$: denotes the set of immediate successors of task n_i. A task with no successors is called an *exit* task, n_{exit}. Like the entry task, if a DAG has multiple exit tasks, a dummy exit

task with zero weight and zero communication edges from current multiple exit tasks to this dummy node is added.

- **makespan**: it is the finish time of the exit task in the scheduled DAG, and is defined by $makespan = AFT(n_{exit})$ where $AFT(n_{exit})$ denotes the Actual Finish Time of the exit task.

- **Critical Path(CP)**: the CP of a DAG is the longest path from n_{entry} to n_{exit} in the graph. The length of this path $|CP|$ is the sum of the computation costs of the tasks and intertask communication costs along the path. The $|CP|$ value of a DAG is the lower bound of the makespan. In this paper, the makespan includes de execution and waiting time spent in the system.

- **EST($\mathbf{n_i}, \mathbf{p_j}$)**: denotes the *Earliest Start Time* of a node n_i on a processor p_j.

- **EFT($\mathbf{n_i}, \mathbf{p_j}$)**: denotes the *Earliest Finish Time* of a node n_i on a processor p_j.

4.3 Related Work

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

Zhao and Sakellariou [ZS06] presented two approaches based on fairness strategy for multi DAGs scheduling. The fairness is defined on the basis of slowdown that each DAG would experience (the slowdown is the difference in the expected execution time for the same DAG when scheduled together with other workflows and when scheduled alone). They proposed two algorithms, one fairness policy based on finish time and another fairness policy based on current time. Both algorithms, at first, scheduled each DAG on all processors with the static scheduling (like HEFT [THW02] or Hybrid.BMCT [SZ04]) as the pivot scheduling algorithm, save its schedule assignment and keep its makespan as the slowdown value of the DAG. Next, sort all DAGs in descending order of their slowdown in the list. Then until there are unfinished DAGs into the list, the algorithm selects the first DAG with highest slowdown and then selects the first ready task that has not been scheduled on the DAG. The key idea is to evaluate the slowdown value of each DAG after scheduling a task and make a decision on which DAG should be selected to schedule the next task. The difference between the two fairness based algorithms proposed is that the Fairness Policy based on Finish Time, calculates the slowdown value of *only* the selected DAG, whereas in the Fairness Policy based on Current Time, the slowdown value is recalculated for *every* DAG. But these two algorithms are designed to schedule multiple workflow applications that are known at the same time (off-line scheduling). Here we consider the scheduling of dynamic workflows, meaning that they arrive at different instants, where the age and remaining time of each concurrent DAG is considered by the scheduler (on-line scheduling).

Z. Yu and W. Shi in [YS08] proposed a planner-guided strategy (called RANK_HYBD algorithm) to deal with dynamic workflow scheduling of applications that are submitted by different

users at different instants of time. The RANK_HYBD algorithm ranks all tasks using $rank_u$ priority measure [THW02]. In each step, the algorithm reads all ready tasks from all DAGs and selects the next task to schedule based on their rank. If the ready tasks belong to different DAGs, the algorithm selects the task with lowest rank and if they belong to the same DAG, the task with highest rank is selected. With this strategy, The RANK_HYBD algorithm allows the DAG with lowest rank (lower makespan) to be scheduled first to reduce the waiting time of the DAG in the system. But this strategy does not achieve good fairness because it always like to finish first lower DAGs in the system and postpones the higher DAGs. For instance, if a longer DAG is being executed and several lower DAGs are submitted to the system, the scheduler postpones the execution of the longer DAG to give priority to the smaller ones.

Hsu, Huang and Wang in [HHW11] proposed Online Workflow Management (OWM) for scheduling multiple online workflows. In OWM algorithm, unlike in the RANK_HYBD that puts all ready tasks from each DAG into the ready list, it selects only a single ready task from each DAG with highest rank into the ready list. Then until there are unfinished DAGs on the system, the OWM algorithm selects the task with highest priority from ready list. Then it calculates the earliest finish time (EFT) for the selected task on each processor and selects the processor with minimum earliest finish time. If the selected processor is free at that time, the OWM algorithm assigns the selected task to the selected processor otherwise keeps the selected task in the ready list in order to be scheduling later. In their results, The OWM algorithm has better performance than RANK_HYBD [YS08] and Fairness_Dynamic (modified version of fairness algorithm [ZS06]) in handling online workflows. The results of different mean arrival intervals according to different performance metrics show that the OWM algorithm outperforms Fairness_Dynamic by 26% and 49%, and outperforms RANK_HYBD by 13% and 20% for average makespan and average SLR (defined by eq. 5), respectively. As the RANK_HYBD algorithm, OWM uses a fairness strategy but, instead of scheduling smaller DAGs first, it selects and schedules tasks from the longer DAGs first. OWM has better strategy by filling the ready list with one task from each DAG so that it gives to all DAGs the chance to be selected in current time for scheduling. In their simulation environment, the number of processors is always near to the number of workflows so that in the most cases the scheduler has suitable number of processors to schedule the ready tasks. This choice does not expose a fragility of the algorithm that occurs when the number of DAGs is significant in relation to the number of processors or, the same is to say, for heavier loaded systems. Another problem with the OWM algorithm is in the processor selection phase where if the processor with earliest finish time for the selected task is not free, the algorithm postpones that task and keeps it in the ready list. If the system is heavy loaded, it is possible that in the next task selection, it is postponed again because meanwhile it may arrive new tasks with highest rank.

Both algorithms, RANK_HYBD and OWM, present results in terms of average makespan. This metric combines in the same way long and short DAGs and do not allow to infer the average waiting time spent by the DAGs individually. In this paper we propose new strategies in both aspects of selecting tasks from ready list and in the processor assignment in order to reduce the

individual completion time of the DAGs, that is the total time between the submission and the completion time which includes execution and waiting time. We use the metric Schedule Length Ratio (SLR) that is a normalized measure of the completion time and that is more appropriated to conclude about the individual performance experienced by each user.

4.4 Fairness Dynamic Workflow Scheduling

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

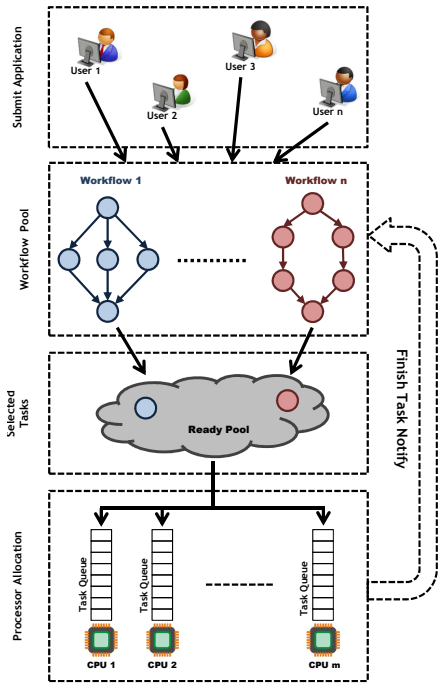


Figure 2: Fairness Dynamic Workflow Scheduling (FDWS) System

Next we describe all these parts in detail:

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

priority to tasks in the DAG, we used the upward rank [THW02]. $rank_u$ represents the length of the longest path from task n_i to the exit node, including the computational cost of n_i and is given by equation 10.

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

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

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

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

The metric $rank_r$ considers the Percentage of Remaining Task number (PRT) of the DAG and its Critical Path Length (CPL). The PRT value gives more priority to DAGs that are almost completed and only have few tasks to execute. This strategy is different from the Smallest Remaining Processing Time (SRPT) [KSW97]. The SRPT algorithm, on each step, selects and schedules the application with the smallest remaining processing time. The remaining processing time is the time needed to execute all remaining tasks of the workflow. This is very different from our strategy. As an example, if we have two workflows with the same number of remaining tasks like it is shown in Figure3 (tasks with same name have the same computational time), using the SRPT strategy, there is no difference between these

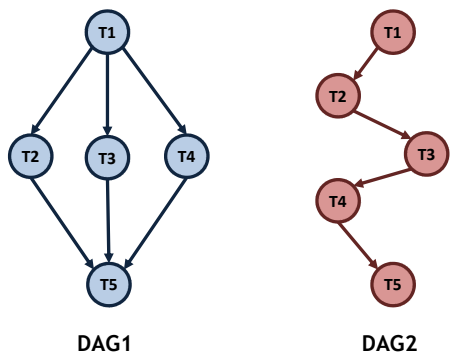


Figure 3: Two sample workflows

DAGs to select the next task, but attending to the response time, it is better to select a task from DAG1 because it has a lower expected finish time. With $rank_r$ we also consider

the existing concurrency in each level by using the critical path (as it can be seen DAG1 has lower critical path length than DAG2 and therefore it would be schedule first). Also selecting only the smallest remaining time is not necessarily better; consider an application that is being executed and that have a few tasks remaining. With SRPT strategy, the application would be postponed if shorter DAGs arrive, which would increase the response time to the user. By using $rank_r$ we consider the percentage of remaining tasks to give the opportunity to the DAGs with fewer remaining tasks to be finished even if the work on those tasks is higher than the work on the new arrived DAGs.

Note that in RANK_HYDB and OWM, only the individual $rank_u$ is used for selecting tasks into the pool and to select one from the pool of ready tasks which leads to a scheduling decision that do not considers the DAG history in the workflow pool.

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

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

4.5 Experimental Results and Discussion

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

4.5.1 DAG structure

To evaluate the relative performance of the algorithm, we considered randomly generated workflow (DAG) application graphs. For this purpose, we used a synthetic DAG generation program[1].

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

Algorithm 1 The FDWS algorithm

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

There are four parameters that define a DAG shape:

- ***n***: number of computation nodes in the DAG (i.e., application tasks);
- ***fat***: gives the width of the DAG by $e^{fat*log(n)}$, that is the maximum number of tasks that can be executed concurrently. A small value will lead to a thin DAG, like a chain, with a low task parallelism, while a large value induces a fat DAG, like a fork-join, with a high degree of parallelism;
- ***density***: denotes the number of edges between two levels of the DAG, with a low value leading to few edges and a large value leading to many edges;
- ***regularity***: the regularity denotes the uniformity of the number of tasks in each level. A low value means that levels contain very dissimilar numbers of tasks, while a high value means that all levels contain similar numbers of tasks;
- ***jump***: indicates that an edge can go from level l to level $l + jump$. A jump of 1 is an ordinary connection between two consecutive levels.

In this paper, we used the synthetic DAG generator only for making the DAG structure which includes the specific number of nodes and their dependencies. In our experiment, for random DAG generation, we consider $n = [10, 20, 30, 40, 50, 60]$, $jump = [1, 2, 4]$, $regularity = [0.2, 0.8]$,

$fat = [0.1, 0.4, 0.8]$ and $density = [0.2, 0.8]$. With these parameters we have 216 different structure DAGs for our experiment. The DAG structure is presented apart from other simulation parameters because the structure of the workflows is dependent from users requests and independent from the hardware environment.

4.5.2 Environment system parameters

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

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

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

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

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

4.5.3 Performance metrics

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

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

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

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

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

  The denominator in the SLR equation is the minimum computation cost of the critical path tasks. As the numerator represents the total time spent by a DAG in the system, waiting and execution time, a low value, such as 1, means that the response time was the lowest possible for that DAG in the target system. On the other hand, a higher value means that, due to the concurrent DAG scheduling, the DAG required more time to complete. In this sense, the SLR is a metric of Quality of Service experienced by the users. We used the SLR value for each Scenario ($SLR_{Scenario}$) and it is equal to average SLR values of all DAGs in each Scenario.

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

4.5.4 Results and discussion

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

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

The results shown in [HHW11] that compare OWM with RANK_HYBD show close results for both algorithms although OWM performs slightly better. In our experiment this difference is not obvious and in some cases RANK_HYBD performs better. This is mainly because in [HHW11] the authors considered that they have 100 DAGs for 90 to 150 processors that is, the number of DAGs is in general less than the number of processors available. Consequently, the concurrency is lower than in our experiment.

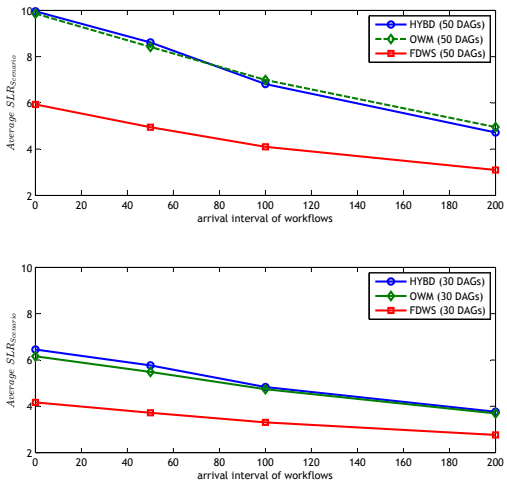


Figure 4: Results of different mean arrival times for average SLR for 30 and 50 concurrent DAGs

Table 8 shows the improvement of FDWS over the two other concerning Overall Makespan. We can see that OWM achieves a better result showing that OWM obtains a total time to process all DAGs shorter than the others.

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

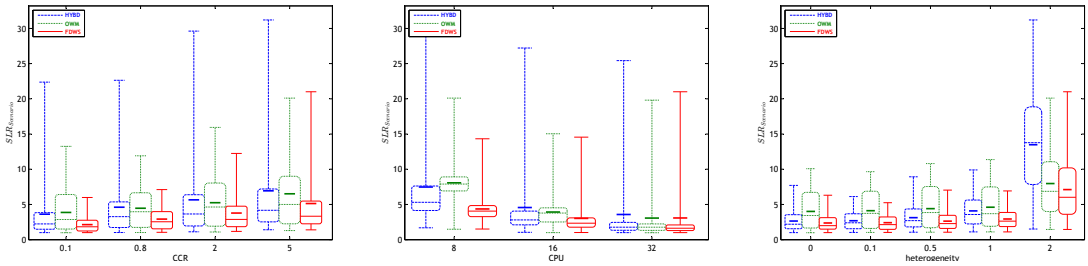
Table 8: Overall Makespan improvement of FDWS over the RANK_HYBD and OWM algorithm; a negative value means that FDWS as a worse result

Figure 5 shows boxplots for $SLR_{scenario}$ as a function of different hardware parameters such as CCR, heterogeneity factor and CPU number. The mean value is also shown by a individual stronger line. We can see that FDWS has statistically better behaviour for all CCRs, heterogeneity factor and CPU number. Lower mean and median values and also less dispersion.

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

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

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



(a) (b) (c)

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

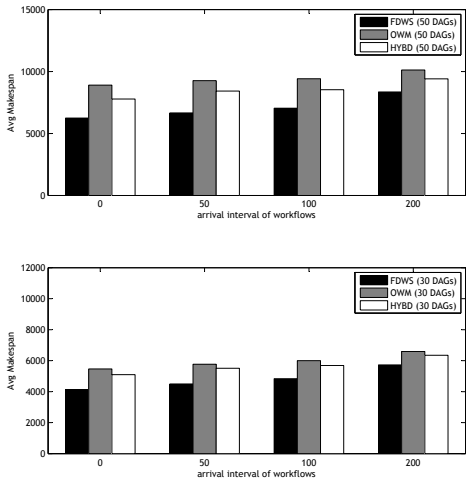


Figure 6: Results of different mean arrival times for Average Makespan

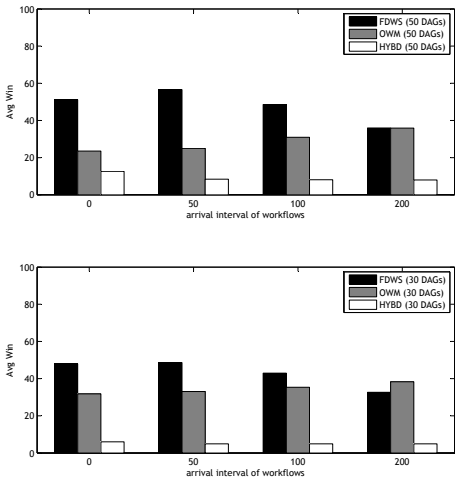


Figure 7: Results of different mean arrival times for Win(%)

4.6 Conclusions

Most workflow scheduling algorithms focused on single workflows and there are only a few works on multiple workflow scheduling. In this paper, we presented a new algorithm called FDWS (Fair-

ness Dynamic Workflow Scheduling) and compared it with two recent algorithms, namely OWM [HHW11] and RANK_HYBD [YS08] algorithms that deal with multiple workflow scheduling in dynamic situations. Based on our experiments, FDWS has better improvement in terms of SLR, Win(%) and Average Makespan, showing better Quality of Service characteristics. The drawback of this feature is to obtain a longer Overall execution time. As future work, we intend to find if the SLR improvement is the main reason to increase the Overall Makespan or if both characteristics can be reduced.

Acknowledgements

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

Chapter 5

Fair Resource Sharing for Dynamic Scheduling of Workflows on Heterogeneous Systems

Hamid Arabnejad, Jorge G. Barbosa and Frédéric Suter

High-Performance Computing on Complex Environments, chapter 9,,

John Wiley & Sons, Parallel and Distributed Computing Series (51), pages 145-167, June 2014,

DOI: 10.1002/9781118711897.ch9,

abstract

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.

5.1 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.

5.1.1 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 $V = \{n_j, j = 1 \ldots v\}$ 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. 5.1.2.

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:

$$c_{i,j} = L_m + \frac{data_{i,j}}{B_{m,n}}. \tag{1}$$

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:

$$\overline{c_{i,j}} = \overline{L} + \frac{data_{i,j}}{\overline{B}}, \tag{2}$$

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 [THW02].

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:

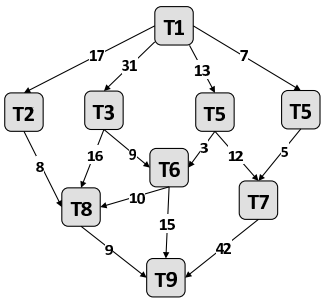
$$\overline{w_i} = \sum_{j=1}^{p} \frac{w_{i,j}}{p}. \tag{3}$$

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 that presents a DAG and a target system with three processors and the corresponding communication and computation costs. In Fig. 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: 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:

$$makespan = AFT(n_{exit}) - AST(n_{entry}), \tag{4}$$

  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:

$$level(n_i) = \max_{q \in pred(n_i)} \{level(q)\} + 1. \tag{5}$$

- *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:

$$EST(n_i, p_j) = \max\left\{T_{Available}(p_j), \max_{n_m \in pred(n_i)} \left\{AFT(n_m) + c_{m,i}\right\}\right\}, \tag{6}$$

  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 $EST\left(n_{entry}, p_j\right) = \max\{T_s, T_{Available}(p_j)\}$, 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:

$$EFT(n_i, p_j) = EST(n_i, p_j) + w_{i,j}, \tag{7}$$

  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).

5.1.2 SYSTEM MODEL

Typically, for executing complex workflows, a high-performance cluster or grid platform is used. As defined in [BB99], 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 2 illustrates the typical cluster architecture.

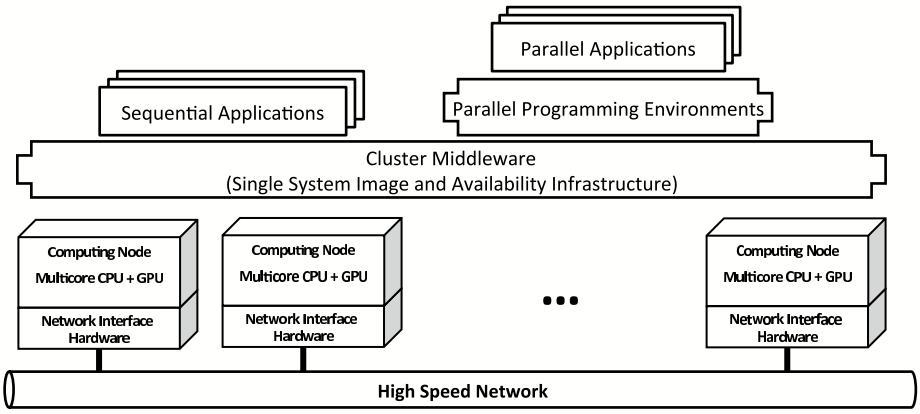


Figure 2: Conceptual cluster architecture.

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) 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.

5.1.3 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:

$$\mathrm{TTR} = \frac{\mathrm{TurnaroundTime}}{\sum\limits_{n_i \in CP} \min\limits_{p_j \in P} \left(w_{(i,j)} \right)}, \tag{8}$$

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:

$$\mathrm{NTT}(G,a_i) = \frac{\min\limits_{a_k \in A}\{\mathrm{TurnaroundTime}(G,a_k)\}}{\mathrm{TurnaroundTime}(G,a_i)}, \tag{9}$$

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.

5.2 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.

5.2.1 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 [ZS06]. 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 [THW02] or Hybrid.BMCT [SZ04]) 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 [NS09], 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 [BM10a], 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 [CDS10], 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 [HCTY+12], 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.

5.2.2 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 [KA99]. 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 [LCJY09], 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 [XCWB09], 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 [BM11a], 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.

5.2.2.1 Rank Hybrid algorithm

A planner-guided strategy, the RANK_HYBD algorithm, was proposed by Yu and Shi [YS08] 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 [THW02], 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:

$$rank_u(n_i) = \overline{w_i} + \max_{n_j \in succ(n_i)} \{\overline{c_{i,j}} + rank_u(n_j)\}, \tag{10}$$

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 getReadyPool algorithm

```
if ( a new workflow has arrived ) {
   calculate rank_u for all tasks of the new workflow
}
Ready_Pool ← Read all ready tasks from all DAGs
multiple ← 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 2.

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.

Algorithm 2 RANK_HYBD algorithm

```
while ( there are workflows to schedule ) {
  Ready_Pool ← getReadyPool()
  Resources_free ← get all idle resources
  while ( Ready_Pool ≠ ϕ & Resources_free ≠ ϕ ) {
    task_selected ← the first task in Ready_Pool
    resource_selected ← 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
  }
}
```

5.2.2.2 On-line Workflow Management

The on-line workflow management algorithm (OWM) for the on-line scheduling of multiple workflows was proposed in [HHW11]. 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 3.

In the results presented by Hsu et al. [HHW11], the OWM algorithm performs better than the RANK_HYBD algorithm [YS08] and the Fairness_Dynamic algorithm (a modified version of the fairness algorithm proposed by Zhao and Sakellariou [ZS06]) 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.

5.2.2.3 Fairness Dynamic Workflow Scheduling

The fairness dynamic workflow scheduling (FDWS) algorithm was proposed in [AB12a]. 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.

Algorithm 3 OWM algorithm

```
while ( there are workflows to schedule ) {
  Ready_Pool ← getReadyPool()
  Resources_free ← get all idle resources
  while ( Ready_Pool ≠ ϕ & Resources_free ≠ ϕ ) {
    task_selected ← the first task in Ready_Pool
    resource_selected ← the processor with the lowest Finish Time for task_selected on Resources_free
    if ( number of free clusters == 1 &
         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
    }
  }
}
```

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, $rank_u$ (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 (2), and the task with highest $rank_r$ is selected:

$$rank_r(n_{i,j}) = \frac{1}{PRT(DAG_j)} \times \frac{1}{|CP(DAG_j)|}. \tag{11}$$

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) [KSW97]. 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 4.

Algorithm 4 FDWS algorithm

```
while ( Workflow_Pool ≠ ϕ ) {
  if ( new workflow has arrived ) {
    Compute rank_u for all tasks of the new Workflow
    Insert the Workflow into Workflow_Pool
  }
  Ready_Pool ← one ready task from each DAG (highest rank_u)
  Compute rank_r(n_{i,j}) for each task n_i ∈ DAG_j in Ready_Pool
  Resources_free ← get all idle resources
  while ( Ready_Pool ≠ ϕ & Resources_free ≠ ϕ ) {
    task_selected ← the task with highest rank_r from Ready_Pool
    resource_selected ← 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
  }
}
```

5.2.2.4 On-line Min-Min and On-line Max-Min

The min-min and max-min algorithms have been studied extensively in the literature [MAS+99], 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.

5.3 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.

5.3.1 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 $n = \{20 \ldots 50\}$, $jump = \{1,2,3\}$, $regularity = \{0.2, 0.4, 0.8\}$, $fat = \{0.2, 0.4, 0.6, 0.8\}$, and $density = \{0.2, 0.4, 0.8\}$. With these parameters, we call the DAG generator for each DAG, and it randomly chooses the value for each parameter from the parameter dataset.

5.3.2 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] [CLQ08] 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. 2. In many research papers

[1] https://github.com/frs69wq/daggen
[2] http://simgrid.gforge.inria.fr

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 [VL09].

To make our simulations even more realistic, we consider platforms derived from clusters in the Grid5000 platform deployed in France[3] [CCD+05]. 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 13 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.

Table 9: 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	

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.

5.3.3 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

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

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.

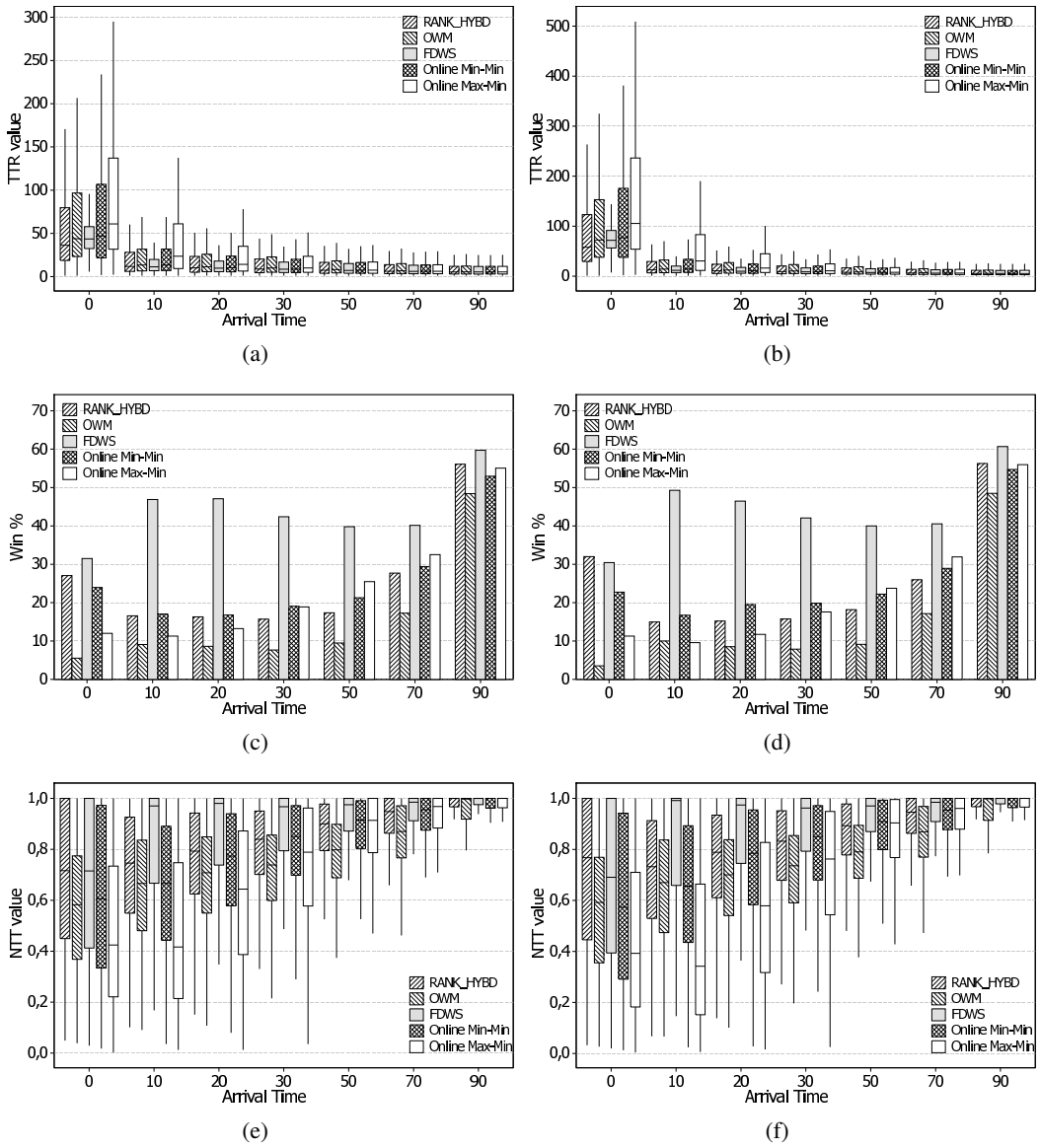


Figure 3: Results of TTR, percentage of wins and NTT on Grenoble site with 9 processors. (a)(c)(e) 30 concurrent DAGs. (b)(d)(f) 50 concurrent DAGs.

Figures 3, 4, 5, and 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

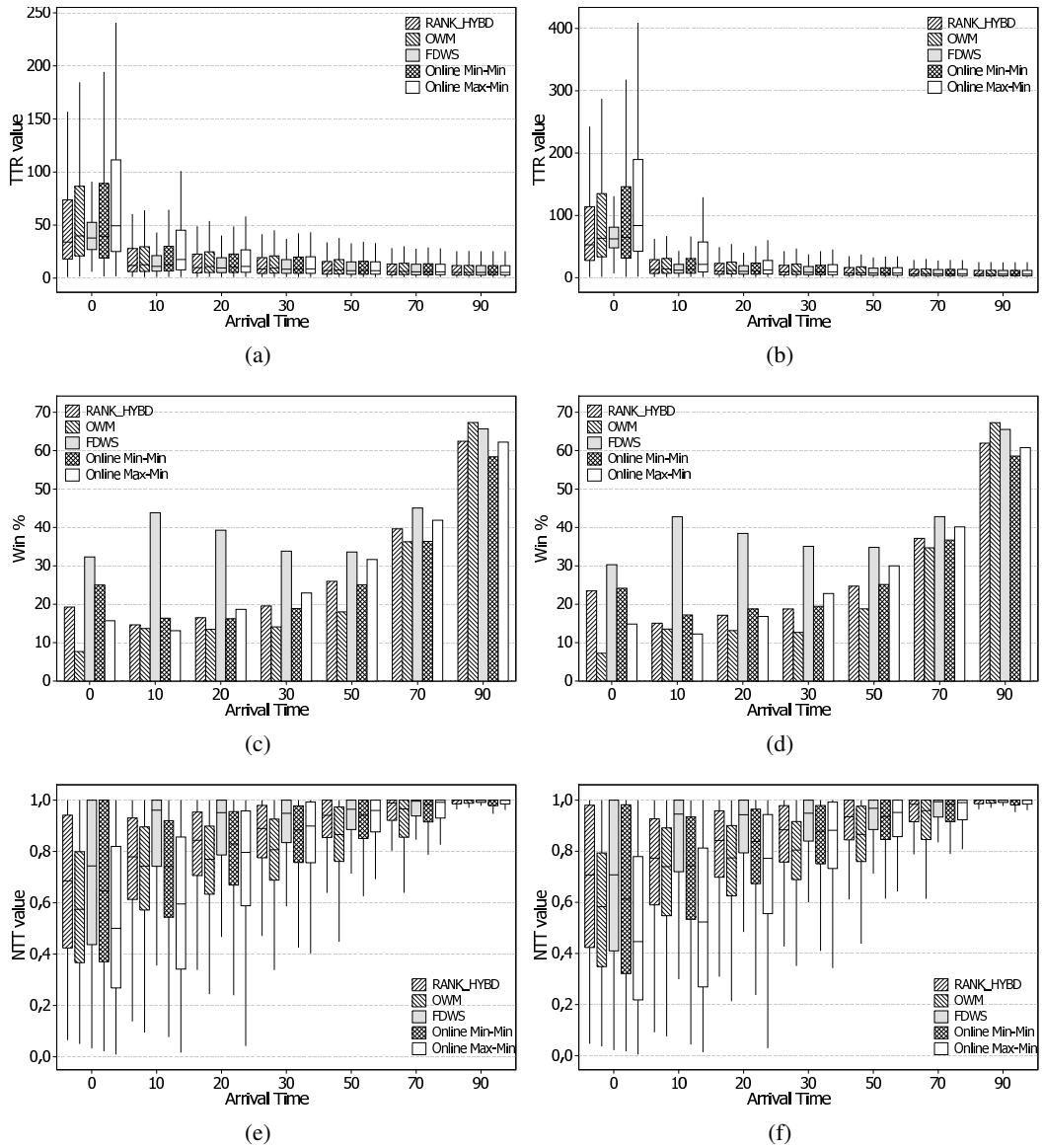


Figure 4: Results of TTR, percentage of wins and NTT on Grenoble site with 15 processors. (a)(c)(e) 30 concurrent DAGs. (b)(d)(f) 50 concurrent DAGs.

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%,

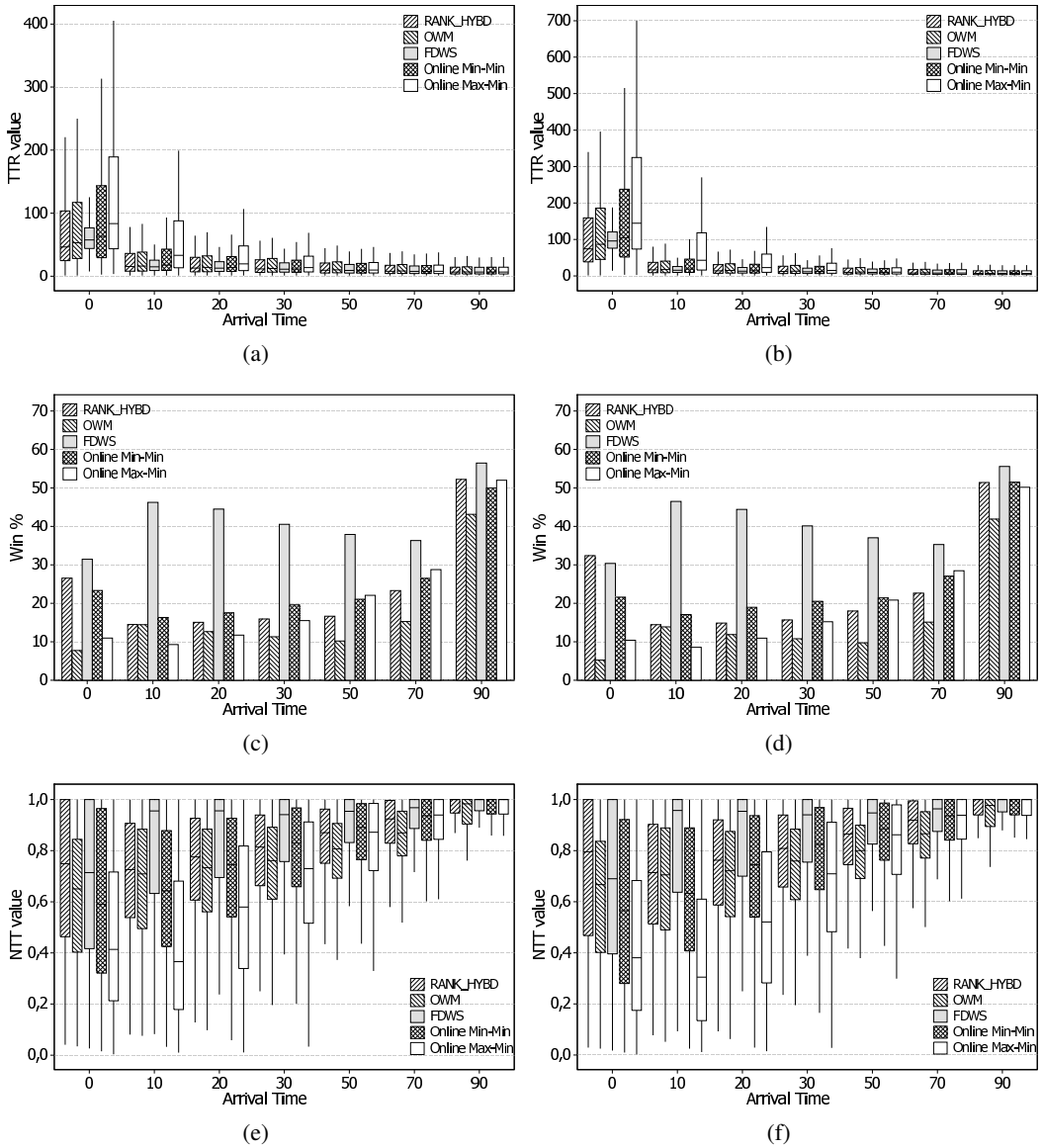


Figure 5: Results of TTR, percentage of wins and NTT on Rennes site for 8 processors. (a)(c)(e) 30 concurrent DAGs. (b)(d)(f) 50 concurrent DAGs.

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 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

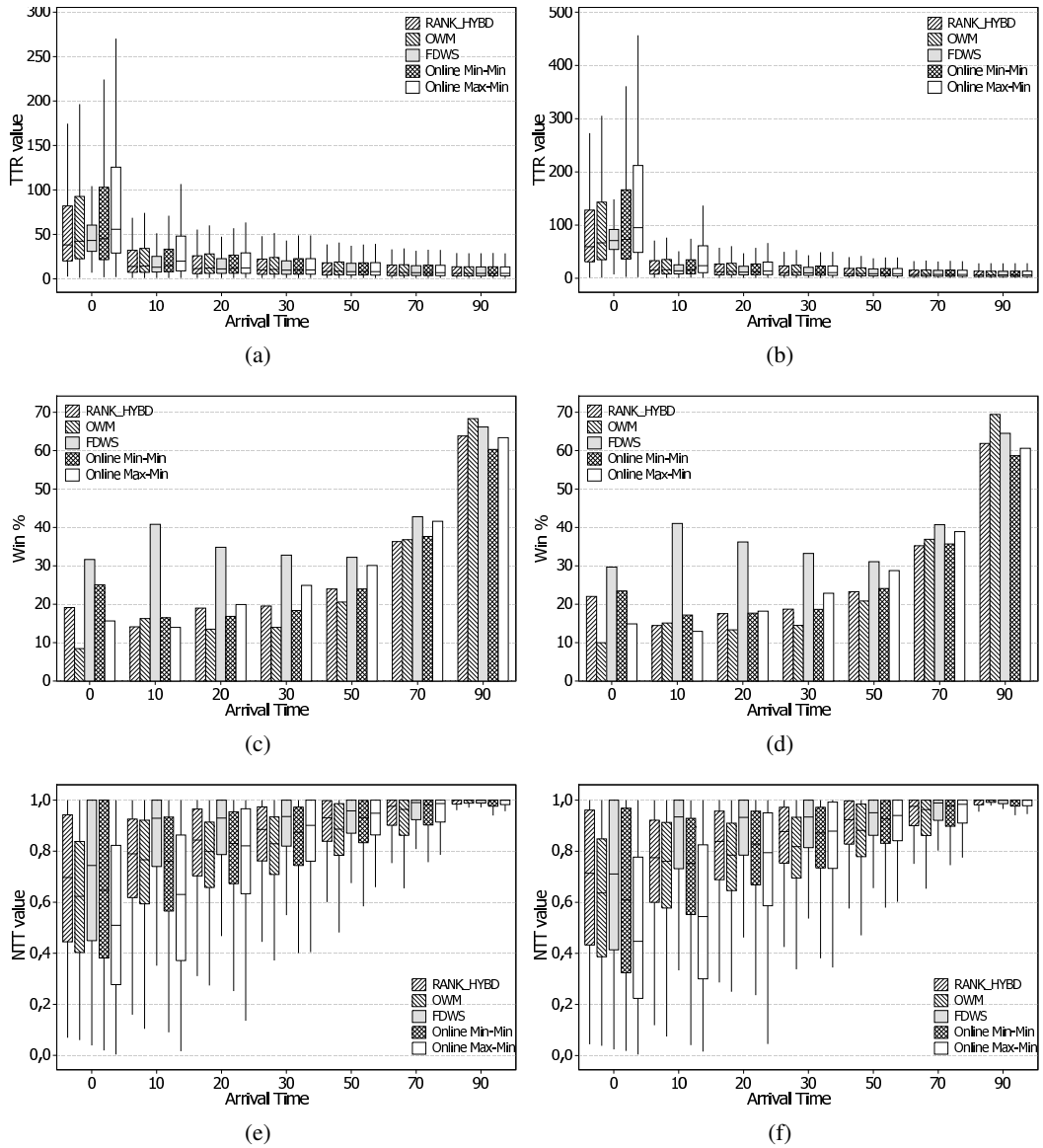


Figure 6: Results of TTR, percentage of wins and NTT on Rennes site with 16 processors. (a)(c)(e) 30 concurrent DAGs. (b)(d)(f) 50 concurrent DAGs.

also has a distribution closer to one, which indicates that its solutions are closer to the minimum turnaround time than the other algorithms.

5.4 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 [AB12a], OWM [HHW11], RANK_HYBD [YS08], 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 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 Computing on Complex Environments, Working Group 3: Algorithms and tools for mapping and executing applications onto distributed and heterogeneous systems.

Chapter 6

A Budget Constrained Scheduling Algorithm for Workflow Applications

Hamid Arabnejad and Jorge G. Barbosa

Journal of Grid Computing,

volume 12, number 4, pages 665–679, Springer, 2014,

DOI: 10.1007/s10723-014-9294-7,

abstract

Service-oriented computing has enabled a new method of service provisioning based on utility computing models, in which users consume services based on their Quality of Service (QoS) requirements. In such pay-per-use models, users are charged for services based on their usage and on the fulfilment of QoS constraints; execution time and cost are two common QoS requirements. Therefore, to produce effective scheduling maps, service pricing must be considered while optimising execution performance. In this paper, we propose a Heterogeneous Budget Constrained Scheduling (HBCS) algorithm that guarantees an execution cost within the user's specified budget and that minimises the execution time of the user's application. The results presented show that our algorithm achieves lower makespans, with a guaranteed cost per application and with a lower time complexity than other budget-constrained state-of-the-art algorithms. The improvements are particularly high for more heterogeneous systems, in which a reduction of 30% in execution time was achieved while maintaining the same budget level.

6.1 Introduction

Utility computing is a service provisioning model that provides computing resources and infrastructure management to customers as they need them, as well as a payment model that charges for usage. Recently, service-oriented grid and cloud computing, which supply frameworks that allow users to consume utility services in a secure, shared, scalable, and standard network environment, have become the basis for providing these services.

Computational grids have been used by researchers from various areas of science to execute complex scientific applications. Recently, utility computing has been rapidly moving towards a pay-as-you-go model, in which computational resources or services have different prices with different performance and Quality of Service (QoS) levels [BVB08]. In this computing model, users consume services and resources when they need them and pay only for what they use. Cost and time have become the two most important user concerns. Thus, the cost/time trade-off problem for scheduling workflow applications has become challenging. Scheduling consists of defining an assignment and mapping of the workflow tasks onto resources. In general, the scheduling problem belongs to a class of problems known as NP-complete [CB76].

Many complex applications in e-science and e-business can be modelled as workflows [DBG+03]. A popular representation of a workflow application is the Directed Acyclic Graph (DAG), in which nodes represent individual application tasks, and the directed edges represent inter-task data dependencies. Many workflow scheduling algorithms have been developed to execute workflow applications. Some typical workflow scheduling algorithms were introduced in [YBR08]. Most of these algorithms have a single objective, such as minimising execution time (makespan). However, additional objectives can be considered when scheduling workflows onto grids, based on the user's QoS requirements. If we consider multiple QoS parameters, such as budgets and deadlines, then the problem becomes more challenging.

The contributions of this paper are as follows: a) a new low time complexity algorithm that obtains higher performance than state-of-the-art algorithms of the same class for the two set-ups considered here, namely i) minimising the makespan for a given budget and ii) budget-deadline constrained scheduling; b) a realistic simulation that considers a bounded multi-port model in which bandwidth is shared by concurrent communications; and c) results for randomly generated graphs, as well as for real-world applications.

The remainder of the paper is organised as follows. Section II describes the system model, including the application model, the utility computing model, and the objective function. Section III discusses related work on budget-constrained workflow scheduling. The proposed scheduling algorithm (HBCS) is presented in Section IV. Experimental details and simulation results are presented in Section V. Finally, Section VI concludes the paper.

6.2 Problem Definition and System Model

A typical workflow application can be represented by a Directed Acyclic Graph (DAG), which is a directed graph with no cycles. In a DAG, an individual task and its dependencies are represented by a node and its edges, respectively. A dependency ensures that a child node cannot be executed before all of its parent tasks finish successfully and transfer the input data required by the child. The task computation times and communication times are modelled by assigning weights to nodes and edges respectively. A DAG can be modelled by a tuple $G(N,E)$, where N is the set of n nodes, each node $n_i \in N$ represents an application task, and E is the set of communication edges between tasks. Each edge $e(i,j) \in E$ represents a task-dependency constraint such that task n_i should complete its execution before task n_j can start.

In a given DAG, a task with no predecessors is called an *entry task*, and a task with no successors is called an *exit task*. We assume that the DAG has exactly one entry task n_{entry} and one exit task n_{exit}. If a DAG has multiple entry or exit tasks, a dummy entry or exit task with zero weight and zero communication edges is added to the graph.

The target utility computing platform is composed of a set of clusters; each cluster has homogeneous processors that have a given capability and cost to execute tasks of a given application. The collection of clusters forms a heterogeneous system. Processors are priced, with the most powerful processor having the highest cost. To normalise diverse price units for the heterogeneous processors, as defined in [ZS12], the price of a processor p_j is assumed to be $Price(p_j) = \alpha_{p_j}(1+\alpha_{p_j})/2$, where α_{p_j} is the ratio of p_j processing capacity to that of the fastest processor. The price will be in the range of $]0\ldots1]$, where the fastest processors, with the highest power, have a price value equal to 1.

For each task n_i, $w_{i,j}$ gives the estimated time to execute task n_i on processor p_j, and $Cost(n_i,p_j) = w_{i,j}.Price(p_j)$ represents the cost of executing task n_i on processor p_j. After assigning a specific processor to execute the task n_i, $AC(n_i)$ is defined as *Assigned Cost* of task n_i. The overall cost for executing an application is defined as $TotalCost = \sum_{n_i \in N} AC(n_i)$.

The edges of the DAG represent a communication cost in terms of time, but they are considered to have zero monetary cost because they occur inside a given site. The schedule length of a DAG, also called *Makespan*, denotes the finish time of the last task in the scheduled DAG, and is defined by:

$$makespan = max\{AFT(n_{exit})\} \tag{1}$$

where $AFT(n_{exit})$ denotes the *Actual Finish Time* of the exit node. In cases in which there is more than one exit node, and no redundant node is added, the *makespan* is the maximum actual finish time of all of the exit tasks.

The *objective* of the scheduling problem is to determine an assignment of tasks of a given DAG to processors such that the *Makespan* is minimised, subject to the budget limitation imposed

by the user, as expressed in equation 2:

$$\sum_{n_i \in N} AC(n_i) \leq BUDGET \tag{2}$$

The user specifies the budget within the range provided by the system, as shown by equation 3:

$$BUDGET = Cheapest_{Cost} + k_{Budget}\left(Highest_{Cost} - Cheapest_{Cost}\right) \tag{3}$$

where $Highest_{Cost}$ and $Cheapest_{Cost}$ are the costs of the assignments produced by an algorithm that guarantees the minimum processing time, such as HEFT [THW02], and the least expensive scheduling, respectively. The least expensive assignment is obtained by selecting the least expensive processor to execute each of the workflow tasks. The algorithm HEFT is used here as one algorithm that produces the minimum *Makespans* for a DAG in a heterogeneous system with complexity $O(v^2.p)$ [CJSZ08]. Therefore, the lower bound of the execution cost is the minimum cost that can be achieved in the target platform, obtained by the cheapest assignment; the upper bound is the cost of the schedule that produces the minimum Makespan. Finally, the budget range feasible on the selected platform is presented to the user; he/she selects a budget inside that range, represented by k_{Budget} in the range of $[0 \ldots 1]$. This budget definition was first introduced by [SZTD07]. The least expensive assignment guarantees that it is always feasible to obtain valid mapping within the user budget, although without guaranteeing the minimisation of the makespan. If the user can afford to pay the highest cost or is limited to the least expensive cost, then the schedule is defined by the HEFT or by the least expensive assignment, respectively. Between these limits, the algorithm we propose here can be used to produce the assignment.

In conclusion, the scheduling problem described in this paper is a single objective function, in which only processing time is optimised, and cost is a problem constraint, the value of which must be guaranteed by the scheduler. This feature is very relevant for users within the context of the utility computing model, and it is a distinguishing feature compared to other algorithms that optimise cost without guaranteeing a user-defined upper bound, as described in the next section.

6.3 Related Work

Generally, the related research in this area can be classified into two main categories: *QoS optimisation* scheduling and *QoS constrained* scheduling. In the first category, the algorithm must find a schedule map that optimises all of the QoS parameters to provide a suitable balance between them for time and cost, as in [DÖ05, SKD07, SLH+13, SK12]. In the second category, the algorithm makes a scheduling decision to optimise for some QoS parameter while subjected to some user-specified constraint values. For example, considering budget and makespan as the QoS parameters, an algorithm in the first category attempts to find a task-to-processor map that best balances between budget and makespan, while an algorithm of the second category takes user-defined values for the budget as an upper bound and defines a mapping that optimises the

makespan. The first category of algorithms can produce schedules with shorter makespans, but the costs of which cannot be limited by the user when submitting the work. Next, we present a review of the second class of algorithms.

The Hybrid Cloud Optimised Cost scheduling algorithm (HCOC), proposed in [BM11b], and a cost-based workflow scheduling algorithm called Deadline-MDP (Markov Decision Process), proposed in [YBT05a], address the problem of minimising cost while constrained by a deadline. Although these models could have applicability in a utility computing paradigm, we do not consider such a paradigm in this paper.

An Ant Colony Optimisation (ACO) algorithm to schedule large-scale workflows with QoS parameters was proposed by [CZ09]. Reliability, time, and cost are three different QoS parameters that are considered in the algorithm. Users are allowed to define QoS constraints to guarantee the quality of the schedule. In [PW10, WPPF08], the Dynamic Constraint Algorithm (DCA) was proposed as an extension of the Multiple-Choice Knapsack Problem (MCKP), to optimise two independent generic criteria for workflows, e.g., execution time and cost. In [YB06a], a budget constraint workflow scheduling approach was proposed that used genetic algorithms to optimise workflow execution time while meeting the user's budget. This solution was extended in [YB06b] by introducing a genetic algorithm approach for constraint-based, two-criteria scheduling (deadline and budget). In [BKK+11], the Balanced Time Scheduling (BTS) algorithm was proposed, which estimates the minimum resource capacity needed to execute a workflow by a given deadline. The algorithm has some limitations, such as homogeneity in resource type and a fixed number of computing hosts.

All of the previous algorithms apply guided random searches or local search techniques, which require significantly higher planning costs and thus are naturally time-consuming. Next, we consider algorithms that were proposed for contexts similar to that considered here, which are heuristic-based and have lower time complexity than the algorithms referred to above and which are used in the Results section for comparison purposes.

In [SZTD07] LOSS and GAIN algorithms were proposed to construct a schedule optimising time and constraining cost. Both algorithms use initial assignments made by other heuristic algorithms to meet the time optimisation objective; a reassignment strategy is then implemented to reduce cost and meet the second objective, the user's budget. In the reassignment step, LOSS attempts to reduce the cost, and GAIN attempts to achieve a lower makespan while attending to the user's budget limitations. In the initial assignment, LOSS has lower makespans with higher costs, and GAIN has higher makespans with lower costs. The authors proposed three versions of LOSS and GAIN that differ in the calculation of the tasks' weights. The LOSS algorithms obtained better performance than the GAIN algorithms, and among the three different types of LOSS strategy, we used LOSS1 to compare to our proposed algorithm. All of the versions of the LOSS and GAIN algorithms use a search-based strategy for reassignments; to obtain their goals, the number of iterations needed tends to be high for lower budgets in LOSS strategies and for higher budgets in GAIN strategies.

The algorithms LOSS and GAIN differ from our approach because they start with a schedule,

and then changes are made iteratively to the schedule until the user budget is guaranteed. We do not consider any initial schedule, and in contrast to those algorithms, ours is not iterative; the time to produce a schedule is constant for a given workflow and platform.

A budget-constrained scheduling heuristic called *greedy time-cost distribution* (GreedyTime-CD) was proposed by [YRB09]. The algorithm distributes the overall user-defined budget to the tasks, based on the estimated tasks' average execution costs. The actual costs of allocated tasks and their planned costs are also computed successively at runtime. This is a different approach, which optimises task scheduling individually. First, a maximum allowed budget is specified for each task, and a processor is then selected that minimises time within the task budget.

In [ZS12, ZS13] the *Budget-constrained Heterogeneous Earliest Finish Time* (BHEFT) was proposed, which is an extension of the HEFT algorithm [THW02]. The context of execution is an environment of multiple and heterogeneous service providers; BHEFT defines a suitable plan by minimising the makespan so that the user's budget and deadline constraints are met, while accounting for the load on each provider. An adequate solution is one that satisfies both constraints (i.e., budget and deadline); if no plan can be defined, it is considered a mapping failure. Therefore, the metric used by the authors was the planning success rate: the percentage of problems for which a plan was found. The BHEFT approach consists of minimising the execution time of a workflow (HEFT based) but within the budget constraints. Our approach is also one of minimising execution time while being constrained to a user-defined budget; therefore, we compare our proposed algorithm to BHEFT in terms of execution time versus budget. Additionally, we compare them in terms of plan success rate, where a deadline is specified for that purpose, similar to [ZS12, ZS13].

Our algorithm differs from BHEFT in two important aspects: first, we allow more processors to be considered as affordable and, therefore, selected; and second, we do not necessarily select the processor that guarantees the earliest finish time, as BHEFT does. Instead, we compute a worthiness value, proposed in this paper, which combines the time and cost factors to decide on the processor for the current task.

GreedyTimeCD and BHEFT have time complexity of $O(v^2.p)$ for a workflow of v nodes and a platform with p processors. Next, we present our proposed scheduling algorithm, which minimises processing time while constrained to a user-defined budget and with low time complexity and which obtains better schedules then other state-of-the-art algorithms, namely LOSS1, GreedyTimeCD, and BHEFT.

6.4 Proposed Budget Constrained Scheduling Algorithm

In this section, we present the Heterogeneous Budget Constrained Scheduling (HBCS) algorithm, which minimises execution time while constrained to a user-defined budget. The algorithm starts by computing two schedules for the DAG: a schedule that corresponds to the minimum execution time that the scheduler can offer (e.g., produced with HEFT) and the highest cost; and another schedule that corresponds to the least expensive schedule cost on the target machines ($Cheapest_{Cost}$), as explained in section 6.2. With the least expensive assignment, the user knows

the minimum cost and corresponding deadline to execute the job; with the highest cost assignment, the user knows the minimum deadline that can be expected for the job and the maximum cost that should be spent to run the job. With this information, the user is able to verify whether the platform can execute the job before the required deadline and within the associated cost range. If these parameters satisfy the user's expectations, he/she specifies the required budget according to equation 3. HBCS is shown in Algorithm 1.

Like most list-based algorithms [KA99], HBCS consists of two phases, namely a *task selection* phase and a *processor selection* phase.

6.4.1 Task Selection

Tasks are selected according to their priorities. To assign a priority to a task in the DAG, the upward rank ($rank_u$) [THW02] is computed. This rank represents the length of the longest path from task n_i to the exit node, including the computational time of n_i, and it is given by equation 10:

$$rank_u(n_i) = \overline{w_i} + \max_{n_j \in succ(n_i)} \{\overline{c_{i,j}} + rank_u(n_j)\} \tag{4}$$

where $\overline{w_i}$ is the average execution time of task n_i over all of the machines, $\overline{c_{i,j}}$ is the average communication time of an edge $e(i,j)$ that connects task n_i to n_j over all types of network links in the network, and $succ(n_i)$ is the set of immediate successor tasks to task n_i. To prioritise tasks, it is common to consider average values because they must be assigned a priority before the location where they will run is known.

6.4.2 Processor Selection

The processor selection phase is guided by the following quantities related to cost. We define the Remaining Cheapest Budget (RCB) as the remaining $Cheapest_{Cost}$ for unscheduled tasks, excluding the current task, and the Remaining Budget (RB) as the actual remaining budget. RCB is updated at each step before executing the processor selection block for the current task, using equation 6 (line 9), which represents the lowest possible cost of the remaining tasks:

$$RCB = RCB - Cost_{lowest} \tag{5}$$

where $Cost_{lowest}$ is the lowest cost for the current task among all of the processors. The initial value for the Remaining Budget is the user budget ($RB = BUDGET$), and it is updated with equation 14 after the processor selection phase for the current task (line 19). RB represents the budget available for the remaining unscheduled tasks:

$$RB = RB - Cost(n_i, P_{sel}) \tag{6}$$

where P_{sel} is the processor selected to run the current task (line 17 of the algorithm), and $Cost(n_i, P_{sel})$ is the cost of running task n_i on processor P_{sel}.

Algorithm 1 HBCS algorithm

Require: DAG and user defined BUDGET
1: Schedule DAG with HEFT and Cheapest algorithm
2: Set task priorities with $rank_u$
3: **if** $HEFT_{cost} < BUDGET$ **then**
4: **return** Schedule Map assignment by HEFT
5: **end if**
6: $RB = BUDGET$ and $RCB = Cheapest_{Cost}$
7: **while** there is an unscheduled task **do**
8: n_i = the next ready task with highest $rank_u$ value
9: Update the Remaining Cheapest Budget (RCB) as defined in Eq.6
10: **for all** Processor $p_i \in P$ **do**
11: calculate $FT(n_i, p_j)$ and $Cost(n_i, p_j)$
12: **end for**
13: Compute $Cost_{Coeff}$ as defined in Eq.7
14: **for all** Processor $p_i \in P$ **do**
15: calculate $worthiness(n_i, p_i)$ as defined in Eq.10
16: **end for**
17: P_{sel} = Processor p_i with highest $worthiness$ value
18: Assign Task n_i to Processor P_{sel}
19: Update the Remaining Budget (RB) as defined in Eq.14
20: **end while**
21: **return** Schedule Map

The quantity Cost Coefficient ($Cost_{Coeff}$), defined by equation 7, is the ratio between RCB and RB, and it provides a measurement of the least expensive assignment cost relative to the remaining budget available. If $Cost_{Coeff}$ is near one, it means that the available budget only allows for selecting the least expensive processors:

$$Cost_{Coeff} = \frac{RCB}{RB} \tag{7}$$

HBCS minimises execution time. Therefore, the finish time of the current task (n_i) is computed for all of the processors ($FT(n_i, p_j)$) at lines 10 and 11, and they constitute one set of factors for processor selection. The other set of factors consists of the costs of executing the task on each processor, that is $Cost(n_i, p_j)$. The variables p_{best} and p_{worst} are defined as the processors with shortest and longest finish times for the current task, respectively.

Processor selection is based on the combination of two factors: time and cost. Therefore, we define two relative quantities, namely Time rate ($Time_r$) and Cost rate ($Cost_r$), for the current task n_i on each processor $p_j \in P$, shown in equations 8 and 9, respectively:

$$Time_r(n_i, p_j) = \frac{FT_{worst} - FT(n_i, p_j)}{FT_{worst} - FT_{best}} \tag{8}$$

$$Cost_r(n_i, p_j) = \frac{Cost_{best} - Cost(n_i, p_j)}{Cost_{highest} - Cost_{lowest}} \tag{9}$$

where FT_{best} and $Cost_{best}$ are the finish time and cost of the current task on processor p_{best}, respectively. FT_{worst} is the finish time on processor p_{worst}, and $Cost_{highest}$ and $Cost_{lowest}$ are the highest and the lowest cost assignments for the current task among all of the available processors, respectively. $Time_r$ measures how much shorter than the worst finish time (FT_{worst}) the finish time is of the current task on processor p_j. Similarly, $Cost_r$ measures how much less the actual cost on p_j is than the cost on the processor that results in the earliest finish time. Both variables are normalised to their highest ranges.

Finally, to select the processor for the current task n_i, the worthiness value for each processor $p_j \in P$ is computed as shown in equation (10):

$$worthiness(n_i, p_i) = \begin{cases} -\infty & \text{if } Cost(n_i, p_j) > Cost_{best} \\ -\infty & \text{if } Cost(n_i, p_j) > RB - RCB \\ Cost_r(n_i, p_j) \times Cost_{Coeff} & \\ +Time_r(n_i, p_j) & \text{otherwise} \end{cases} \tag{10}$$

The first two statements guarantee that if the cost of task n_i on processor p_j is higher than the cost on the processor that gives the minimum FT and if that cost is higher than the available budget for task n_i, then processor p_j cannot be selected. With these statements, the resulting schedule does not exceed the user budget and is guaranteed to be valid. In the third statement, the worthiness value depends on the available budget and on the time during which a processor can finish the task. If the remaining budget (RB) is high, then $Time_r$ has more influence, and a processor with the greater difference in FT compared to the worst processor will have higher worthiness. In contrast, if the remaining budget is smaller, then the cost factor will increase the worthiness of the processors with lower cost to run task n_i. After testing all of the processors, the one with highest worthiness value is selected, and the remaining budget is updated according to equation 14.

The two phases, task selection and processor selection, are repeated until there are no more tasks remaining to schedule. In terms of time complexity, the HBCS requires the computation of HEFT and the least expensive schedule, which are used as pre-requisites to calculate several variables in the HBCS algorithm; these are $O(v^2.p)$ and $O(v^2.p^*)$, respectively, where p^* is number of cheapest processors. In our platform with heterogeneous clusters of homogeneous processors, there is more than one processor with the cheapest cost. The least expensive strategy attempts to assign each task to the least expensive processor; considering insertion policy to calculate the Earliest Finish Time (EFT) among all of the cheapest processors, the complexity of the cheapest algorithm is $O(v^2.p^*)$. The complexities of the two phases of HBCS are of the same order as the HEFT algorithm: $O(v^2.p)$. In conclusion, the total time is $O(v^2.p + v^2.p^* + v^2.p)$, which results in time complexity of the order $O(v^2.p)$.

6.5 Experimental Results

This section presents performance comparisons of the HBCS algorithm with the LOSS1 [SZTD07], GreedyTimeCD [YRB09], BHEFT [ZS13], and Cheapest scheduling algorithms. We consider synthetic randomly generated and Real Application workflows to evaluate a broader range of loads. The results presented were produced with SimGrid [CLQ08], which is one of the best-known simulators for distributed computing and which allows for a realistic description of the infrastructure parameters.

6.5.1 Workflow Structure

To evaluate the relative performances of the algorithms, both the randomly generated and real-world application workflows were used, namely LIGO and Epigenomics [BCD+08]. The randomly generated workflows were created by the synthetic DAG generation program[1]. The computational complexity of a task was modelled as one of the three following forms, which are representative of many common applications: $a.d$ (e.g., image processing of a $\sqrt{d}.\sqrt{d}$ image), $a.d\log d$ (e.g., sorting an array of d elements), and $d^{3/2}$ (e.g., multiplication of $\sqrt{d}.\sqrt{d}$ matrices), where a is chosen randomly between 2^6 and 2^9. As a result, different tasks exhibit different communication/computation ratios.

The DAG generator program defines the DAG shape based on four parameters: *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; 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 indicates that the levels contain very dissimilar numbers of tasks, whereas a high value indicates that all of the 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 used as the number of tasks $n = [10...60]$, $jump = [1,2,3]$, $regularity = [0.2,0.4,0.8]$, $fat = [0.2,0.4,0.8]$, and $density = [0.2,0.4,0.8]$. With these parameters, each DAG was created by choosing the value for each parameter randomly from the parameter data set. The total number of DAGs generated in our simulation was 1000.

6.5.2 Simulation platform

We resorted to simulation to evaluate the algorithms discussed in the previous sections. Simulation allows us to perform a statistically significant number of experiments for a wide range of application configurations in a reasonable amount of time. We used the SimGrid toolkit [2] [CLQ08]

[1] https://github.com/frs69wq/daggen
[2] http://simgrid.gforge.inria.fr

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 heterogeneous computing system, such as the grid platform considered in this work. In many research papers on scheduling, the authors have assumed a contention-free network model, in which processors can simultaneously send data 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 must share bandwidth. This scheme corresponds well to the behaviour of TCP connections on a LAN. The validity of this network model was demonstrated by [VL09].

We consider two sites that comprise multiple clusters. Table 13 provides the name of each cluster, along with its number of processors, processing speed expressed in GFlop/s, and the standard deviation (σ) of CPU capacity as a heterogeneity factor. Column $\#CPU_{used}$ shows the number of processors used from each cluster for 8-, 16- and 32-processor configurations.

Site	Cluster	$\#CPU_{Total}$	$\#CPU_{used}$			Power(GFlop/s)	σ
	chicon	26	2	4	9	8.9618e9	
lille	chimint	20	2	4	7	23.531e9	0.69
	chinqchint	46	4	8	16	22.270e9	
	helios	56	3	6	12	7.7318e9	
sophia	sol	50	3	5	10	8.9388e9	0.90
	suno	45	2	5	10	23.530e9	

Table 10: Description of the cappello2005grid clusters from which the platforms used in the experiments were derived

6.5.3 Budget-Deadline Constrained Scheduling

Budget-deadline constrained scheduling was introduced in [ZS12, ZS13], in which the BHEFT algorithm was presented. Its aim is to produce mappings of tasks to resources that meet the deadline and budget constraints imposed by the user. Although BHEFT considers the load of the providers, the algorithm can be evaluated by attending exclusively to the deadline and budget constraints and ignoring the providers' loads, as also presented in [ZS12, ZS13]. This process allows for evaluating and comparing the algorithms, without the additional random variables that are the providers' loads. The budget factor is considered here, as defined by equation 3. The deadline constraint is defined by equation 23, which specifies a deadline value for a given workflow, based on the

mapping obtained by HEFT:

$$DeadLine = LB_{deadline} + k_{Deadline}\left(UB_{deadline} - LB_{deadline}\right) \tag{11}$$

where $LB_{deadline}$ is equal to $HEFT_{Makespan}$, the makespan obtained with HEFT, and $UB_{deadline}$ is $3.HEFT_{Makespan}$. The range of values for $k_{Deadline}$ is $[0\ldots1]$. $HEFT_{Makespan}$ is the minimum time that the user is allowed to choose to obtain a valid schedule. We restrict the upper bound deadline as the cube of $HEFT_{Makespan}$ to evince the quality of the algorithms in generating valid mappings.

6.5.4 Performance metrics

To evaluate and compare our algorithm with other approaches, we consider the metric Normalised Makespan (NM), defined by equation 12. NM normalises the makespan of a given workflow to the lower bound, which is the workflow execution time obtained with HEFT:

$$\text{NM} = \frac{\text{schedule makespan}}{HEFT_{Makespan}} \tag{12}$$

To evaluate the algorithms using the budget-deadline constrained methodology, we consider the Planning Success Rate (PSR), as expressed by equation 25 and defined in [ZS12, ZS13]. This metric provides the percentage of valid schedules obtained in a given experiment.

$$\text{PSR} = 100 \times \frac{\text{Successful Planning}}{\text{Total Number in experiment}} \tag{13}$$

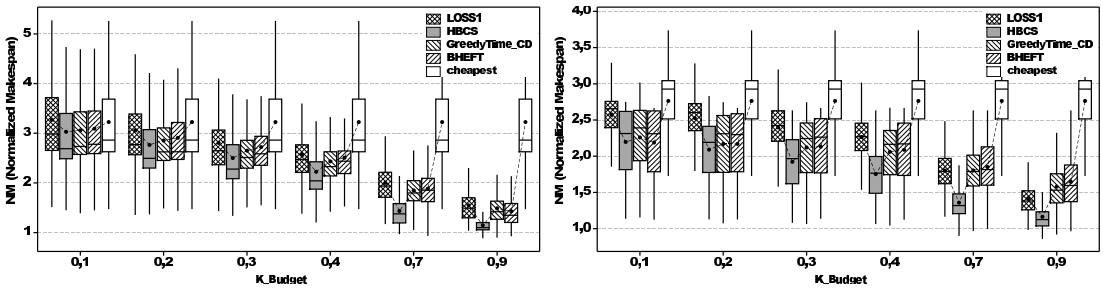
6.5.5 Results and Discussion

Among all of the algorithms mentioned above in the related work section, we selected LOSS1, GreedyTimeCD, BHEFT, and Cheapest scheduling algorithms; these match our goal and conditions, but we have made some modifications in their original strategies. The original implementation of the LOSS scheduling algorithm assumed that all of the processors had different costs, and therefore, there was no conflict in selecting a processor based on the cost parameter. In our grid environment, each cluster was homogeneous and was based only on the cost parameter, so there could be more than one processor candidate. In this case, we tested all of the possible processors and selected that which achieved the smallest makespan. The same procedure was applied to the least expensive scheduling strategy, which attempts to schedule each task on the service with the lower execution cost.

To evaluate the performance of each algorithm, we computed the makespan for each DAG and normalised it using equation 12. The results presented next were obtained by SimGrid with the bounded multi-port model, and they are presented for two data sets: randomly generated workflows and real applications.

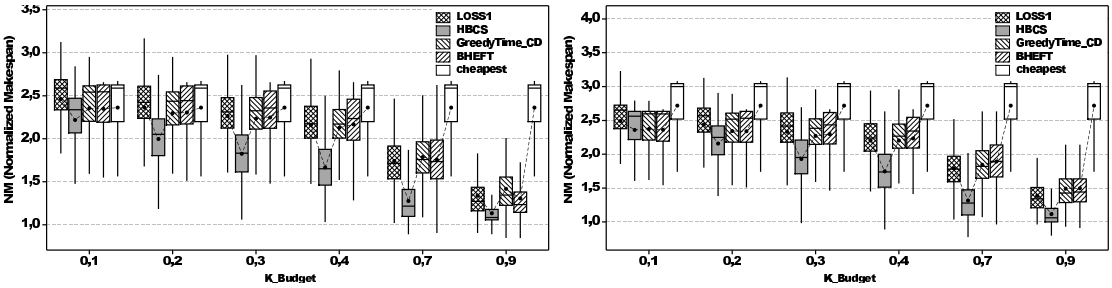
6.5.5.1 Results for Randomly generated workflows

For random DAG generation, as stated previously, we model the computational complexity of common applications tasks, such as image processing, array sorting, and matrix multiplication. Figures 1 and 2 show the Normalised Makespan values obtained on two CPU configurations per site, namely, 8 and 32 processors. We can see that the HBCS algorithm obtains significant performance improvements over the other algorithms for most of the user-defined budget values and configurations.



(a) Cluster$_{name}$ = Lille (b) Cluster$_{name}$ = Sophia

Figure 1: Normalized Makespan for Random workflows with CPU$_{used}$=8



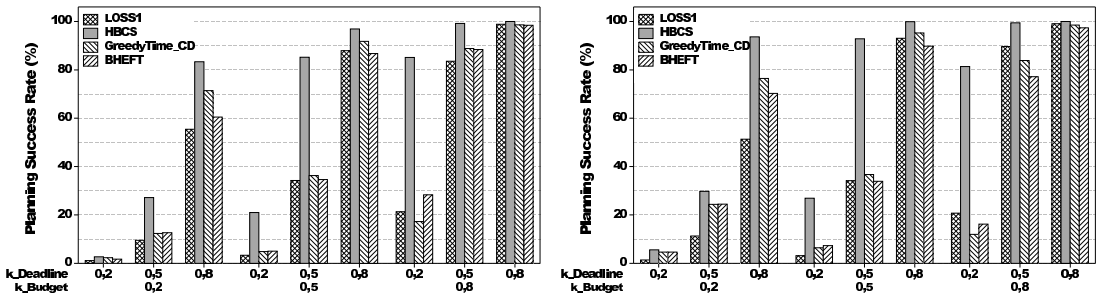
(a) Cluster$_{name}$ = Lille (b) Cluster$_{name}$ = Sophia

Figure 2: Normalized Makespan of scheduling for Random workflows with CPU$_{used}$=32

For 8 processors on the Lille site, the improvement of HBCS over BHEFT is 2.0% for $k = 0.1$, and it increases to 19.7% for $k = 0.9$. On the Sophia site, the improvement is 0% for $k = 0.1$, and it increases to 29.4% for $k = 0.9$. For $k \geq 0.7$, the second-best algorithm is LOSS1, and the improvements of HBCS over LOSS1 are 24.7% and 18.2% for $k = 0.7$ and $k = 0.9$, respectively.

For 32 processors, the same pattern of results is obtained on the Lille and Sophia sites but with greater differences for lower k values. On the Lille site, the improvement of HBCS over BHEFT is 5.4%, 13.4% and 18.8% for k values of 0.1, 0.2, and 0.3, respectively. The performance of BHEFT and GreedyTimeCD are very similar for all of the ranges of k on both sites. For $k \geq 0.7$, the second best algorithm is again LOSS1.

In conclusion, HBCS outperforms all of the other algorithms, indicating that HBCS can achieve lower makespans for any given budget.



(a) Cluster$_{name}$ = Lille (b) Cluster$_{name}$ = Sophia

Figure 3: Planning Success Rate for Random workflows

The results of the evaluation made with the budget-deadline constrained methodology are shown in Figure 2, where the deadline and budget factors assume the values of 0.2, 0.5, and 0.8. The figures present the average values for 8, 16 and 32 processors for each site. HBCS yielded higher PSR values for all of the cases. The PSR obtained by HBCS for shorter deadline factors is significantly higher than the PSR values obtained with the other algorithms. On both sites, for deadline and budget factors of 0.2 and 0.8, respectively, the HBCS PSR is greater than 80%, while for the remaining algorithms, the PSR is less than 30%. We can conclude that HBCS obtains the highest Planning Success Rates for the two levels of heterogeneity considered here.

6.5.5.2 Results for Real World Applications

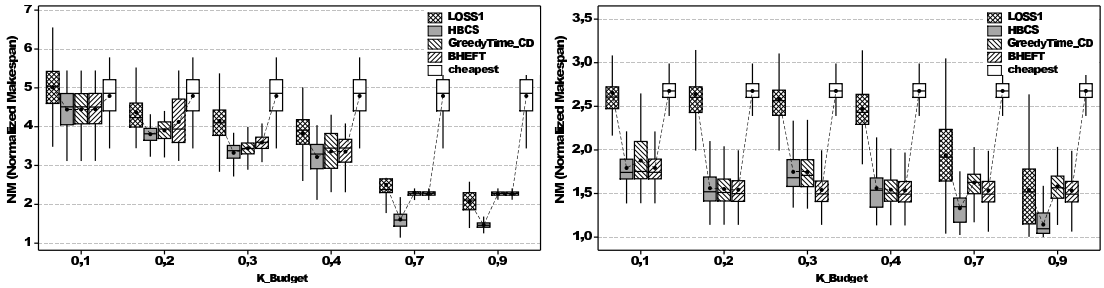
To evaluate the algorithms on a standard and real set of workflow applications, synthetic workflows were generated using the code developed in [Peg13]. Two well-known structures were chosen [JCD+13], namely Epigenomics[3], for mapping the epigenetic state of human DNA, and LIGO [4]. The Epigenomics workflow is a highly pipelined application with multiple pipelines operating on independent chunks of data in parallel. In these two real applications, most of the jobs have high CPU utilisation and relatively low I/O, so they can be classified as CPU-bound workflows.

Workflows with 24 and 30 tasks were created for both Epigenomics and LIGO. For each workflow size, 300 different workflow instances were generated, obtaining a total collection of 600 workflows for each type of application.

For Normalised Makespan, Figures 4 and 5 show that HBCS achieves better performance for most of the budget factor values. The improvements increase with the budget factor and the site heterogeneity. Similar behaviour is obtained for the LIGO application.

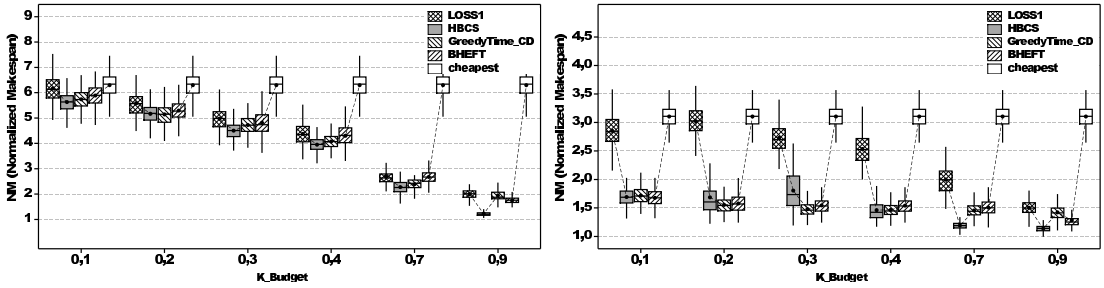
Figures 6 and 7 show the Planning Success Rate for the Epigenomics and LIGO workflows. For each site, the average PSRs for 8, 16, and 32 CPUs are presented. These results are consistent with those obtained for the randomly generated workflows. It is also observed that for some cases, particularly for Epigenomics, only HBCS obtained valid schedules and significant success rates.

[3] USC Epigenome Center, http://epigenome.usc.edu
[4] http://www.advancedligo.mit.edu



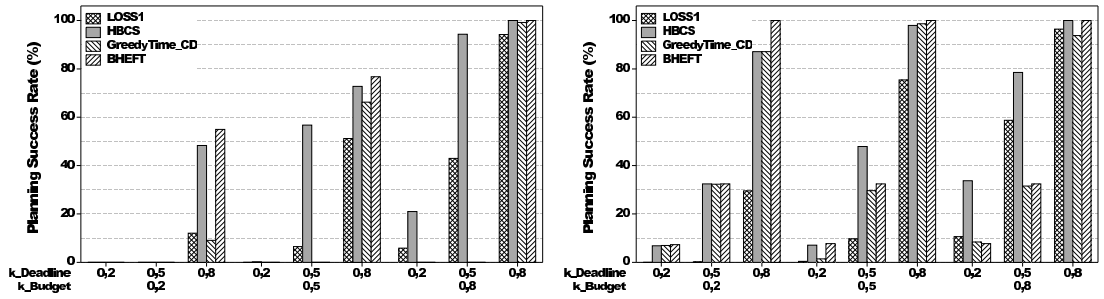
(a) Cluster$_{name}$ = Lille , CPU$_{used}$=8 (b) Cluster$_{name}$ = Sophia , CPU$_{used}$=8

Figure 4: Normalized Makespan for Epigenomics workflows on cappello2005grid



(a) Cluster$_{name}$ = Lille , CPU$_{used}$=8 (b) Cluster$_{name}$ = Sophia , CPU$_{used}$=8

Figure 5: Normalized Makespan for LIGO workflows on cappello2005grid

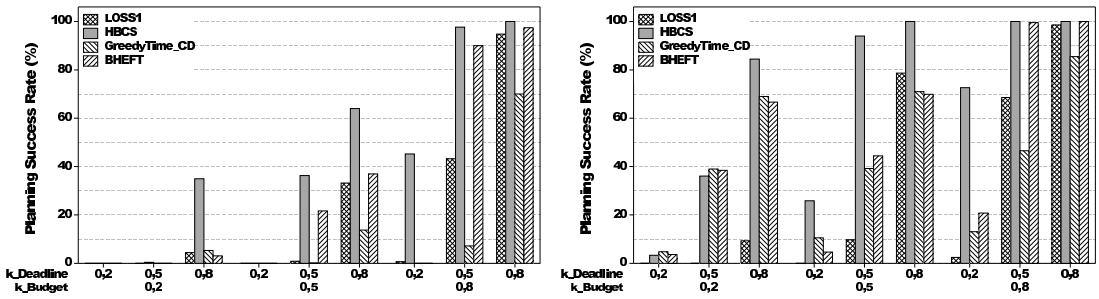


(a) Cluster$_{name}$ = Lille (b) Cluster$_{name}$ = Sophia

Figure 6: Planning Success Rate for Epigenomics workflows

6.6 Conclusions and Future Work

In this paper, we have presented the Heterogeneous Budget Constrained Scheduling algorithm, which maps a workflow to a heterogeneous system and minimises the execution time constrained to a user-defined budget. The algorithm was compared with other state-of-the-art algorithms and was shown to achieve lower makespans for all of the budget factors in higher heterogeneity platforms; that is, HBCS can produce shorter makespans for the same budget. A reduction of up to 30% in execution time was achieved while maintaining the same budget level. Considering Budget-Deadline constrained scheduling, HBCS achieves a higher planning success rate for more



(a) Cluster$_{name}$ = Lille (b) Cluster$_{name}$ = Sophia

Figure 7: Planning Success Rate for LIGO workflows

heterogeneous systems. In some particular cases, the PSR obtained with HBCS is considerably higher than the PSRs obtained with the other algorithms. For example, for the budget and deadline factors of 0.8 and 0.2, respectively, on the Lille and Sophia sites, the PSR of HBCS is greater than 80%, while all of the other algorithms achieve PSRs less than 30%. On these sites, HBCS performs significantly better than the other algorithms for the more restricted deadline factors of 0.2 and 0.5.

The results achieved for two real applications, namely LIGO and Epigenomics, are consistent with the randomly generated workflows for both metrics.

Concerning time complexity, HBCS has the same complexity as GreedyTimeCD and BHEFT, having a constant running time for all ranges of budget factors for a given workflow and platform, because it has bounded complexity. In contrast, the LOSS algorithms have a search-based step, and their running time therefore depends on the number of iterations to find a solution, where for lower budgets, the number of iterations is significantly greater.

In conclusion, we have presented the HBCS algorithm for budget constrained scheduling, which has proved to achieve better performance with lower time complexity than other state-of-the-art algorithms. The results were obtained in a simulation with a realistic model of the computing platform and with shared links, as occurs in a common grid infrastructure.

In future work, we intend to extend the algorithm to consider the dynamic concurrent DAG scheduling problem. This consideration will allow users to execute concurrent workflows that might not be able to start together but that can share resources so that the total cost for the user can be minimised.

6.7 acknowledgements

This work was supported in part by the Fundação para a Ciência e Tecnologia, PhD Grant FCT - DFRH - SFRH/BD/80061/2011.

Chapter 7

Budget Constrained Scheduling Strategies for On-line Workflow Applications

Hamid Arabnejad and Jorge G. Barbosa

14th International Conference on Computational Science and Its Applications (ICCSA),

Volume 8584 of Springer LNCS, pages 532–545, 2014,

DOI: 10.1007/978-3-319-09153-2_40,

abstract

To execute scientific applications, described by workflows, whose tasks have different execution requirements, efficient scheduling methods are essential for task matching (machine assignment) and scheduling (ordered for execution) on a variety of machines provided by a heterogeneous computing system. Several algorithms for concurrent workflow scheduling have been proposed, being most of them off-line solutions. Recent research attempted to propose on-line strategies for concurrent workflows but only address fairness in resource sharing among applications while minimizing the execution time. In this paper, we propose a new strategy that extends on-line methods by optimizing execution time constrained to the user budget. Experimental results show a significant improvement of the produced schedules when our strategy is applied.

7.1 Introduction

With the raising in usage of high-speed internet, grid infrastructures are becoming a novel design structure of distributed computing. The Utility Computing model to deploy those systems has become more popular and widely used to provide a price regulated high performance computing to the users. In the utility model, users are allowed to submit their jobs to different resources, based on the computational cost and jobs deadline. Resources are shared so that the provider can optimize the running costs, but the provider has also to guarantee a high Quality of Service (QoS) by ensuring a fairness access to the shared resources to all users.

Scientific jobs are commonly represented as workflow applications that consist of many tasks, with logical or data dependencies, that can be dispatched to different compute nodes. When scheduling multiple independent workflows that represent user jobs and are thus submitted at different moments in time, a dynamic behaviour is required to redistribute the workload. Most concurrent workflow scheduling algorithms proposed are for static, or off-line, conditions such as [ZS06, NS09, BM10a]. However, there are some proposed methods which address the problem of scheduling on-line multiple workflows, namely OWM [HHW11], RANK_HYBD [YS08] and FDWS [AB12a], which target is to minimize the average relative waiting time of the jobs. In this context the common definition of makespan is extended to account for the waiting time and execution time of a given workflow [ABS14].

In the utility model, users consume the services and resources when they need to, and pay only for what they use. Cost and time become the two most important factors that users are concerned about. Thus, the cost/time tradeoff problem for scheduling workflow applications becomes a challenging problem. In this paper we propose a scheduling strategy that extend the former concurrent on-line scheduling algorithms by considering fairness resource sharing constrained to the user defined budget, for each job.

The remainder of this paper is organized as follows: section II describes the scheduling system model; section III reviews related works; section IV presents the proposed algorithm; section V presents the experimental results and discussion; and finally, section VI concludes the paper.

7.2 Scheduling System Model

The proposed scheduling system model consists of an application model, a utility grid model, budget model and a performance criterion for scheduling.

A typical workflow application can be represented by a Directed Acyclic Graph (DAG), a directed graph with no cycles. In a DAG, an individual task and its dependency is represented by a node and its edges. A dependency ensures that a child node cannot be executed before all its parent tasks finish successfully and transfer the required child input data. The task computation time and communication time is modelled by assigning weight to nodes and edges respectively. A DAG can be modeled by a tuple $G(V,E)$ where V is the set of v nodes and each node $v_i \in V$ represents an application task, and E is the set of communication edges between tasks. Each edge $e(i,j) \in E$

represents the task-dependency constraint such that task v_i should complete its execution before task v_j can be started. In a given DAG, a task with no predecessors is called an *entry task* and a task with no successors is called an *exit task*. We assume that the DAG has exactly one entry task n_{entry} and one exit task n_{exit}. If a DAG has multiple entry or exit tasks, a dummy entry or exit task with zero weight and zero communication edges is added to the graph.

The service-oriented architecture for Utility Grids is shown in Figure 1 [YVB06]. Each user should submit its applications with user QoS requirements into the system. Then, all information about each application is collected into the application Data Base (DB). A utility grid model consists of several Grid Service Providers (GSPs), each of which provides some services to the users. GSPs charge different services by their QoS. Users only execute their jobs in GSPs that satisfy their QoS requirements, and only pay for what they use. To inform and attract users, each GSP should publish their services into Grid market directory (GMD). Ready Tasks Pool component collects tasks which are ready to execute among accepted workflow applications in application DB. A task is ready when all required information are prepared, i.e., its parents are executed. Then, Grid scheduler enquires GMD to query about available services for each task and their QoS attributes. Also contacts the Grid Service Information to gather detailed information of each service, especially the available time slots for processing tasks. Using these information, the Grid scheduler executes a Task Scheduler algorithm to decide the map allocation based on user QoS parameters for each ready task of a workflow to one of the available services. The Service Executer is used to monitor task assignment on the service and to get notifications of successfully or failure of task execution on GSPs resources.

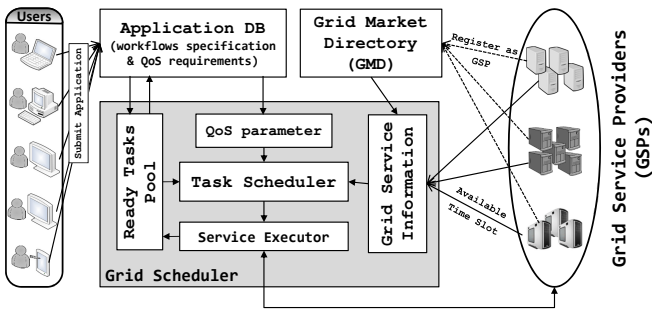


Figure 1: A General View of Grid Scheduler System

After collecting the job and platform information, the budget model is the third parameter in our model. The available user defined budget for a workflow application must never be exceeded. Generally, the cost model can be divided into two main categories: (1) charge computing resources on hourly or monthly usage (time interval), and any partial hours are rounded up such as Amazon Elastic Compute Cloud (Amazon EC2) [Ama], (2) charges based on the number of CPU cycles required by a customer's application such as Google AppEngine [Goo]. In this paper, we consider the second category of pay-as-you-go pricing model.

The general objective is to minimize the completion time, also called *Makespan*. It is denoted by the finish time of the last task of the scheduled workflow application. But, when schedul-

ing various independent workflows that represent user jobs and therefore are 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.

Therefore, our performance criteria to assign tasks to processors, from all workflows, considers the minimization of the turnaround time of each workflow, subjected to the user budget limitation imposed to each workflow.

7.3 Related Work

Generally, the related research in this field can be classified into two main categories: off-line and on-line scheduling. 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 method, although restricted, can be applied in many real-world applications, e.g. when a user has a set of nodes to run a set of workflows. On-line scheduling exhibits dynamic behavior in which users can submit the workflows whenever they need to.

There are some algorithms proposed specifically to schedule concurrent workflows to improve individual QoS, namely, RANK_HYBD [YS08] and Fairness Dynamic Workflow Scheduling (FDWS) [AB12a]. The first algorithm minimize 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 [ABS14], it is presented a review of on-line and off-line scheduling algorithms, for concurrent workflow scheduling, and a performance comparison for on-line scheduling.

In [YS08], Yu and Shi proposed a planner-guided strategy, the RANK_HYBD algorithm to address dynamic scheduling of workflow applications that are submitted by different users at different moments in time. The RANK_HYBD algorithm, collects all new arrival workflows submitted by users, and ranks all tasks using the $rank_u$ priority measure [THW02], which represents the length of the longest path from task n_i to the exit node, including the computational cost of n_i, and it is expressed as follows:

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

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 $rank_u(n_{exit}) = 0$. In each step, the RANK_HYBD algorithm fills the ready tasks pool by all ready tasks from each submitted workflows and selects the next task to schedule based on their rank. If all tasks in the ready tasks pool belong to different workflows, the algorithm selects the task with lowest rank and if they belong to the same workflow, the task with highest rank is selected. Finally, among of all free processor, the processor with lowest finish time for selected task is chosen. The RANK_HYBD algorithm has some weakness points in its strategy. It allows the workflow with

the lowest rank (lower makespan) to be scheduled first to reduce the waiting time of the workflow 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.

To overcome these problems with the RANK_HYBD algorithm, the FDWS algorithm was proposed in [AB12a]. FDWS applied new strategies for selecting the tasks from the ready tasks pool 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, unlike the RANK_HYBD algorithm, adds only a single ready task with highest priority ($rank_u$) from each workflow to the ready pool. Therefore, instead of scheduling smaller workflows first like RANK_HYBD, it selects and schedules tasks from the longer workflows as well and all workflows have chance to be scheduled in each step. In the Task Selection phase, FDWS assigns a secondary priority to each task in ready tasks pool. To be inserted into the ready tasks pool, the $rank_u$ was computed for each workflow individually. After filling the ready tasks pool by one ready task from each workflow, $rank_r$ is computed as the secondary priority rank for task n_i belonging to DAG_j, defined by (2). Then, the task with highest $rank_r$ from the ready pool is selected.

$$rank_r(n_{i,j}) = \frac{1}{PRT(DAG_j)} \times \frac{1}{CPL(DAG_j)}. \tag{2}$$

The $rank_r$ metric considers the Percentage of Remaining Tasks value (PRT) of the workflow (DAG) and its Critical Path Length (CPL). The PRT value gives more priority to workflows that are almost completed and only have few tasks to execute. Unlike RANK_HYBD algorithm which uses only the individual $rank_u$ for selecting tasks into ready tasks pool and for execution in each round, in FDWS, the workflow history in the workflow application DB pool is considered to make a scheduling decision.

An alternative strategy is the resource reservation policy, where a static scheduling decision based on resource availability is made when an application workflow is submitted into the system. But this strategy lead us to unfair scheduling because if a workflow application with higher number of tasks and execution time is submitted, it may reserve most of the resources and make higher waiting time for next applications.

But none of these approaches, consider cost as a QoS parameter in their scheduling strategies. In this paper, we propose a strategy for on-line scheduling of concurrent workflows with budget constraints defined by users for each workflow, as described in the next section. We apply our strategy to RANK_HYBD and FDWS in order to consider the budget constraint.

7.4 The Proposed Strategy

In our model the user specifies the budget constraint by selecting a budget value in the interval limited by the cheapest and highest costs. For the job j the budget is specified as expressed by

equation 3:

$$BUDGET(j) = \begin{array}{l} Cost_{lowest}(j) + k_{Budget}(j) \times \\ \left(Cost_{highest}(j) - Cost_{lowest}(j)\right) \end{array} \tag{3}$$

where $Cost_{highest}(j)$ and $Cost_{lowest}(j)$ are the total cost of the assignment produced by the highest and cheapest scheduling, that is obtained by selecting the most expensive and most cheapest processors to execute each task of workflow j. Therefore, for each workflow, it is presented to the user the budget range that is possible to achieve in the selected platform and he/she selects a budget inside that range, that is represented by k_{Budget}. This budget definition was first introduced in [SZTD07].

The workflow management system represented in Figure 1 gathers applications information submitted by users, scheduling and runtime monitoring. Next we describe how tasks are selected to the Ready Task Pool and the scheduling strategy.

7.4.1 Ready Tasks Pool

There is a *ready tasks pool* which is filled by the ready tasks belonging to each submitted and unfinished workflow at each scheduling round. In general, two methods are used to fill the *ready tasks pool*, first, like FDWS algorithm, gather only a single ready task with highest priority ($rank_u$) from each workflow, or insert all ready tasks belonging to each unfinished workflow application into *ready tasks* pool such as RANK_HYBD. But the important key is how to order these ready tasks, i.e., how to assign priority to each ready task based on our QoS parameters to have higher quality solutions and system performance. For selecting a task from *ready tasks pool* to be schedule on resources, we defined a new strategy to assign a secondary priority to each task of the *ready tasks pool*. Because our goal is to execute applications in the lowest turnaround time with its limited budget, the cost factor should be taken into account. To achieve this purpose, for each workflow j, we define the Task Proportion (TP_j), equation 4, which is the ratio of unscheduled number of tasks to the total number of tasks in the workflow (DAG), and Budget Proportion (BP_j), equation 5, which is the ratio of the Remaining Cheapest Budget (RCB_j) to the Remain Budget (RB_j). RCB_j is the remaining $lowest_{Cost}$ for unscheduled tasks and RB_j is the actual remaining budget.

$$TP_j = \frac{unscheduled\ number\ of\ tasks}{Total\ number\ of\ tasks} \tag{4}$$

$$BP_j = \frac{RCB_j}{RB_j} \tag{5}$$

RCB_j is updated in each step after making the processor selection for the selected task belonging to workflow(j) by using equation 6.

$$RCB_j = RCB_j - lowest_{Cost}(T_{sel}) \tag{6}$$

where $lowest_{Cost}(T_{sel})$ is the lowest cost for current task (selected task for scheduling). The initial value for Remaining Cheapest Budget is $RCB_j = lowest_{Cost}$ where $lowest_{Cost}$ is the total cost of the assignment for workflow j produced by the cheapest scheduling. In addition, the initial value for Remain Budget is $RB_j = BUDGET_j$ where $BUDGET$ is user defined budget for application workflow, and it will be updated, by equation 14, after the processor selection phase for the selected task.

$$RB_j = RB_j - Cost(T_{sel}, P_{sel}) \tag{7}$$

where P_{sel} is the processor selected to run the current task (T_{sel}), and $Cost(T_{sel}, P_{sel})$ is the cost of running task T_{sel} on processor P_{sel}.

To assign the secondary priority to each task in *ready tasks pool*, we propose $rank_B$ for each task t_i in the pool, that belongs to workflow j, defined by equation 8. The task with highest $rank_B$ is selected to be schedule.

$$rank_B(t_{i,j}) = \frac{1}{TP_j} \times \frac{1}{BP_j} \tag{8}$$

The $rank_B$ value is the product of two factors: a) the first one is the inverse of the fraction of the workflow j that is remaining in the system; and b) the ratio of the budget value over the remaining cheapest budget. This priority factor gives higher priority to the workflows that have a lower percentage of tasks unscheduled and to workflows that have higher budgets when compared to the cheapest budget for the DAG. The rational for the first factor is to give higher priority to workflows that where submitted earlier, so that a longer workflow with several tasks already executed may have priority over a short and recent workflow. And the rational of the budget factor is that the scheduler will consider first tasks that can spend more budget and therefore they will select more expensive and faster processors, resulting in a lower turnaround time for the workflow.

7.4.2 Task scheduler

The Task scheduler is another important key in workflow management system which has responsibility for selecting affordable resource for the current selected task. In this part, we propose a new strategy for processor selection phase based on QoS requirements. The processor to be selected to execute the current task is guided by the following strategy related to cost. To achieve minimum execution time under limited budget, first, the task with highest $rank_B$ is selected from *ready tasks pool*, then the finish time of selected task (t_{sel}) is computed for all processors ($FT(t_{sel}, p_i)$) and these values are one of the factors for processor selection. The variables $FT_{min}(t_{sel})$ and $FT_{max}(t_{sel})$ are defined as lowest and highest finish time for selected task, respectively. The other factor is the cost of executing the task on each processor $Cost(t_{sel}, p_i)$.

The new processor selection strategy is based on the combination of the two factors, time and cost, and therefore two relative quantities are defined, namely, Time quota ($Time_q$) and Cost quota ($Cost_q$) for selected task (t_{sel}) on each processor $p_i \in P$, shown in equations 9 and 10, respectively.

$$Time_q(t_{sel}, p_i) = \frac{FT_{max}(t_{sel}) - FT(t_{sel}, p_i)}{FT_{max}(t_{sel}) - FT_{min}(t_{sel})} \tag{9}$$

$$Cost_q(t_{sel}, p_i) = \frac{Cost_{best}(t_{sel}) - Cost(t_{sel}, p_i)}{Cost_{highest}(t_{sel}) - Cost_{lowest}(t_{sel})} \tag{10}$$

where $Cost_{best}(t_{sel})$ is the cost of selected task on the processor with lowest finish time (processor with $FT_{min}(t_{sel})$). $Time_q$ shows how far is the finish time of selected task on processor p_i, from the worst finish time ($FT_{max}(t_{sel})$). Similarly, $Cost_q$ shows how far the actual cost, on p_i, is from the best cost ($Cost_{best}(t_{sel})$). Both variables are normalized with their highest range.

In addition to these variables that give time and cost relative processor performance, there is a limitation on cost consumption. This constraint is represented by the spare budget, that is defined by the difference between the remaining budget available (RB) and the remaining cheapest assignment (RCB). Once RCB includes the minimum cost of t_{sel}, this quantity is added to the spare budget available, as expressed by equation 11.

$$Cost_{lim}(t_{sel}) = Cost_{lowest}(t_{sel}) + (RB_j - RCB_j) \tag{11}$$

The Budget Proportion (BP), defined by equation 5, is the ratio between RCB and RB, and gives a measure of how far the cheapest assignment is from the remaining budget available. If BP is near to one, it means that the available budget only allows to select the cheapest assignment. In addition, the $Cost_{lim}$ controls the processor decision to avoid cost consumption higher than user defined budget.

Finally, to select the processor for current task t_{sel} belonging to workflow j, it is computed the Quality value (Q) for each available and free processor $p_i \in P$ as shown in equation 12.

$$\underset{p_i \in P}{Q(t_{sel}, p_i)} \begin{cases} -\infty & \text{if } Cost(t_{sel}, p_i) \\ & > Cost_{best}(t_{sel}) \\ -\infty & \text{if } Cost(t_{sel}, p_i) \\ & > Cost_{lim}(t_{sel}) \\ Time_q(t_{sel}, p_i) + & \text{otherwise} \\ BP_j \times Cost_q(t_{sel}, p_i) & \end{cases} \tag{12}$$

The first two statements guaranty that if the cost of task t_{sel} on processor p_i is higher than the cost on the processor that gives the minimum FT, and if that cost is higher than the available budget for task t_{sel}, then processor p_i cannot be selected. Otherwise, the processor is evaluated considering the time and cost quantities. The processor with higher Quality value is selected.

Algorithm 1 shows the general algorithm used to implement the algorithm versions shown in table 11. The characteristics that differentiate each version are the strategy to select ready tasks from each workflow, the priority assigned to each ready task in the task pool and, the processor selection policy. These policies are parameters of the general algorithm.

Algorithm 1 The General Budget Constrained Scheduling Strategies for On-Line Workflow Applications

Require: Input Strategies
1- A filling strategy to add ready tasks from each workflow into *Ready Tasks* pool
2- A priority strategy for assigning a rank to each task into *Ready Tasks* pool
3- A processor selection strategy

while Application DB $\neq \phi$ **do**
- Fill *Ready Tasks* pool based on the input filling strategy
for all $t_i \in$ *Ready Tasks* pool **do**
- Assign a rank value for t_i according to the input priority strategy for ready tasks
end for
while *Ready Tasks* $\neq \phi$ **do**
- Select task T_{sel} with highest priority from *Ready Tasks* pool
- Select best suitable processor based on the input processor selection strategy
- Assign Task T_{sel} to selected Processor
- Update the Remain Budget (*RB*) and the Remain Cheapest Budget (RCB) as defined in Eq.14 and Eq.6
- Remove Task T_{sel} from *Ready Tasks* pool
end while
end while

Algorithm Name	Strategies		
	Filling Ready Pool	Selecting task to schedule	Processor Selection
B-RANK_HYBD1	**for each** *workflow* **Insert** *all ready tasks*	**if all** $t_i \in$ *ready pool* belong to *same workfolow* **then select** t_i with highest $rank_u$, **else select** t_i with lowest $rank_u$	**select** $p_j \in P$ with lowest $FT(t_i, p_j)$ **where** $cost(t_i, p_j) \leq Cost_{lim}(t_i)$
B-RANK_HYBD2			**select** $p_j \in P$ with Highest $Q(t_i, p_j)$ value (Eq.12)
B-FDWS1	**for each** *workflow* **Insert** *Single ready task* **with** highest $rank_u$	**select** $t_i \in$ *ready pool* **with** highest $rank_r$ (Eq.2)	**select** $p_j \in P$ with lowest $FT(t_i, p_j)$ **where** $cost(t_i, p_j) \leq Cost_{lim}(t_i)$
B-FDWS2			**select** $p_j \in P$ with Highest $Q(t_i, p_j)$ value (Eq.12)
B-FDWS3		**select** $t_i \in$ *ready pool* with highest $rank_B$ (Eq.8)	**select** $p_j \in P$ with lowest $FT(t_i, p_j)$ **where** $cost(t_i, p_j) \leq Cost_{lim}(t_i)$
B-FDWS4			**select** $p_j \in P$ with Highest $Q(t_i, p_j)$ value (Eq.12)

Table 11: Description of the modified algorithms for on-line budget constrained scheduling

7.5 Experimental Results and Discussion

This section presents performance results obtained with the new strategies applied to two scheduling algorithms. We implemented modified versions of RANK_HYBD [YS08] and FDWS [AB12a], called Budget RANK_HYBD (B-RANK_HYBD) and Budget FDWS (B-FDWS) to consider budget limitation imposed by users. In the processor selection phase of these two algorithms, cost is not taken into account and there is a possibility to have higher cost than the limited budget defined by the user. So, in modified version, instead of considering all processors to compute the finish time of current task, processors are filtered based on the cost limitation value

defined by Equation 11 and we select the processor that allows the lowest finish time among all affordable processors. To evaluate the influence of the new strategy proposed in this study, $rank_B$ for selecting tasks and quality measure value (Q) for selecting processors, we consider several version of the scheduling algorithms as described in Table(11). For all algorithms, we select processors that are free on current time (no reservation policy).

This section is divided into four parts, namely, workflow structure description, the environment scheduling system and hardware parameters, the performance metric, and finally results and discussion are presented.

7.5.1 Workflow structure

To evaluate the relative performance of the algorithms, a model of execution time of the tasks on resources is needed. We use the method described in [ASMH00] to model the application execution times for our simulation. The model consists of an Expected Time to Compute (ETC) matrix, that contains the estimation execution times of each task on all resources. The parameters of this model can be changed to investigate the performance of algorithms under different heterogeneous computing systems and under different types of tasks. The ETC model is based on three parameters: machine heterogeneity, task heterogeneity and consistency, which allow us to simulate various possible heterogeneous scheduling problems as realistically as possible [BSB+01]. The task heterogeneity is defined as the variety among the execution times of the tasks and, Machine heterogeneity, on the other hand, represents the possible variation of the running time of a particular task across all the processors.

Based on the two first parameters, four categories can be proposed for ETC matrix:

1. $\text{Machine}_{\text{heterogeneity}}$ = High, $\text{Task}_{\text{heterogeneity}}$ = High
2. $\text{Machine}_{\text{heterogeneity}}$ = High, $\text{Task}_{\text{heterogeneity}}$ = Low
3. $\text{Machine}_{\text{heterogeneity}}$ = Low, $\text{Task}_{\text{heterogeneity}}$ = High
4. $\text{Machine}_{\text{heterogeneity}}$ = Low, $\text{Task}_{\text{heterogeneity}}$ = Low

In our simulation we consider two values 0.2 and 0.8 as low and high heterogeneity values.

The ETC matrix can be further classified into two categories, *consistent* and *inconsistent*. In the consistent model, if for a task t_i a processor p_j has shorter execution time than processor p_k, them the same is true for any other task. In inconsistent model a processor p_j may execute some jobs faster than p_k and be slower for some other jobs. The consistent model can be seen as modelling a heterogeneous system in which the processors differ only in their processing speed and, a inconsistent model may represent a network in which there are different types of machines architectures. In this paper we consider the consistent model.

In order to generate dynamic and concurrent workflows scheduling, we consider 100 workflow applications, in each scenario, that arrive with time intervals that range from 10% to 90% of completed tasks, i.e., a new workflow is inserted when the corresponding percentage of tasks from

the last workflow currently in the system is completed. Each workflow application consists of a number of tasks between 10 and 100 tasks. For each workflow, we generate random values for k_{Budget} in the range $(0 \ldots 1)$.

7.5.2 Simulation platform

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 [1] [CLQ08] 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 heterogeneous computing systems, such as the Grid platform considered in this work. 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. In this experiments, we connected all processor over one shared bandwidth.

We consider platforms with 8 and 32 processors as a low and high number of processors, compared to the number of workflows, to analyze the behaviour of the algorithms with respect to the system load. The maximum load configuration is observed for 8 processors and 100 workflows.

7.5.3 Performance metric

The metric to evaluate a dynamic scheduler of independent workflows, must represent the individual completion time in order to measure the QoS experienced by the users related to the finish time of each user application. A global measure for the set of workflows would hide relevant delays on shorter workflows.

To evaluate the algorithms we consider the relative improvement on Turnaround Time ($TurnaroundTime_{imp}$) achieved with our strategy, for a given workflow, when compared to the maximum Turnaround Time achieved for that workflow among all strategies. The Turnaround time is the difference between submission and final completion of an application. The ($TurnaroundTime_{imp}$) is obtained by the ratio of the difference of turnaround time for a given workflow G and an algorithm alg_i, and the maximum turnaround time among all algorithms, as shown by equation 13.

$$TurnaroundTime_{imp}(alg_i) = \frac{\max_{alg_k \in alg_{set}}\{TT_G(alg_k)\} - TT_G(alg_i)}{\max_{alg_k \in alg_{set}}\{TT_G(alg_k)\}}, \tag{13}$$

where alg_{set} is the set of algorithms under comparison and $alg_i \in alg_{set}$. This metric for an algorithm alg_i gives the improvement achieved by each algorithm in comparison to the maximum

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

Turnaround Time obtained for a given workflow G. The algorithm that generates more relative improvements is the best algorithm.

7.5.4 Results and discussion

In this section, we compare the modified versions B-FDWS and B-RANK_HYBD in terms of $TurnaroundTime_{imp}$ value. We present results for 1000 instance of the scenario with a set of 100 DAGs that arrive with time intervals that range from 10% 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. In addition we generate 4 different types based on task heterogeneity and machine heterogeneity. In total we tested 4000 instances. Also we consider two different level of processors (low and high) compared to the number of DAGs, to analyse the behaviour of the algorithms with respect to different system load. We consider 2 sets of processors with 8 and 32 where the maximum load configuration is observed for eight processors. Based on our observations, task heterogeneity does not have a significant effect on results, therefore we do not categorize results based on this parameter.

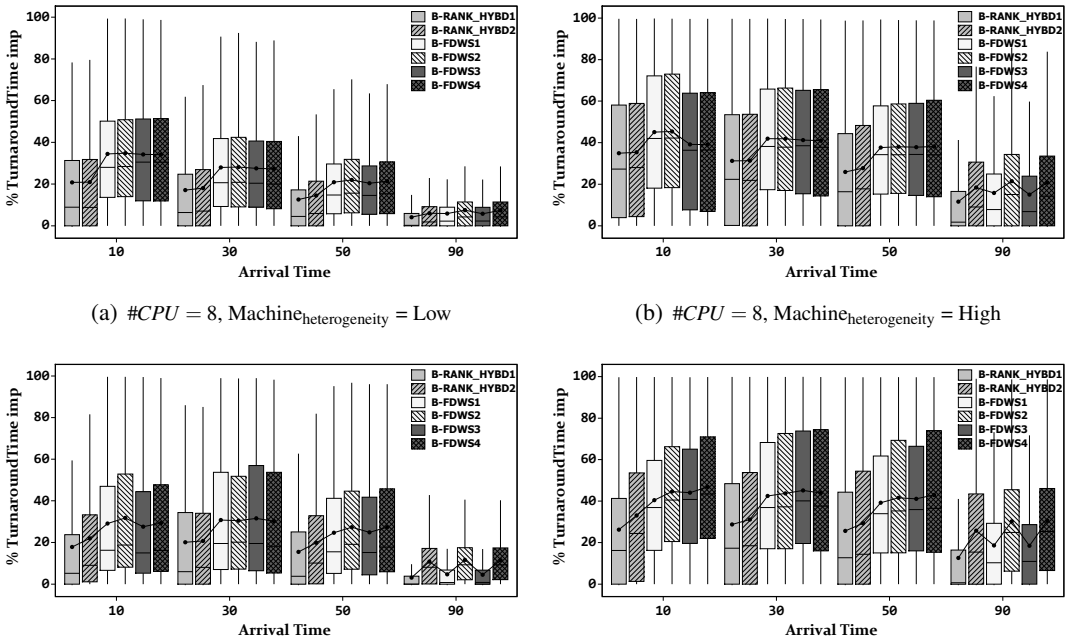
Results are presented attending to the three strategies referred to in Table 11, namely, Ready Pool Filling strategy, Task Selection strategy and Processor Selection strategy. Figure 2 shows the Turnaround Time percentage improvement achieved for the 6 algorithm's versions.

For low number of CPUs, as we can see in Fig 2(a) and Fig 2(b), the filling policy for adding ready tasks from workflow applications into ready tasks pool, is the strategy that differentiates the two algorithms RANK_HYBD and FDWS. The FDWS filling strategy, which selects a single task from each workflow, leads to higher fairness in scheduling and avoids the postponing of larger workflows as happens with RANK_HYBD, contributing to a better relative turnaround time. The improvements are more significant when we have higher concurrency in the system, i.e. low arrival time, starting on 67% improvement for arrival time interval of 10%, and 30% improvement for arrival time interval of 90%.

As we move to higher arrival time intervals, when comparing the algorithms versions that use the quality measure Q, with the ones that do not use it, we conclude that Q improves the algorithms performance. For instance, B-RANK_HYBD2 has 46% improvment over B-RANK_HYBD1, as well as 27% improvement for B-FDWS2 over B-FDWS1 and 28% for B-FDWS4 over B-FDWS3.

For higher arrival time intervals, which means lower concurrency, using the quality measure Q achieves higher turnaround time percentage improvement on platforms with larger values of machine heterogeneity. We obtained improvements of 58%, 36% and 39% of turnaround time, with arrival time of 90% for B-RANK_HYBD2, B-FDWS2 and B-FDWS4 over B-RANK_HYBD1, B-FDWS1 and B-FDWS3, respectively.

On the other hand, for higher number of CPUs, in addition to filling ready task policy, two other strategies, $rank_B$ and quality measure Q, proposed here, have higher influence in the improvements obtained with both algorithms. Fig. 2(c) and Fig. 2(d) show that, besides the filling ready task policy used by FDWS which improves algorithm performance over RANK_HYBD, quality measure Q always improves the algorithm's performance. The improvements of B-RANK_HYBD2,



(a) $\#CPU = 8$, $\text{Machine}_{\text{heterogeneity}}$ = Low

(b) $\#CPU = 8$, $\text{Machine}_{\text{heterogeneity}}$ = High

(c) $\#CPU = 32$, $\text{Machine}_{\text{heterogeneity}}$ = Low

(d) $\#CPU = 32$, $\text{Machine}_{\text{heterogeneity}}$ = High

Figure 2: Turnaround Time improvement values for 8 and 32 processors and, for low and high machine heterogeneity

FDWS2 and FDWS4 over B-RANK_HYBD1, FDWS1 and FDWS3, respectively, start at 55% for arrival time interval of 10% and increase to 240% for arrival time interval of 90%.

Comparing results of FDWS1 to FDWS3 or FDWS2 to FDWS4, we can conclude that $rank_B$, as the policy for selecting the task from ready tasks pool to be schedule, improves slightly the algorithm performances, in comparison to $rank_r$. The highest improvement observed is 9%.

7.6 Conclusion

In this paper we present a new strategy for dynamic scheduling of concurrent workflows with two conflicting QoS requirements. To the best of our knowledge, there is no previous research that deal with multiple workflow scheduling that are submitted at different moments in time and that are based on the two conflicting QoS parameters, namely, time optimization and cost constraint at the same time. We propose a new priority rank, called, $rank_B$ for task selection and a quality value Q for the processor selection phase on the workflow management system.

The new strategies allowed to obtain better performances in almost all presented cases. For some other configurations, such as lowest processor number (1/8 of the number of concurrent DAGs) and lower arrival time, the most significant factor is the filling of the ready pool. For the other configurations the most influence factor to improve performance is the usage of the quality factor Q proposed in this paper. Additionally, we apply this strategy to two state of the art

algorithms in order to consider the budget constraint, namely, FDWS and RANK_HYBD algorithms. In the future work, we intend to add a deadline constraint as another QoS parameter into our strategies.

Acknowledgments

This work was supported in part by the Fundação para a Ciência e Tecnologia, PhD Grant FCT-DFRH-SFRH/BD/80061/2011.

Chapter 8

Low-time complexity budget-deadline constrained workflow scheduling on heterogeneous resources

Hamid Arabnejad, Jorge G. Barbosa and Radu Prodan

Journal of Future Generation Computer Systems (FGCS),

volume 55, pages 29 - 40, 2016,

DOI: 10.1016/j.future.2015.07.021,

abstract

The execution of scientific applications, under the utility computing model, is constrained to Quality of Service (QoS) parameters. Commonly, applications have time and cost constraints such that all tasks of an application need to be finished within a user-specified Deadline and Budget. Several algorithms have been proposed for multiple QoS workflow scheduling, but most of them use search-based strategies that generally have a high time complexity, making them less useful in realistic scenarios. In this paper, we present a heuristic scheduling algorithm with quadratic time complexity that considers two important constraints for QoS-based workflow scheduling, time and cost, named Deadline-Budget Constrained Scheduling (DBCS). From the deadline and budget defined by the user, the DBCS algorithm finds a feasible solution that accomplishes both constraints with a success rate similar to other state-of-the-art search-based algorithms in terms of the successful rate of feasible solutions, consuming in the worst case only approximately 4% of the time. The DBCS algorithm has a low-time complexity of $O(n^2.p)$ for n tasks and p processors.

8.1 Introduction

Utility computing is a service provisioning model that provides computing resources and infrastructure management to consumers as they need them, as well as a payment model that charges for usage. Service-oriented grid and cloud computing, which supply frameworks that allow users to consume utility services in a secure, shared, scalable, and standard network environment, have become the basis for providing these services.

Computational grids have been used by researchers from various areas of science to execute complex scientific applications. Utility computing has been rapidly moving towards a pay-as-you-go model, in which computational resources or services have different prices with different performance and Quality of Service (QoS) levels [BVB08]. In this computing model, users consume services and resources when they need them and pay only for what they use. In this context, cost and time become two of the most relevant user concerns. Thus, the cost/time trade-off problem for scheduling workflow applications has become challenging. Scheduling consists of defining an assignment and mapping of the workflow tasks onto resources. In general, the scheduling problem belongs to a class of problems known as NP-complete [CB76].

Most research on workflow QoS aware scheduling considers the optimization of one QoS parameter, such as time, constrained to another QoS parameter, such as cost [AB14a, ZS13]. Other approaches consider a bi-objective approach that consists in optimizing two QoS parameters, such as time and cost simultaneously [PW10, SKD07, SLH+13], the constraints being mainly related to processor availability and load. Other combination of QoS parameters may be considered, such as time and reliability [DÖ05].

Workflow scheduling to satisfy multiple QoS parameters is becoming an active research area in the context of utility computing. Many algorithms have been proposed for multi-objective scheduling, but in most of them, meta-heuristic methods or search-based strategies have been used to achieve good solutions. However, these methods based on meta-heuristics or search-based strategies usually need significantly high planning costs in terms of the time consumed to produce good results, which makes them less useful in real platforms that need to obtain map decisions on the fly. In this paper, a low-time complexity heuristic, named Deadline-Budget Constrained Scheduling (DBCS), is proposed to schedule workflow applications on computational heterogeneous infrastructures constrained to two QoS parameters. In our model, the QoS parameters are time and cost. The objective of the proposed DBCS algorithm is to find a feasible schedule map that satisfies the user defined deadline and budget constraint values. To fulfill this objective, DBCS implements a mechanism to control the time and cost consumption by each task when producing a schedule solution. To the best of our knowledge, the algorithm proposed here is the first low-time complexity heuristic for a bounded number of heterogeneous resources addressing two QoS parameters that obtains similar performances to higher-time complexity scheduling algorithms in a small fraction of the scheduling time.

The contributions of this paper are:

- a review of multiple QoS parameter workflow scheduling on heterogeneous resources;

- a new heuristic algorithm with quadratic complexity for workflow application scheduling, constrained to time and cost, on a bounded set of heterogeneous resources;
- similar performances of search-based state-of-the-art algorithms in a small fraction of the time that ranges from 0.004% to 4%;
- a realistic simulation considering a bounded multi-port model in which bandwidth is shared by concurrent communications;
- extensive evaluation with results for randomly generated graphs as well as for real-world applications.

The remainder of the paper is organized as follows. Section 2 describes related work. Section 3 defines the scheduling problem and describes the system model. Section 4 presents the proposed scheduling algorithm. Section 5 presents results, and Section 6 concludes the paper.

8.2 Related Work

Workflow scheduling has been extensively investigated. The scheduling strategies can be classified into two main categories: single and multiple QoS parameters.

On the single QoS parameter, the execution time of a workflow application, also called *makespan*, has been the major concern in most of the scheduling strategies. In [AB14c, CJSZ08], a wide study on list scheduling algorithms is presented for *makespan* optimization.

The problem becomes more challenging when two or more QoS parameters are considered in the scheduling problem. Time, cost, energy and reliability are common QoS parameters considered in recent research work in this area. Many algorithms consider time and cost in their formulation but most of them perform: a) optimization of one parameter constrained to the other; b) optimization of both parameters in a bi-objective formulation; and c) a consideration of an unlimited number of resources, in particular for cloud platforms, where the strategy to accomplish time constraints is by allocating new computational instances.

Concerning the optimization of one parameter constrained to the other, Mao et al. [MH11] proposed an auto-scaling mechanism that automatically scales computing instances based on workload information to minimize the cost of the scheduling map while meeting application deadlines on cloud environments. Zeng et al. [ZVL12] proposed *ScaleStar*, a budget-conscious scheduling algorithm to minimize the execution time of large-scale many-task workflows in Clouds with monetary costs. Yu et al. [YBT+05b] proposed a QoS-based workflow scheduling algorithm utilizing a Markov Decision Process approach for the service Grid that minimizes the total cost of the application while meeting the deadline constraints imposed by the user. Their algorithm first categorizes tasks into two classes: synchronization tasks (the nodes that have more than one parent or child) and simple tasks. Then, the original workflow is partitioned into sub-workflows, and based on the two classes of tasks, sub-deadlines are assigned to each partition. Finally, the cost optimized mapping for each partition is obtained, guaranteeing the application deadline. In [YLW07],

Yuan et al.proposed a time-cost tradeoff dynamic heuristic scheduling strategy to optimize the cost and time of the whole workflow. In addition, [YLWZ09] presented a heuristic scheduling algorithm called DET (Deadline Early Tree), which minimizes cost with a deadline constraint. The communication time between tasks is not considered in their model. Sakellariou et al. [SZTD07] presented the LOSS1 algorithm to construct schedules that optimize time constrained to a cost. The algorithm uses initial assignments made by other heuristic algorithms to meet the time optimization objective; then, a reassignment strategy is implemented to reduce cost and meet the cost constraint that is specified by the user budget. Zheng et al., in [ZS12] and [ZS13], proposed the algorithm Budget-constrained Heterogeneous Earliest Finish Time (BHEFT), which optimizes the execution time of a workflow application constrained to a budget. Arabnejad et at.[AB14a] proposed a Heterogeneous Budget Constrained Scheduling (HBCS) algorithm that guarantees an execution cost within the user's specified budget and that minimizes the application execution time similarly to BHEFT. The results presented show that the HBCS algorithm achieves lower makespans, with a guaranteed cost per application. The algorithm proposed in this paper extends the HBCS algorithm to consider deadline (time) and budget (cost) as constraints. Similar to HBCS, in this paper, we propose a quality measure for each processor that combines time and cost constraints, which is used for processor selection and may not necessarily select the processor that guarantees the earliest finish time. BHEFT uses a different formulation that, in two steps, selects the set of affordable processors, the cost factor, and then selects the processor that minimizes the processing time. LOSS1 and BHEFT are also selected for comparison to the algorithm proposed in this paper, DBCS, as an alternative approach that optimizes time constrained to a budget.

The following algorithms use a bi-objective formulation. Talukder et al. [TKB09] proposed a workflow execution planning approach using Multi-objective Differential Evolution (MODE) to satisfy the user time and cost constraint parameters. Chen et al. [CZ09] proposed an ant colony optimization (ACO) to schedule large-scale workflows with various QoS parameters such as reliability, time, and cost in computational grids. Garg et al. [GS11] proposed a multi-objective non-dominated sort particle swarm optimization (NSPSO) approach to find schedule maps minimizing the makespan and total cost under the specified deadline and budget constraints. In [YB06b], Yu et al. proposed a genetic algorithm (GA) approach for scheduling workflow applications constrained to budget and deadline, on heterogeneous environments. Two fitness functions are used to encourage the formation of individuals who satisfy the deadline and budget constraints. Prodan et al. [PW10] proposed a general bi-criteria scheduling heuristic called the Dynamic Constraint Algorithm (DCA) based on dynamic programming to optimize two independent generic criteria for workflows, e.g., execution time and cost. The DCA scheduling algorithm has two main phases: The first selects one criterion as primary and optimizes it and, in the second phase, optimizes the secondary criteria while keeping the primary criteria within a defined sliding constraint. GA and DCA are considered in this paper for comparison to the DBCS algorithm as state-of-the-art search-based algorithms with higher-time complexity. In particular, DCA performs a full domain search in the second phase of the algorithm.

Other algorithms were proposed considering the cloud on-demand resource allocation. In

[ANE13], two scheduling algorithms for cost minimization constrained to a deadline for IaaS Cloud environments were proposed. Due to on-demand resource provisioning, the time constraint can always be accomplished as long as the cloud provides an unlimited number of computational resources. This model corresponds to an unbounded set of resources, which differs from our context that considers a bounded set of processors.

Other approaches, such as Malawski et. al [MJDN15], proposed three scheduling algorithms for scientific workflow ensembles on clouds to complete workflows from an ensemble under budget and deadline constraints. Our problem differs from such a model because we did not consider the composition of several inter-related workflows grouped into ensembles. A comprehensive survey about grid and cloud workflow scheduling is presented in [WWT15].

Among all of these previous works, we select a few that are closer to our context. Most of the works consider a scheduling decision to optimize for some QoS parameter while being subjected to some user-specified constraint or optimize all of the QoS parameters to provide a suitable balance between them. In this study, our goal is not the optimization of QoS parameters; instead, we consider budget and deadline constraints rather than cost and time optimization as goals similarly to [YB06b].

In terms of time complexity, most of the scheduling strategies mentioned above are search based and usually require long execution time before producing good results, making them less useful in a realistic computational infrastructure.

In this paper, our goal is to propose a novel low-time complexity workflow scheduling algorithm that addresses the budget and deadline constraints. To evaluate the performance of our scheduling strategy, we select four well-known algorithms, namely, DCA [PW10], Genetic Algorithm (GA) [YB06b], LOSS1 [SZTD07] and BHEFT [ZS12, ZS13]. The DCA and GA scheduling algorithms are search-based methods that obtain near-optimal schedule maps among the set of solutions. The LOSS1 algorithm tries to accomplish the objective function by using a task reassignment (rescheduling) strategy. The BHEFT is a low-time complexity scheduling strategy that minimizes time constrained to a budget, but it is included here as a low-time complexity alternative. In addition to these algorithms, we include a *RANDOM* strategy for task assignment, i.e., the scheduler randomly selects a resource without taking into account the execution time. This is the lower-time complexity algorithm, and it is also the base line for comparison.

8.3 Problem Definition

This section presents the system model, the application model and the scheduling objectives.

8.3.1 System Model

The target utility computing platform is composed of a set of heterogeneous resources that provide services of different capabilities and costs [BVB08]. Processor price is defined so that the most powerful processor has the highest cost and the less powerful processor has the lowest cost. In a utility grid, the resource price is commonly defined and charged per time unit [PW10, YB06b,

ZS12], so that if a task takes k time units to process in a resource that costs y euros per time unit, the cost of executing the task in that resource is $k \times y$ euros. In a cloud environment, the granularity of charging resource usage varies, charging per hour being a common practice, such as in Amazon Elastic Compute Cloud (Amazon EC2) [1], and partial hours are rounded up. This approach has the disadvantage that there is no cost benefit of adding resources for workflows that run for less than an hour [JDB+12]. It also makes the effective evaluation of scheduling algorithms and solutions difficult. Therefore, in this paper, we consider that a heterogeneous resource has a set of processors available and that each one has a price per time unit, so that the cost imputed to the workflow execution is only the effective used time. Although this assumption may seem unrealistic for the hour pricing model, it is only true if we consider the current dominant policy and individual user contracts with the provider. However, other models may be explored such as, for example, a group of users that rent a set of resources so that resources are shared and probably cost less than individual contracts. This is a feasible approach for a research group. Other players are more flexible and charge per minute, with a minimum of 10 minutes, such as the Google Compute Engine [2]. Additionally, there is an emergent market of cloud providers that put in the market the spare resources of their private clouds, so that a wider set of pricing models may be available in the near future. Therefore, for the sake of an unambiguous comparison of the results produced by the scheduling algorithms, as in [PW10, YB06b, ZS12], we consider the second as the time unit, as well as the unit for processor charging.

8.3.2 Application Model

A typical workflow application can be represented by a Directed Acyclic Graph (DAG), a directed graph with no cycles. A DAG can be modeled by a three-tuple $G =< T, E, Data >$. Let n be the number of tasks in the workflow. The set of nodes $T = \{t_1, t_2, \cdots, t_n\}$ corresponds to the tasks of the workflow. The set of edges E represent their data dependencies. A dependency ensures that a child node cannot be executed before all its parent tasks finish successfully and transfer the required child input data. $Data$ is a $n \times n$ matrix of communication data, where $data_{i,j}$ is the amount of data that must be transferred from task t_i to task t_j. The average communication time between the tasks t_i and t_j is defined as:

$$\overline{C}_{(t_i \to t_j)} = \overline{L} + \frac{data_{i,j}}{\overline{B}} \tag{1}$$

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 [THW02].

Due to heterogeneity, each task may have a different execution time on each processor. Then, $ET(t_i, p_j)$ represents the Execution Time to complete task t_i on processor p_j in available processors

[1] http://aws.amazon. com/ec2
[2] http://cloud.google.com/compute/

set P. The average execution time of task t_i is defined as:

$$\overline{ET}(t_i) = \frac{\sum_{p_j \in P} ET(t_i, p_j)}{|P|} \tag{2}$$

where $|P|$ denotes the number of resources in processor set P.

In a given DAG, a task with no predecessors is called an *entry task* and a task with no successors is called an *exit task*. We assume that the DAG has exactly one entry task t_{entry} and one exit task t_{exit}. If a DAG has multiple entry or exit tasks, a dummy entry or exit task with zero weight and zero communication edges is added to the graph.

In addition to these definitions, we next present some of the common attributes used in task scheduling, which will be used in the following sections.

$pred(t_i)$ and $succ(t_i)$ denote the set of immediate predecessors and immediate successors of task t_i, respectively. $FT(t_i)$ is defined as the Finish Time of task t_i on the processor assigned by the scheduling algorithm.

Schedule length or *makespan* denotes the finish time of the last task of the workflow and is defined as $makespan = FT(t_{exit})$.

$EST(t_i, p_j)$ and $EFT(t_i, p_j)$: denotes Earliest Start Time (EST) and the Earliest Finish Time (EFT) of a task t_i on processor p_j, respectively, and are defined as:

$$EST(t_i, p_j) = \max\left\{ T_{Available}(p_j), \max_{t_{parent} \in pred(t_i)} \{AFT(t_{parent}) + C_{(t_{parent} \to t_i)}\} \right\} \tag{3}$$

$$EFT(t_i, p_j) = EST(t_i, p_j) + ET(t_i, p_j) \tag{4}$$

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 t_i arrive at the processor p_j. The communication time $C_{(t_{parent} \to t_i)}$ is zero if the predecessor node t_{parent} is assigned to processor p_j. For the entry task, $EST(t_{entry}, p_j) = 0$. Then, to calculate EFT, the execution time of task t_i on processor p_j (ET) is added to its Earliest Start Time.

The financial cost $Cost(t_i, p_j)$ of executing task t_i on specific processor p_j during the time span of $ET(t_i, p_j)$ is the sum of three cost components:

$$Cost(t_i, p_j) = EC(t_i, p_j) + TC(t_i) + SC(t_i) \tag{5}$$

where $EC(t_i, p_j)$ denotes the cost of running task t_i on processor p_j and is defined as $EC(t_i, p_j) = ET(t_i, p_j) \times Price(p_j)$ where $Price(p_j)$ denotes the processor price per time unit. $TC(t_i)$ denotes the cost of transferring the data required for task t_i. In addition, $SC(t_i)$ denotes the data storage cost of task t_i. These cost components are determined by the target platform infrastructure.

$TotalCost$ is the overall cost for executing an application and is defined as:

$$TotalCost = \sum_{t_i \in T} AC(t_i) \tag{6}$$

where $AC(t_i)$ is defined as Assigned Cost of task t_i. After assigning a processor p_{sel} to execute task t_i, the assigned cost value is equal to $AC(t_i) = Cost(t_i, p_{sel})$. In the case of an intra-cluster data transfer, zero monetary costs for communications between tasks are considered, i.e $TC(t_i) = 0$. We also considered zero cost for task storage usage, $SC(t_i) = 0$, as this factor is common to all algorithms and does not influence the comparison of results.

8.3.3 Scheduling problem

For a given workflow G, a scheduling is defined by a function $sched_G : T \rightarrow P$ which assigns to each task $t_i \in T$ a processor $p_j \in P$, subject to:

- Processor constraint so that no processor executes more than one task at the same time;
- Precedence constraint represented by the edges of the workflow G;
- Budget constraint defined as:

$$\sum_{t_i \in T} AC(t_i) \leq BUDGET \tag{7}$$

- Deadline constraint defined as:

$$Makespan_G \leq DEADLINE \tag{8}$$

The scheduling consists in finding a schedule map between tasks and resources that can meet user-defined QoS constraints, i.e., the total execution time of the workflow must not be larger than the user defined deadline, and the total cost must not be larger than the user defined budget. These values of $DEADLINE$ and $BUDGET$ should be negotiated between users and providers in a range of feasibility values so that there are feasible scheduling solutions. Note that there is no optimization function in the formulation, so that any scheduling solution that matches the constraints is a plausible solution.

8.4 Proposed Deadline-Budget Constrained Scheduling Algorithm

In this section, we present the Deadline-Budget Constrained Scheduling algorithm (DBCS), which aims to find a feasible schedule within a budget and deadline constraints. The DBCS algorithm is a heuristic strategy that in a single step obtains a schedule that always accomplishes the budget constraint and that may or may not accomplish the deadline constraint. If the time constraint is met, we have a successful schedule; otherwise, we have a failure, and no schedule is produced. The

algorithm is evaluated based on the success rate. Before the description of the DBCS algorithm, we next present the attributes used in the algorithm:

- t_{curr} denotes the current task to be scheduled, selected on the task selection phase among all ready tasks;
- $FT_{min}(t_{curr})$ and $FT_{max}(t_{curr})$ denote the minimum and maximum finish time of current tasks among all available processors. Idle slots in the processor schedule, which can accommodate the current task, are also considered;
- $Cost_{min}(t_{curr})$ and $Cost_{max}(t_{curr})$ denote the minimum and maximum execution cost of the current task among all available processors on the target platform;
- $Cost_{best}(t_{curr})$ is the execution cost of the current task on the processor that obtains the lowest finish time among all available processors;
- Δ_{Cost} represents the spare budget defined as the difference between unconsumed budget and cheapest cost assignment for unscheduled tasks. The initial value is $\Delta_{Cost} = BUDGET_{user} - Cost_{cheapest}$ where $BUDGET_{user}$ is the user defined budget as maximum allowed cost and $Cost_{cheapest}$, defined as $Cost_{cheapest} = \sum_{t_i \in T} Cost_{min}(t_i)$, is the cost of the cheapest assignment and represents the cost lower bound for executing the application. Δ_{Cost} is updated at each step after selecting the processor for the current task t_{curr}, as shown in eq (9):

$$\Delta_{Cost} = \Delta_{Cost} - \left[AC(t_{curr}) - Cost_{min}(t_{curr}) \right] \tag{9}$$

The DBCS, as a list scheduling algorithm, consists of two phases, namely, a *task selection* phase and a *processor selection* phase as described next.

8.4.1 Task Selection

Tasks are selected according to their priorities. To assign a priority to a task in the DAG, the upward rank ($rank_u$) [THW02] is computed. This rank represents, for a task t_i, the length of the longest path from task t_i to the exit node(t_{exit}), including the computational time of t_i, and it is given by Eq.10:

$$rank_u(t_i) = \overline{ET}(t_i) + \max_{t_{child} \in succ(t_i)} \left\{ \overline{C}_{t_i \to t_{child}} + rank_u(t_{child}) \right\} \tag{10}$$

where $\overline{ET}(t_i)$ is the average execution time of task t_i over all resources, $\overline{C}_{t_i \to t_{child}}$ is the average communication time between two tasks t_i and t_{child}, and $succ(t_i)$ are the set of immediate successor tasks of task t_i. To prioritize tasks, it is common to consider average values because they have to be prioritized before knowing the location where they will run. For the exit node, $rank_u(t_{exit}) = \overline{ET}(t_{exit})$.

8.4.2 Processor Selection

The processor to be selected to execute the current task is guided by the following quantities related to cost and time. To control the consumed cost and time, a limit value for each factor is needed. We define two variables, CL and DL as limits for cost and time. To select the best suitable processor, a trade-off between these two variables is evaluated. In the following paragraphs, we describe these two variables in detail.

$CL(t_{curr})$ is the maximum available budget for the current task t_{curr} that can be consumed by its assignment, and it is defined as the minimum cost for t_{curr} plus the spare budget available:

$$CL(t_{curr}) = Cost_{min}(t_{curr}) + \Delta_{Cost} \tag{11}$$

All available processors are filtered by $CL(t_{curr})$ to guarantee that the application can be executed without exceeding the budget constraint. We defined this filtered processor set as admissible processors, $P_{admissible}$. In the most restricted case, only the cheapest processors are considered. Otherwise, no feasible schedule exists under the user defined budget.

$DL(t_{curr})$ is defined as the sub-DeadLine that is assigned to each task based on the total application deadline. There are some studies that proposed different strategies to distribute workflow deadlines among tasks. In [YBT05a], tasks are grouped in different levels based on their depth in the graph, and then the final deadline is divided into levels in such a way that all tasks belonging to the same level have the same sub-deadline. In [YBT+05b], first the original workflow is partitioned into sub-workflows, and then the total deadline is divided among partitions. In this paper, we apply the common and direct project planning sub-deadline distribution strategy. The sub-deadline value for each task t_i is computed recursively by traversing the task graph upwards, starting from the exit task. Due to heterogeneity, sub-Deadline can be defined in several different forms. Here, we consider the minimum execution time of the current task, as shown by Eq.11:

$$DL(t_{curr}) = \min_{t_{child} \in succ(t_{curr})} \left[DL(t_{child}) - \overline{C}_{(t_{curr} \to t_{child})} - ET_{min}(t_{child}) \right] \tag{12}$$

where ET_{min} is defined as the minimum execution time of task t_{curr} among available processors. For the exit task, the sub-deadline is equal to the user defined deadline, $DL(t_{exit}) = DEADLINE$.

Unlike the cost limit, the sub-Deadline is a soft limit as in most deadline distribution strategies on grid platforms with a fixed number of available resources [YRB09]; if the scheduler cannot find a processor that satisfies the sub-deadline for the current task, the processor that can finish the current task at the earliest time is selected.

The processor selection phase is based on the combination of the two QoS factors, time and cost, to obtain the best balance between time and cost minimum values. We define two relative quantities, namely, Time Quality ($Time_Q$) and Cost Quality ($Cost_Q$), for current task t_{curr} on each admissible processor $p_j \in P_{admissible}$, shown in (16) and (17), respectively. Both quantities are

normalized by their maximum values.

$$Time_Q(t_{curr}, p_j) = \frac{\Omega \times DL(t_{curr}) - FT(t_{curr}, p_j)}{FT_{max}(t_{curr}) - FT_{min}(t_{curr})} \tag{13}$$

$$Cost_Q(t_{curr}, p_j) = \frac{Cost_{best}(t_{curr}) - Cost(t_{curr}, p_j)}{Cost_{max}(t_{curr}) - Cost_{min}(t_{curr})} \times \Omega \tag{14}$$

where

$$\Omega = \begin{cases} 1 & \text{if} \quad FT(t_{curr}, p_j) < DL(t_{curr}) \\ 0 & otherwise \end{cases} \tag{15}$$

$Time_Q$ measures how much closer to the task sub-deadline (DL) the finish time of the current task on processor p_j is. Processors with higher $Time_Q$ values have a greater possibility of being selected. If the current task has a higher finish time on processor p_j than its sub-deadline, $Time_Q$ assumes a negative value for p_j, reducing the possibility of this processor being selected.

Similarly, $Cost_Q$ measures how much less the actual cost on p_j is than the cost on the processor that results in the earliest finish time ($Cost_{best}$). Although CL, eq(11), is the maximum allowed cost for the current task, here, $Cost_{best}$ is used to avoid selecting a processor that performs worse and costs more than the processor that guarantees the earliest finish time.

In the case that none of the processors from $P_{admissible}$ can guarantee t_{curr} sub-deadline, $Cost_Q$ is zero for all of them, and $Time_Q$ for each processor p_j is a negative value that represents the relative finish time obtained with p_j. The processor from $P_{admissible}$ with higher $Time_Q$, i.e., closer to zero, would be selected. Note that, in any case, cost will be lower than CL, the maximum available budget for the current task.

Finally, to select the most suitable processor for the current task, the Quality measure (Q) for each processor $p_j \in P_{admissible}$ is computed as shown in Eq(18):

$$Q(t_{curr}, p_j) = Time_Q(t_{curr}, p_j) \; + Cost_Q(t_{curr}, p_j) \times \frac{Cost_{Cheapest}}{Budget_{Unconsumed}} \tag{16}$$

where the cost quality factor is weighted by the ratio of the cheapest cost execution for unscheduled tasks over the unconsumed budget, so that the effectiveness of the cost quality factor can be controlled. A higher value of the fraction means that the unconsumed budget is close to the cheapest cost execution for unscheduled tasks, so that the cost factor is more predominant in the processor Quality measure. In the same way, a lower value means a higher difference between unconsumed budget and cheapest cost execution for unscheduled tasks, so that the cost factor is less influential, allowing the selection of more expensive processors that guarantee a lower processing time for t_{curr}.

The DBCS algorithm is shown in Algorithm 1. First, the possibility of finding a schedule map under a user defined budget is checked in lines 1-3. After some initializations in lines 4-5, the algorithm starts to map all tasks of the application (while looping in lines 6-14). At each step, on

line 7, among all ready tasks, the task with highest priority ($rank_u$) is selected as the current task (t_{curr}). Then, in lines 8-10, the Quality measure for assigning t_{curr} to processor p_j ($Q(t_{curr}, p_j)$) is calculated. Note that, first, the finish time (FT) and execution cost of the current task is calculated and then the quality measure for all *admissible* processors is calculated. Next, the processor with the highest quality measure among all processors is selected (line 11-12). Finally, after assigning the processor to the current task, the Δ_{Cost} variable is updated using Eq.9 (line 13).

Algorithm 1 DBCS algorithm

Require: a DAG and user's QoS Parameters values for time ($DEADLINE_{user}$) and cost ($BUDGET_{user}$)

1: **if** $BUDGET_{user} < Cost_{cheapest}$ **then**
2: **return** no possible schedule map
3: **end if**
4: Initialize $\Delta_{Cost} = BUDGET_{user} - Cost_{cheapest}$
5: Compute the upward rank ($rank_u$) and sub-DeadLine value (DL) for each task
6: **while** there is an unscheduled task **do**
7: t_{curr} = the next ready task with highest $rank_u$ value
8: **for all** $p_j \in P_{admissible}$ **do**
9: calculate Quality measure $Q(t_{curr}, p_j)$ using Eq.18
10: **end for**
11: P_{sel} = Processor p_j with highest Quality measure (Q)
12: Assign current task t_{curr} to Processor P_{sel}
13: Update Δ_{Cost} using Eq.(9)
14: **end while**
15: **return** Schedule Map

In terms of time complexity, DBCS requires the computation of the upward rank ($rank_u$) and sub-DeadLines (DL) for each task that have complexity $O(n.p)$, where p is the number of available resources and n is the number of tasks in the workflow application. In the processor selection phase, to find and assign a suitable processor for the current task, the complexity is $O(n.p)$ for calculating FT and $Cost$ for the current task among all processors, plus $O(p)$ for calculating the Quality measure. The total time is $O(n.p + n(n.p + p))$, where the total algorithm complexity is of the order $O(n^2.p)$.

8.5 Experimental Results

This section presents performance comparisons of the DBCS algorithm with DCA [PW10], LOSS1 [SZTD07], GA [YB06b], BHEFT [ZS12, ZS13] and RANDOM scheduling algorithms. We consider synthetic randomly generated and Real Application workflows to evaluate a wider range of loads. The results presented were produced with SIMGRID [CLQ08], which is one of the simulators for distributed computing and allows for a realistic description of the infrastructure parameters.

8.5.1 Workflow Structure

To evaluate the relative performances of the algorithms, both the randomly generated and real-world application workflows were used; namely, MONTAGE and EPIGENOMICS [BCD+08] are used. The randomly generated workflows were created by the synthetic DAG generation program[3]. The computational complexity of a task is modelled as one of the three following forms, which are representative of many common applications: $a.d$ (e.g., image processing of a $\sqrt{d} \times \sqrt{d}$ image), $a.d\log d$ (e.g., sorting an array of d elements), $d^{3/2}$ (e.g., multiplication of $\sqrt{d} \times \sqrt{d}$ matrices) where a is picked randomly between 2^6 and 2^9. As a result, different tasks exhibit different communication/computation ratios.

The DAG generator program defines the DAG shape based on four parameters: *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, it was considered as the number of tasks $n = [30, 70, 90]$, $jump = [1, 2, 3]$, $fat = [0.2, 0.4, 0.8]$, $regularity = [0.2, 0.8]$, and $density = [0.2, 0.8]$. With these parameters, each DAG is created by picking the value for each parameter randomly from the parameter data set. The total number of DAGs generated in our simulation is 10000.

8.5.2 Simulation Platform

We resorted to simulation to evaluate the algorithms discussed in the previous sections. Simulation allows us to perform a statistically significant number of experiments for a wide range of application configurations in a reasonable amount of time. We used the SIMGRID toolkit[4] [CLQ08] 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 heterogeneous computing system, such as the grid platform considered in this work.

We consider three sites that comprise multiple clusters and have different CPU power compositions. The Rennes site has normal distribution of CPU power among its clusters, and the Sophia site contains a higher number of low-speed processors, while the Lille site has a higher number of fast processors.

[3] https://github.com/frs69wq/daggen
[4] http://simgrid.gforge.inria.fr

To normalize diverse price units for the heterogeneous processors, as defined in [ZS12], the price of a processor $p_j \in P$ is assumed to be $Price(p_j) = \alpha_{p_j}(1+\alpha_{p_j})/2$, where α_{p_j} is the ratio of p_j processing capacity to that of the fastest processor of the set P. The price will be in the range of $]0 \cdots 1]$, where the fastest processor, with the highest power, has a price value equal to 1.

Table 13 provides the name of each site, along with the set of clusters that compose the site. For each cluster, it presents the total number of processors ($\#CPU_{Total}$), processing speed expressed in GFlop/s and processor cost. $\#CPU_{used}$ shows the number of processors used from each cluster for 8-, 16- and 32-processor configurations.

Site	Cluster	$\#CPU_{Total}$	$\#CPU_{used}$	Power(GFlop/s)	Cost($)
rennes	paradent	64	3 7 13	21.496e9	0.61$
	paramount	33	2 3 6	12.910e9	0.31$
	parapide	25	1 2 4	30.130e9	1.00$
	parapluie	40	2 4 9	27.391e9	0.87$
sophia	helios	56	3 6 12	7.7318E9	0.16$
	sol	50	3 5 10	8.9388e9	0.19$
	suno	45	2 5 10	23.530e9	0.70$
lille	chicon	26	2 4 9	8.9618e9	0.19$
	chimint	20	2 4 7	23.531e9	0.70$
	chinqchint	46	4 8 16	22.270e9	0.64$

Table 12: Description of the Grid5000 clusters from which the platforms used in the experiments were derived

8.5.3 Budget and Deadline parameters

To evaluate the DBCS algorithm, in our simulation, we need to define a value for time and cost as DEADLINE and BUDGET constraint parameters. For each site, these parameters are computed independently of the number of CPUs on that site. Additionally, to have better performance analysis, the DEADLINE and BUDGET values are calculated among all possible sites in our tested platform, i.e., all resources in the three possible sites `rennes`, `sophia` and `lille` are considered regardless of the site at which the DAG is to be executed.

To specify the deadline parameter, we define the min_{time} and max_{time} as the lowest and highest execution time of the application as shown in Eq.17 and Eq.18.

$$min_{time} = \sum_{t_i \in CP} \left(ET^*_{min}(t_i) + \overline{C}_{(t_{parent_{CP}} \rightarrow t_i)} \right) \tag{17}$$

$$max_{time} = \sum_{t_i \in CP} \left(ET^*_{max}(t_i) + \overline{C}_{(t_{parent_{CP}} \rightarrow t_i)} \right) \tag{18}$$

where CP is the set of tasks belonging to the critical path, $t_{parent_{CP}}$ is the critical parent of task t_i and $\overline{C}$ is the average communication time between task t_i and its critical parent. $ET^*_{min}(t_i)$ and $ET^*_{max}(t_i)$ are defined as the minimum and the maximum execution time for task t_i on the fastest and the slowest processor among all sites. In our tested sites, the slowest and fastest processors,

$P_{fastest}$ and $P_{slowest}$, belong to cluster `parapide` and `helios`, respectively. Please note that both of these values are defined as the lowest and highest possible makespans based on an infinite number of CPUs. For a bounded number of resources, the minimum and maximum execution time may be very optimistic and not reachable. The processing time range is defined based on the critical path to specify a deadline based on the lower bound of the makespan.

Similarly, to specify a budget constraint, we need to estimate the maximum and the minimum cost to obtain a range of feasible budgets to execute the application. We defined the max_{cost} and min_{cost} as the absolute highest and lowest possible costs for executing the application, which are calculated by summing the maximum and the minimum execution costs for each task, respectively, among all resources in all sites.

With these highest and lowest bound values, we define for the current application a unique DEADLINE and BUDGET constraint, independently of the execution site, as described by Eq (23) and Eq 24):

$$DEADLINE_{user} = min_{time} + \alpha_D \times (max_{time} - min_{time}) \tag{19}$$

$$BUDGET_{user} = min_{cost} + \alpha_B \times (max_{cost} - min_{cost}) \tag{20}$$

where the deadline parameter α_D and budget parameter α_B can be selected in the range of $[0 \ldots 1]$.

8.5.4 Performance Metric

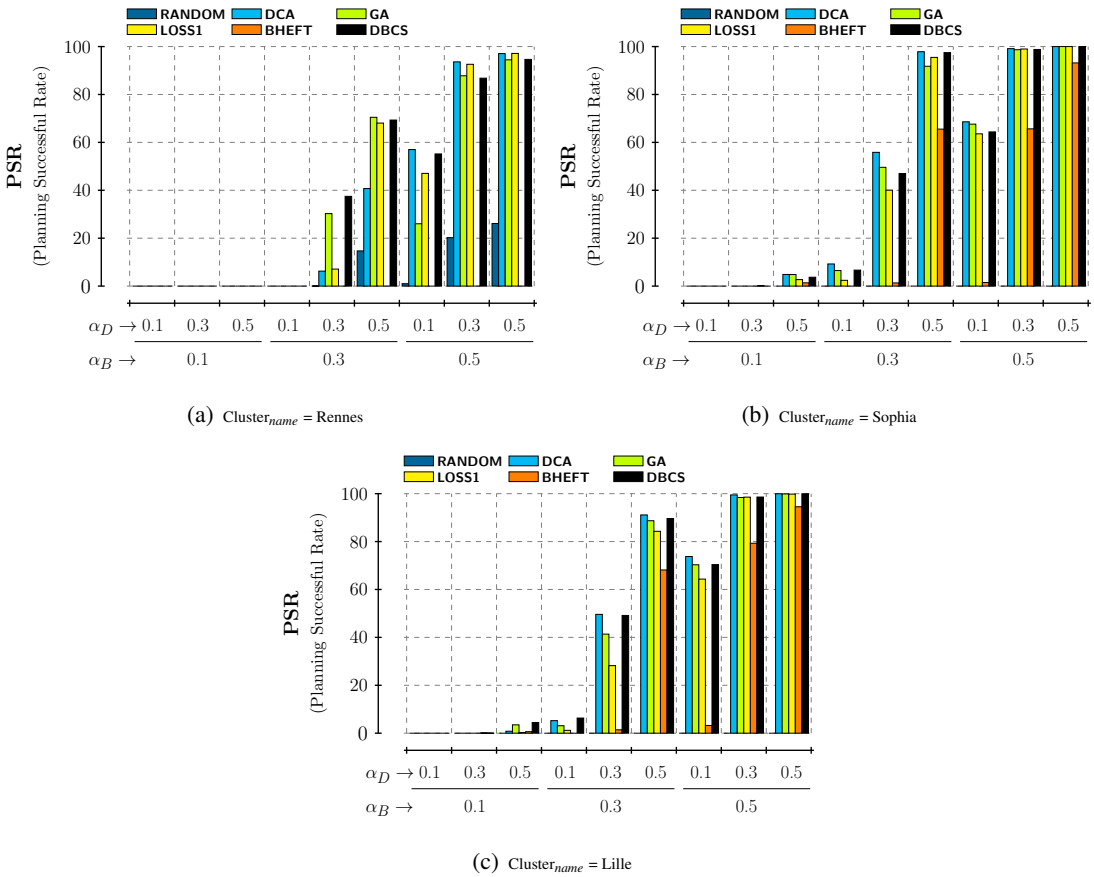
To evaluate and compare our algorithm with other approaches, we consider the Planning Successful Rate (PSR), as expressed by Eq (25). This metric provides the percentage of valid schedules obtained in a given experiment.

$$\text{PSR} = 100 \times \frac{Successful\ Planning}{Total\ Number\ in\ experiment} \tag{21}$$

8.5.5 Results and Discussion

As we selected GA, DCA, LOSS1 and BHEFT algorithms for comparison, we first describe the adaptations considered from their original strategies. A RANDOM scheduling algorithm is also considered.

The original implementation of the LOSS1 scheduling algorithm assumed that all of the processors had different costs, and therefore, there was no conflict in selecting a processor based on the cost parameter. In our heterogeneous environment, each cluster is homogeneous, so there could be more than one processor candidate. In this case, we test all of the possible processors and selected the one that achieves the earliest finish time. Also, to apply the time and cost constraint parameters in the algorithm, the loop exit point to test and generate a schedule map changed to cover QoS parameters time and cost. For the GA scheduling algorithm, the default configuration used for producing results is: population size equals 300, swapping mutation, replacing mutation



(a) Cluster$_{name}$ = Rennes

(b) Cluster$_{name}$ = Sophia

(c) Cluster$_{name}$ = Lille

Figure 1: Planning Success Rate for Random workflows

probability equal to 0.5 and a generation limit of 100. For the DCA algorithm, we considered 100 cells for the memorization table and a maximum of 10 intermediate solutions per cell. In both algorithms, as stated before, we stop the solution search after obtaining the first feasible solution for the scheduling problem in our implementation. Undoubtedly, increasing the configuration parameters for the GA and DCA algorithms, we would be able to achieve higher successful percentage rates, but it would increase the execution time of the algorithms exponentially. Here, we use the same configurations as in their original papers. For BHEFT, we stop the makespan optimization constrained to the defined BUDGET, as the current makespan is lower than the defined DEADLINE. The RANDOM scheduling strategy uses the task selection phase, as described in 8.4.1, to select the current task but selects the processor randomly for each task.

8.5.5.1 Results for Randomly Generated Workflows

For randomly generated workflows, we model the computational complexity of common application tasks, such as image processing, array sorting, and matrix multiplication. To observe the ability of finding valid schedule maps, we selected a low set of values, $\{0.1, 0.3, 0.5\}$, for time and cost parameters (α_D and α_B) to test the performance of each algorithm on harder conditions.

Figure 2 shows the average Planning Successful Rate (PSR) obtained on three different sites with CPU configurations per site equal to 8, 16 and 32 processors. The main result is that the algorithm DBCS obtains similar performance to other state-of-the-art search-based algorithms for the range of budget and deadline values considered here. For the Sophia and Lille sites, DBCS is very close to DCA performance and better than GA and LOSS1 for most of the cases. In addition, DBCS always obtains better results than BHEFT, which presents very low PSR values for the most restricted cases. On the contrary, even on the most restricted cases, DBCS always obtains PSR values close to the DCA results. Due to the definition of the budget that is based on the cheapest processor across all sites, the lower budget values are too restrictive on the `Rennes` site, as its cheapest processor cost about twice the $P_{slowest}$. Therefore, the PSR values in `Rennes` are lower than in the other two sites. DBCS achieves similar PSR values as the best higher-time complexity algorithms, which in some cases is DCA and in other cases is GA. On the `Rennes` site, BHEFT did not produce valid schedules. The RANDOM scheduling strategy gives the lowest PSR values, which were near zero.

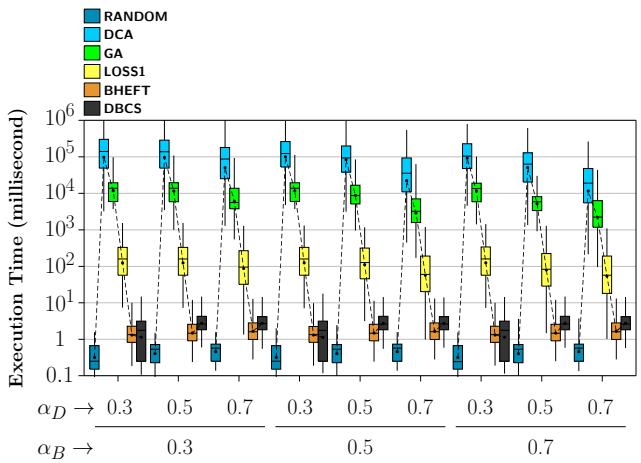


Figure 2: Execution time for each algorithm to find a valid solution (log scale)

As a heuristic algorithm, the main advantage of the DBCS consists in having an execution time in the range of the heuristic algorithms, such as BHEFT and RANDOM, but a planning success rate similar to the higher-time complexity search-based algorithms. Figure 2 shows, in logarithmic scale, the processing time of each of the algorithms to find a valid solution. We can observe that the algorithm DBCS requires an average of 4 milliseconds to obtain a valid solution, whereas other algorithms take substantially more time on average, DCA being the most expensive, with an average value of approximately 100 seconds, followed by GA with approximately 10 seconds and LOSS1 with approximately 100 milliseconds. Compared to those algorithms, DBCS produces schedules in a fraction of the time, which ranges from 0.004% to 4%. The three heuristic algorithms, DBCS, BHEFT and RANDOM, have a time complexity of $O(n^2.p)$ for n nodes and p processors and, therefore, obtain an execution time in the same range, below 10 milliseconds. However, DBCS achieves higher planning success rates than both of them, as shown above.

8.5.5.2 Results for Real World Applications

To evaluate the algorithms on a standard and real set of workflow applications, a set of workflows were generated using the code developed in Pegasus toolkit[5]. Two well-known structures were chosen [JCD+13], namely: MONTAGE and EPIGENOMICS. MONTAGE is an I/O-intensive workflow, and EPIGENOMICS is a compute-intensive workflow. For each of the MONTAGE and EPIGENOMICS workflows, we generated 300 DAGs with a number of tasks equal to 46 and 50, respectively. Additionally, we consider the application Wien2K [BSM+01], which is a material science workflow for performing electronic structure calculations of solids that contains two parallel sections with sequential synchronization activities in between. In this experiment, we use Wien2k workflows with 30 tasks that were obtained from trace data collected from historical executions conducted in the Austrian Grid[6]. To complement the characterization of these real world applications, Figure 3 shows the Communication to Computation Ratio (CCR) associated with each application. Once a deadline is specified based on the workflow critical path, if the workflow has higher CCR distribution or higher average CCR value, it will decrease the success rate of that workflow.

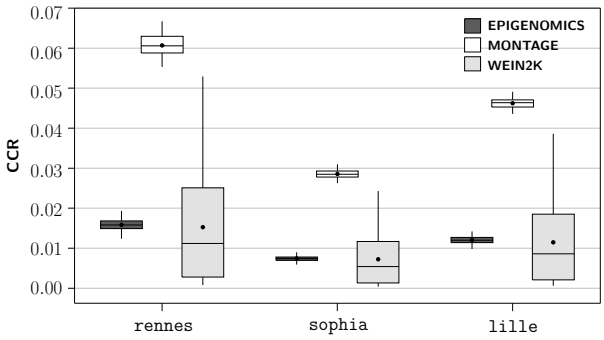
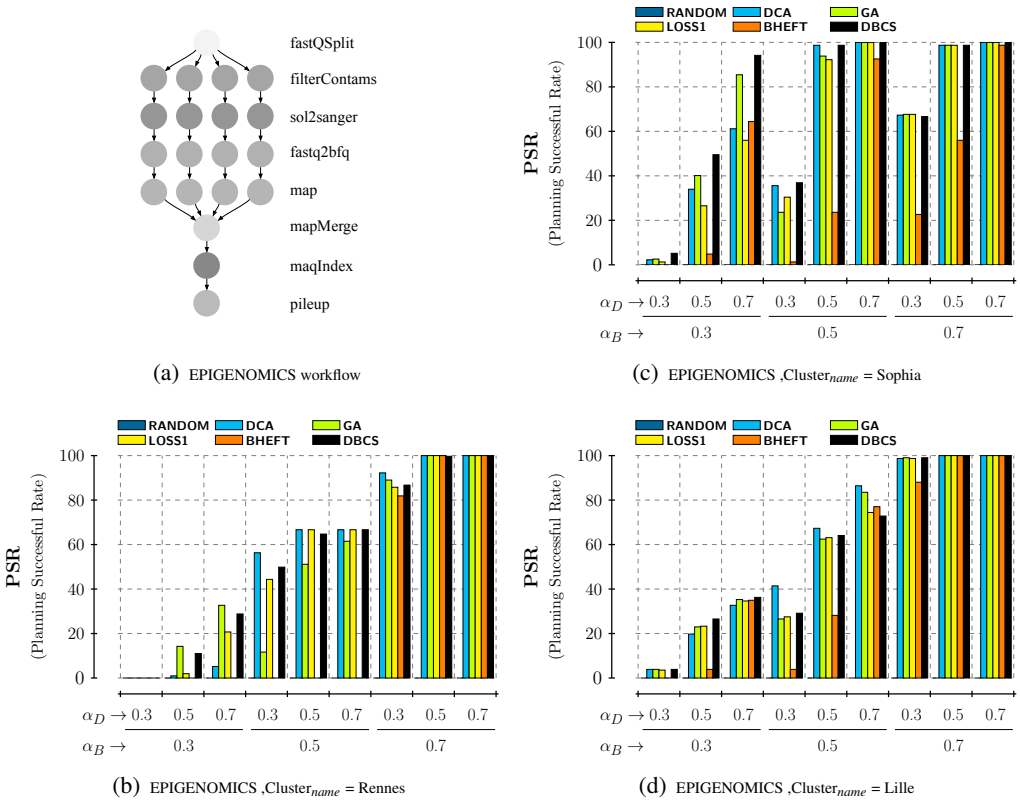


Figure 3: CCR values for each workflow on different sites

We consider the following values for time and cost parameters: $\alpha_D, \alpha_B \in \{0.3, 0.5, 0.7\}$. Figure 4, 5 and 6 shows the PSR for each real application and for three different sites.

As the results for random graph applications indicate, none of the algorithms always performs better for all sites and applications. For the EPIGENOMICS workflow, Figures 4.b, 4.c and 4.d, DBCS obtained similar performances to other higher complexity algorithms, always achieving the best result for the `Sophia` site. For the `Lille` site, the results are similar to `Sophia`, DBCS being the best algorithm for the lower budget ($\alpha_B = 0.3$) and producing slightly lower performance than DCA for medium and larger budgets. DBCS is always better than BHEFT, except for a single case in `Lille`, with significantly better performance in most configurations of the deadline and budget. For the MONTAGE workflow, Figures 5.b, 5.c and 5.d, the DCA algorithm consistently obtained higher PSR values, but DBCS obtained comparable PSR values for higher budgets. This is explained by the fact that MONTAGE has a higher average CCR; thus, the deadline imposed,

[5] https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator
[6] http://www.austriangrid.at



(a) EPIGENOMICS workflow

(c) EPIGENOMICS ,Cluster$_{name}$ = Sophia

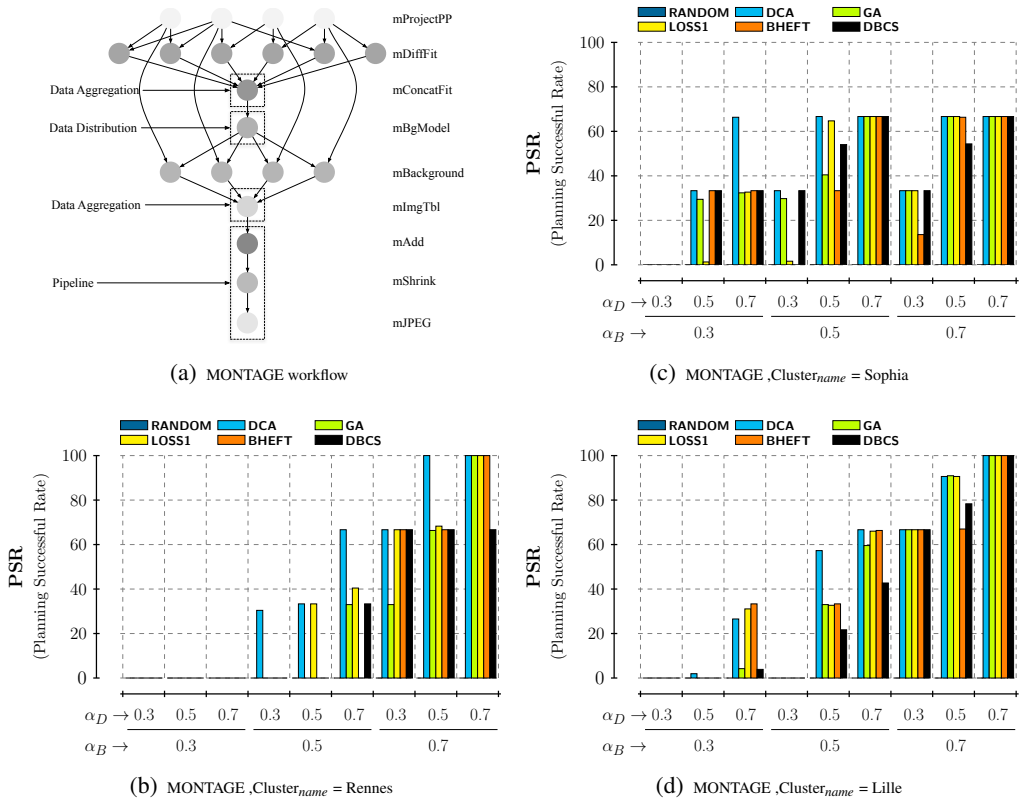
(b) EPIGENOMICS ,Cluster$_{name}$ = Rennes

(d) EPIGENOMICS ,Cluster$_{name}$ = Lille

Figure 4: Normalized Makespan for EPIGENOMICS workflows on GRID5000

based on the critical path, is too restrictive, in particular on `Rennes`. This only affects DBCS, BHEFT and RANDOM because due to their heuristic nature, a moving forward strategy, they do not roll back and change the task assignment. The other algorithms that are search-based strategies may change the task assignment during the solution space search. BHEFT also obtains good performances, comparable to DBCS, in particular for `Sophia` and `Lille`. For the Wien2K algorithm, Figures 6.b, 6.c and 6.d, DCA presents generally better performances, but DBCS achieves the same value for many cases and a PSR value very close to the best algorithm that alternates from DCA, GA and LOSS1. In `Rennes` site, BHEFT produces results only for the higher deadline ($\alpha_D = 0.7$), while DBCS produces very good PSR values for all cases as well as DCA, GA and LOSS1. In `Sophia`, the results are similar to `Rennes` but with lower PSR values. BHEFT produces results for more cases but significantly lower than DBCS. In `Lille`, the results follow the same pattern as in `Rennes` with a PSR value of 100% for all algorithms, except RANDOM in the highest deadline. In the other cases, DBCS is close to the best algorithms, which are DCA and GA, and significantly better than BHEFT.

EPIGENOMICS workflow has the highest rate of successful schedules due to its lower average CCR as well as lower CCR distribution values, which makes it more feasible to find schedule maps for the range of budgets and deadlines. The RANDOM algorithm only produces valid schedules for Wien2K in `Rennes`, with a much lower performance than DBCS, showing that the slightly



(a) MONTAGE workflow

(c) MONTAGE ,Cluster$_{name}$ = Sophia

(b) MONTAGE ,Cluster$_{name}$ = Rennes

(d) MONTAGE ,Cluster$_{name}$ = Lille

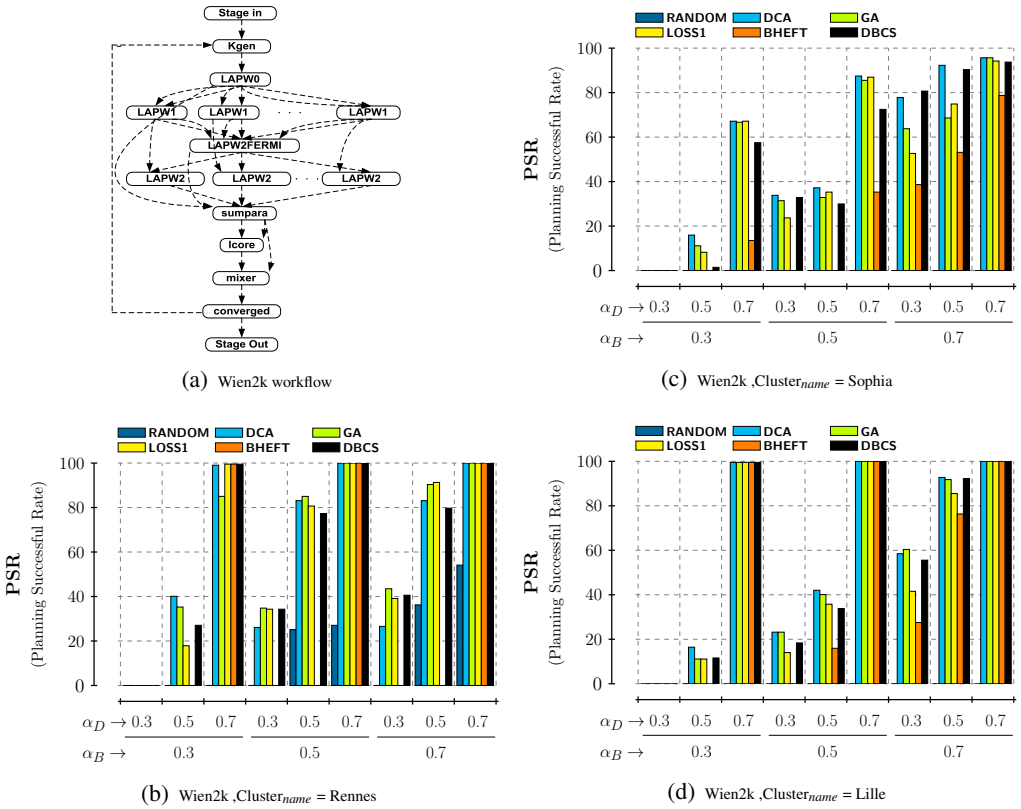
Figure 5: Normalized Makespan for MONTAGE workflows on GRID5000

higher processing time of DBCS compensates. The same conclusion can be drawn in relation to BHEFT, where with a similar scheduling time, DBCS obtains globally better performances.

8.6 Conclusions and Future Work

In this paper, we presented the Deadline-Budget Constrained Scheduling algorithm (DBCS), which maps a workflow to a heterogeneous system constrained to user-defined deadline and budget values. The algorithm was compared with other state-of-the-art algorithms. In terms of time complexity, which is a critical factor for effective usage on real platforms, our algorithm has the lowest time complexity (quadratic time complexity), while other algorithms mostly have cubic or polynomial time complexities. In terms of the quality of results, DBCS achieves rates of successful schedules similar to higher-time complexity algorithms for both random and real application workflows on diverse platforms and also for the range of values of deadline and budget constraints considered in this paper. Compared to other low-time complexity algorithms, namely, BHEFT and RANDOM, DBCS performs, in general, significantly better for the workflows and platforms considered.

In conclusion, we have presented the DBCS algorithm for budget and deadline constrained



(a) Wien2k workflow

(c) Wien2k ,Cluster$_{name}$ = Sophia

(b) Wien2k ,Cluster$_{name}$ = Rennes

(d) Wien2k ,Cluster$_{name}$ = Lille

Figure 6: Normalized Makespan for Wien2K workflows on GRID5000

scheduling, which has proved to achieve a similar performance to higher-time complexity algorithms, namely, DCA, GA and LOSS1, but with a time complexity of the heuristic algorithms, namely, BHEFT and RANDOM, of the order $O(n^2.p)$ for n tasks and p processors.

In future work, we intend to extend the algorithm to consider the dynamic concurrent DAG scheduling problem. This model will allow users to execute concurrent workflows that might not be able to start together but that can share resources so that the total time and cost for the user can be minimized to meet their deadlines and budgets.

Acknowledgments

This work is partially supported by Fundação para a Ciência e Tecnologia, PhD Grant FCT - DFRH - SFRH/BD/80061/2011, and by EU under the COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS).

Chapter 9

A framework for concurrent workflow Constraint-conscious scheduling

Hamid Arabnejad and Jorge G. Barbosa

Special Issue on High Performance Scheduling for Heterogeneous Distributed Systems of FGCS , answered to reviewers ,

Submitted,

abstract

Workflow applications described by directed acyclic graphs (DAGs) present intrinsic parallelism among tasks and their processing time can be optimized by a task parallel approach. However, the optimization process has limitations due to task dependencies. As common resource managers allocate a set of resources to execute a single workflow, in a user centric approach, those resources cannot be fully utilized by a single job, incurring higher costs to the user without obtaining any improvement in the job processing time. In this paper we propose a framework that executes simultaneously several workflow applications, where resources are shared among jobs. A user, when submitting a job, specifies a budget and a deadline for the job, in a range of values specified by the framework, and that can be accomplish with the available infrastructure. The framework is dynamic (on-line), so that it can receive jobs at any moment in time. For concurrent workflow scheduling, several algorithms have been proposed, being most of them off-line solutions. Recent research attempted to propose on-line strategies for concurrent workflows but only addressing fairness in resource sharing among applications while minimizing the execution time. The Multi-Workflow Deadline-Budget scheduling algorithm (MW-DBS) is proposed here to schedule multiple and concurrent workflow applications that may be submitted at different moments in time and with individual user's budget and deadline constraints. We study the scalability of the algorithm with different types of workflows and infrastructures. Experimental results

show that our strategy is able to increase the scheduling success rate of finding a scheduling solution for each job as well as to obtain higher revenue for the provider through a higher rate of completed jobs.

1 Introduction

Many scheduling algorithms have been proposed to optimize the execution of a single workflow application on Heterogeneous Computing Systems (HCS) and with a single Quality of Service (QoS) parameter, usually minimizing the execution time of the application [AB14c, BSB+01, THW02]. However, with the introduction of other factors such as execution cost, more objectives need to be considered based on user's QoS requirements. On the other hand, other studies [BM08] show that the execution of a single DAG on a set of processors leads to a wastage of resources. Although the provider may charge for all processors during the execution time, it is an evidence that the user is paying more than it is able to use.

The framework proposed in this paper intends to reduce costs for the user when running workflow applications as well as to allow that a provider increases the number of successful applications and therefore obtains also higher revenues. In this context, scheduling algorithms must be able to support the scheduling of concurrent workflow applications, that may be submitted at different moments in time and with individual QoS parameter values. Therefore, scheduling frameworks should consider not only a schedule solution map for each single application, but also focus on the overall performance, defined as the success rate of finding a valid solution for all submitted applications.

To address the problem mentioned above, we introduce here a new scheduling strategy, Multi-Workflow Deadline-Budget scheduling algorithm (MW-DBS), for scheduling concurrent workflow applications with multiple QoS constraints, here, time and cost. The MW-DBS algorithm contains two main steps: first, it selects a task from each ready workflow and assigns a priority to each task based on the remaining time to application deadline; and second, for the high priority task, selects a suitable resource based on a quality measure computed to each resource, based on the job QoS parameters and provider profit.

The main contributions of this paper are: a) proposal of a framework for concurrent job schedule; b) a new low time complexity scheduling algorithm to deal with concurrent workflows constrained to time and cost. Our algorithm, increases the provider profit by increasing the acceptance rate of successful applications; c) a realistic simulation that considers a bounded multi-port model in which bandwidth is shared by concurrent communications; and d) we present results for randomly generated graphs, as well as for real-world applications.

The remainder of the paper is organized as follows. In the next section, we describe the framework and the computational model. Then, we describe the related work, followed by the details of our MW-DBS scheduling algorithm. Then we show the benefits of the MW-DBS by comparison and simulation. Finally, we present conclusions and directions for future work.

2 Scheduling Framework

In the proposed framework, presented in Figure 1, applications can be submitted by any user and at any moment in time, into the system. The aim of this structure is to schedule tasks from

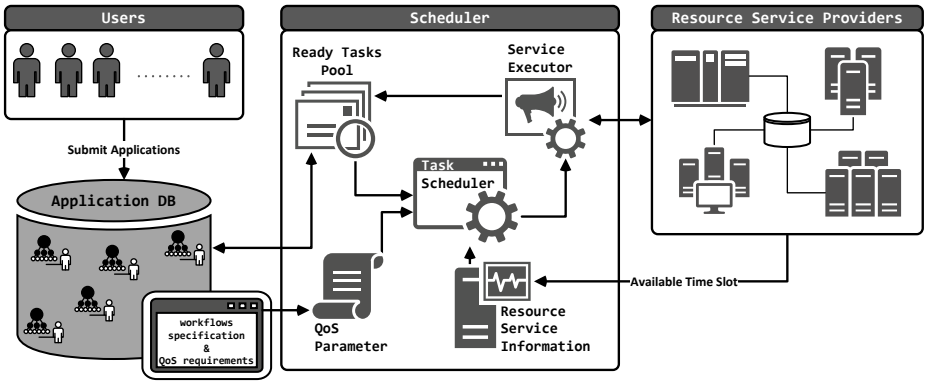


Figure 1: A general view of the Framework for concurrent workflow scheduling

the workflow applications into available resources, constrained by user's QoS demands. Submitted applications are collected by Application Data Base (DB) with their specifications and QoS requirements. In this architecture, a *Framework Scheduler* (FS) receives applications from Application DB and generates tasks-to-resource maps for all applications. To make a proper decision in resource selection strategy for each application's task, FS needs the status information of available resources. The *Resource Service Information* is responsible for observing and collecting information about the current state of resources such as resource capacities, memory size, network bandwidth, availability, functionality and especially the available time slots for processing tasks. The Globus Monitoring and Discovery System (MDS) [CFFK01] is an example of Resource Service Information. Besides resource status, the list of ready-to-execute tasks and user's QoS requirements for each application are also necessary for making a feasible schedule. The *Ready Task pool* module collects tasks which are ready to execute among accepted workflow applications in Application DB. A task is considered ready when its parents are executed. The *QoS Parameter* module contains the user's QoS requests for their workflow applications. These two modules are used to select the task and related application at each step of the scheduling process. The *Service Executor* module implements the task assignment by submitting each task to the selected resource, and monitors their execution. The Globus GRAM (Grid Resource Allocation and Management) [CFK+98] is a good example of service executor module.

Finally, *Task scheduler* finds the suitable task-to-processor map for executing each ready task based on its QoS attributes and on the detailed information of each service.

2.1 Application model

A workflow application can be represented by a Directed Acyclic Graph (DAG) in which nodes represent tasks and edges represent task and data dependencies. A dependency ensures that a child node cannot be executed before all its parent tasks finish successfully and transfer the required child input data. The overall finish or completion time of an application is usually called the

schedule length or *makespan*. The model to obtain the total execution cost for an application may consider computation costs, storage costs and data transfer costs.

A DAG can be modeled by a tuple $G =< T, E >$, consisting of a set of tasks, $T = \{t_1, t_2, \ldots, t_n\}$ where n is the number of tasks in the workflow, and a set of dependencies among the tasks $E = \{< t_a, t_b >, \ldots, < t_x, t_y >\}$ where t_a and t_x are parent tasks of t_b and t_y, respectively. The $\overline{C}_{(t_i \to t_j)}$ represents the average communication time between the parent tasks t_i and child task t_j which is calculated based on the average bandwidth and latency among all processor pairs. 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 [THW02]. Due to heterogeneity, each task may have a different execution time on each processor.

In a given DAG, a task with no predecessors is called an *entry task* and a task with no successors is called an *exit task*. We assume that the DAG has exactly one entry task t_{entry} and one exit task t_{exit}. If a DAG has multiple entry or exit tasks, a dummy entry or exit task with zero weight and zero communication edges is added to the graph.

2.2 Task execution cost and time model

We consider there is a central storage connected to all resources in order to keep all required files. In this model, in addition to the execution time of task t_i on processor $\acute{p}$ ($ET(t_i, \acute{p})$), the target processor $\acute{p}$ will be occupied by task t_i during its data input and output file transfer operations, into and from central storage. Therefore, Time Reservation (TR) for executing task t_i on processor $\acute{p}$ is defined as:

$$TR(t_i, \acute{p}) = T_{in}(t_i) + ET(t_i, \acute{p}) + T_{out}(t_i) \tag{1}$$

where $T_{in}(t_i)$ and $T_{out}(t_i)$ are the transfer time of all input and output files required for task t_i execution, and are defined as:

$$T_{in}(t_i) = \sum_{t_p \in pred(t_i)} \overline{C}_{(t_p \to t_i)} \tag{2}$$

$$T_{out}(t_i) = \sum_{t_c \in succ(t_i)} \overline{C}_{(t_i \to t_c)} \tag{3}$$

where $pred(t_i)$ and $succ(t_i)$ denote the set of immediate predecessors and immediate successors of task t_i, respectively. Unlike previous research that considered only the maximum transfer time of the input or output data, in this paper, we consider the sum of all transfer time from parents/to children for each task. In realistic platforms each concurrent file cannot use full bandwidth. In this paper, the total time of serial transmission are considered as transfer time of input/output files, since it would be equivalent to send all files in parallel on a shared link. Also, we assume each processor has its own Network Interface Card (NIC) to handle packet flow control between processors and central storage. In this case, the transmission time can be performed and overlapped with

the computation. In this paper, we assume that only output transmission time can be performed in parallel with another task execution. Thus, the Time Reservation (TR) can be simplified as:

$$TR(t_i, \acute{p}) = T_{in}(t_i) + ET(t_i, \acute{p}) \tag{4}$$

Based on these definitions, we can compute the Finish Time of task t_i on processor $\acute{p}$, $FT(t_i, \acute{p})$, considering the execution time of itself and its predecessors as:

$$FT(t_i, \acute{p}) = \begin{cases} TR(t_i, \acute{p}) & , pred(t_i) = \emptyset \\ \max\limits_{t_p \in pred(t_i)} & \left\{ FT(t_p) + TR(t_i, \acute{p}) \right\} \\ & , pred(t_i) \neq \emptyset \end{cases} \tag{5}$$

The *makespan* denotes the finish time of the last task of the workflow and is defined as:

$$\text{DAG}_{makespan} = FT(t_{exit}) \tag{6}$$

$FT(t_i)$ denotes the Finish Time of task t_i on the processor assigned by the scheduling algorithm.

The economic cost depends on three parameters: a) the total occupied time of the resource which starts by transferring input files for the task, then its execution time and, finally, the time used by the processor to transfer output files; b) the cost of transferring data (input/output) required for the task; and c) the data storage cost of the task. In the case of intra-cluster data transference, as in this paper, zero monetary costs are considered for communications between tasks and task storage usage. So, the financial cost of executing task t_i on target processor $\acute{p}$ is simplified to $Cost(t_i, \acute{p}) = TR(t_i, \acute{p}) \times Price(\acute{p})$ where $Price(\acute{p})$ denotes the price of processor per time unit.

$TotalCost$ is the overall cost for executing an application and is defined as:

$$\text{DAG}_{Cost} = \sum_{t_i \in T} AC(t_i) \tag{7}$$

where $AC(t_i)$ is defined as Assigned Cost of task t_i. After assigning a processor p_{sel} to execute task t_i, the assigned cost is equal to $AC(t_i) = Cost(t_i, p_{sel})$.

2.3 Scheduling objectives

For a given scenario which includes a set of workflow applications that may be submitted at different moments in time by users with individual time and cost constraints, the objective of our workflow scheduling strategy is to find a schedule map between tasks and resources for each workflow application that can meet its user-defined QoS constraints, i.e, the completion time of the workflow must not be larger than the user defined time constraint (DEADLINE), and the execution cost must not be larger than the user defined total cost (BUDGET). Due to various independent applications, 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 [KA99]. These

values of DEADLINE and BUDGET for each workflow application should be negotiated between the user and the provider in a range of feasibility values so that there are feasible scheduling solutions. Note that there is not any optimization function in the formulation, so that any scheduling solution that match the constraints, for a given workflow, is a plausible solution.

3 Related Work

In general, scheduling algorithms on heterogeneous computing systems have focused on single workflow applications. In contrast, multiple workflow scheduling has not received much attention. In [ZH14] it was proposed a scheduling algorithm in order to optimize performance and cost for Workflows in the Cloud. Bochenina in [Boc14] introduced a strategy for mapping the tasks of multiple workflows with different deadlines on the static set of resources. Kumar et al. [KR12] proposed a time and cost optimization algorithm for hybrid clouds. Using a schedule map of a workflow, authors in [JHC+11] proposed a method to minimize the total execution time of a scheduling solution for concurrent workflows in HPC cloud. In [BM10a] four strategies, which differ in the ordering of tasks, for scheduling multiple workflows on Grids are discussed. Zhao and Sakellariou in [ZS06] proposed to merge multiple workflow application into a single DAG which can be scheduled by traditional DAG scheduling algorithms in order to minimize the overall makespan and achieve fairness, defined based on the slowdown experienced by each DAG due to competition for resources with other DAGs. But most of these strategies are designed for off-line workflow scheduling which imposes limitations on the management of a dynamic system where users can submit jobs at any moment in time, i.e, they scheduled only available and submitted workflows at this time and after a schedule is produced and initiated, no other workflow is considered.

There are few on-line algorithms proposed specifically to schedule concurrent workflows with the aim of improving individual QoS requirements. In [YS08] and [HHW11] there were proposed two algorithms namely RANK_HYBD and OWM (Online Workflow Management), to schedule multiple online workflows and targeting to have lower average makespan and turnaround time for submitted workflows at different instants of time and by different users. In [AB12a, ABS14] authors proposed the fairness dynamic workflow scheduling (FDWS) algorithm. The main objective of FDWS algorithm is to reduce the individual turnaround time for each workflow application in the system. The FDWS algorithm focuses on the QoS experienced by each application (or user) by minimizing turnaround time, while RANK_HYBD and OWM algorithms try to reduce the average completion time of all workflows.

Xu et al.[XCWB09] propose a multiple QoS constrained scheduling strategy of multi-workflows (MQMW) for cloud computing. The MQMW algorithm minimizes the makespan and cost of workflows which can be submitted and start at any time. In [AB14b], authors proposed on-line strategies for concurrent workflows that extend the former concurrent on-line scheduling algorithms by considering fairness resource sharing constrained to the user defined budget and optimize the turnaround time, for each workflow application.

However, our approach here is different from the algorithms described above, in that we simultaneously consider time and cost as constraints of the scheduling problem and do not perform optimization. The scheduling framework proposed in this paper is the first multiple online workflow scheduling system that simultaneously considers user's budget and deadline constraints for concurrent workflow scheduling in heterogeneous computing systems.

4 Concurrent Scheduling Algorithm

In this section, we present the Multi-Workflow Deadline-Budget scheduling algorithm (MW-DBS), an improved version of [AB15], which aims to find a feasible schedule within an individual BUDGET and DEADLINE constraints for each submitted application. The common scheduling objective of concurrent applications is to increase the number of successful applications, but in addition to this objective, the proposed framework tries to increase the revenue of the provider by giving higher priority to jobs with higher budgets.

Generally, in most on-line scheduling system, without in advance reservation, the scheduler is called when an executing task finishes and there is at least one free available processor to execute new tasks. Like other online scheduler, MW-DBS consists of two main phases, namely a *task selection* phase and a *processor selection* phase as described next.

4.1 Task selection

The MW-DBS algorithm should select a suitable task to be executed among all tasks from *ready tasks* pool which is filled by the ready tasks belonging to each submitted and unfinished workflow application. In general, two methods are used to fill the *ready tasks* pool: a) first collect a single ready task from each workflow [AB12a, AB14b, ABS14, HHW11], and b) insert all ready tasks [YS08] belonging to each unfinished workflow application. Adding all ready tasks from each available workflow leads to an unfair strategy because the high number of ready tasks may cause that some workflow applications may not participate in the current scheduling round. In MW-DBS algorithm, to fill the *ready tasks* pool, a single ready task with highest priority upward rank[THW02] ($rank_u$) is selected and added to the list. From the set of ready tasks we need to select one to schedule based on the QoS parameters defined to each application.

The key point in task selection phase is which task should be selected for scheduling among all ready-to-execute tasks. So, a priority assignment strategy based on our QoS parameters is defined to assign a secondary rank to each task in *ready tasks* pool. Since we are dealing with both time and cost factors, the secondary rank assignment strategy should considers both measures. Additionally, results in [AB12a, ABS14] show that taking into account the workflow history of scheduled tasks, leads to a better performance.

In this paper, we propose a new strategy ($rank_D$) to assign a secondary priority to each task t_i belonging to workflow j in the *ready tasks* pool, defined by equation 8:

$$rank_D(t_{i,j}) = Cost^R_{t_{i,j}} \times \frac{1}{Time^R_{t_{i,j}} \times PRT_j} \tag{8}$$

where $Cost^R_{t_{i,j}}$ is the relative Cost Ratio of task t_i from workflow j and it is calculated as:

$$Cost^R_{t_{i,j}} = \frac{B_j}{CA_j} \tag{9}$$

where B_j is the cost constraint value (user's BUDGET) and CA_j is defined as Cheapest Assignment, i.e. all tasks from application j scheduled to cheapest processors. $Time^R_{t_{i,j}}$ is the relative Time Ratio of task t_i from workflow j and it is calculated as:

$$Time^R_{t_{i,j}} = \frac{D_j - SD(t_i)}{D_j} \tag{10}$$

where D_j is the time constraint value (user's DEADLINE) and $SD(t_i)$ is defined as Sub-Deadline assigned to task t_i. We applied the common and direct project planning sub-deadline distribution strategy. $SD(t_i)$ is computed recursively by traversing the task graph upwards, starting from the exit task. Due to heterogeneity, sub-Deadline can be defined in several different forms. Here, we consider the average execution time of the current task, as shown by Eq(11):

$$SD(t_i) = \min_{t_{child} \in succ(t_i)} \left[SD(t_{child}) - \overline{TR}(t_{child}) \right] \tag{11}$$

where $\overline{TR}$ is defined as the average time reservation of task t_i among available processors. For the exit task, the sub-deadline is equal to the user defined deadline ($SD(t_{exit}) = D_j$). PRT_j is the Percentage Remaining Tasks of workflow j and calculated as:

$$PRT_j = \frac{Unscheduled\ tasks\ of\ workflow\ j}{Total\ tasks\ of\ workflow\ j} \tag{12}$$

The $rank_D$ priority value contains two major factors: a) the cost parameter which gives higher priority to the submitted and unfinished workflow applications that have higher budget ratio in order to maximize the provider profit, and b) the time parameter which contains two time measures, $Time^R$ and PRT. The first has responsibility of assigning higher priority to workflows which have lower sub-deadline. The second ensures that a workflow with few unscheduled tasks has higher priority. Finally, the task with highest $rank_D$ in *ready tasks* pool is selected to be schedule in the next phase.

4.2 Processor selection

The processor selection phase has the responsibility for selecting an affordable resource for the current task (t_{curr}) and it is repeated until there is no more tasks left in *ready tasks* pool. A new

strategy for processor selection phase based on QoS requirements is proposed. In order to control the consumed cost and time, a bound value for each factor is needed. Next, we describe first bound values for cost and time, and then we present a new strategy for processor selection.

The Cost bound value ($Cost_{Bound}$) is a limitation on budget consumption by each task based on used budget, by previously scheduled tasks, and available budget for the current task that can be consumed by its assignment:

$$Cost_{Bound}(t_{curr}) = Cost_{min}(t_{curr}) + \Delta_j^{Cost} \tag{13}$$

where $Cost_{min}$ denotes the minimum execution cost of the current task among all processors and $\Delta_j^{Cost} = RB_j - RCA_j$ represents the spare budget defined as the difference between unconsumed budget and cheapest cost assignment for unscheduled tasks for workflow j which task t_{curr} belongs to. The Remain unconsumed Budget of workflow j (RB_j) has an initial value equal to the available user budget (B_j) and is updated at each step after selecting the processor for t_{curr} as shown in Eq(14), where $AC(t_{curr})$ is the Assigned Cost. Similarly, RCA_j is defined as Remaining Cheapest Assignment of workflow j with initial value equal to Cheapest Assignment (CA_j) and updated by Eq(15).

$$RB_j = RB_j - AC(t_{curr}) \tag{14}$$

$$RCA_j = RCA_j - Cost_{min}(t_{curr}) \tag{15}$$

All free available processors are filtered by $Cost_{Bound}(t_{curr})$ in order to guarantee that the application can be executed without exceeding the budget constraint. For the current assignment, we defined this set of acceptable processors as $P_{admissible}$. In the most restricted case, only the cheapest processors are considered. Otherwise, no feasible schedule exists under the user defined budget.

For the Time bound value it is used the sub-Deadline (SD), introduced in *task selection* phase, and it is a soft limitation as in most deadline distribution strategies for a fixed number of available resources [YRB09]; if the scheduler cannot find a processor that satisfy the sub-deadline for the current task, the processor that can finish the current task at earliest time is selected.

The processor selection phase is based on a quality measure assigned to each processor that combines the QoS factors. Once there in no optimization step, each resource is evaluated in terms of the processing time and cost for the current task t_{curr}. Two quantities are defined, namely, Time Quality ($Time_Q$) and Cost Quality ($Cost_Q$), on each admissible processor $\acute{p} \in P_{admissible}$, shown in (16) and (17), respectively. Both quantities are normalized by their maximum values.

$$Time_Q(t_{curr}, \acute{p}) = \begin{cases} 1 - \dfrac{FT(t_{curr}, \acute{p})}{SD(t_{curr})} & \text{if } FT(t_{curr}, \acute{p}) < SD(t_{curr}) \\ \\ 1 - \dfrac{FT(t_{curr}, \acute{p})}{FT_{min}(t_{curr})} & \text{otherwise} \end{cases} \tag{16}$$

$$Cost_Q(t_{curr}, \acute{p}) = \begin{cases} 1 - \dfrac{Cost(t_{curr}, \acute{p})}{Cost_{max}(t_{curr})} & \text{if } FT(t_{curr}, \acute{p}) < SD(t_{curr}) \\ \\ 1 & \text{otherwise} \end{cases} \tag{17}$$

Where $FT_{min}(t_{curr})$ and $Cost_{max}(t_{curr})$ denote the minimum finish time and the maximum execution cost of current task among all available processors.

Finally, to select the most suitable processor for t_{curr}, the Quality measure (Q) for each processor $\acute{p} \in P_{admissible}$ is computed as shown in Eq(18) and the processor with highest Q is selected.

$$Q(t_{curr}, \acute{p}) = Time_Q(t_{curr}, \acute{p}) \times Cost_Q(t_{curr}, \acute{p}) \tag{18}$$

Resources for which the finish time is lower than the deadline ($FT(t_{curr}, \acute{p}) < SD(t_{curr})$), $Time_Q$ measures how much the finish time of current task on a processor is closer to the task sub-deadline. The processor with higher $Time_Q$ has higher possibility to be selected. Otherwise, $Time_Q$ assumes a negative or zero value, reducing the processor quality value. Processors with a lower cost to execute t_{curr} have a higher $Cost_Q$, increasing their quality measure. However, for the processors that do not cover the sub-deadline, the processor with lowest finish time should be selected regardless of what cost it has. In this case, $Cost_Q$ is set to 1, being the processors quality only influenced by $Time_Q$. It should be noted that in both cases, the tested processors are selected from admissible processor list so that it can be guaranteed that the application can be executed without exceeding its budget constraint.

The MW-DBS algorithm is shown in Algorithm 1. First, all ready-to-execute tasks in *ready tasks* pool are ranked by $rank_D$ priority value (Eq.8). To fill *ready tasks* pool, the framework collects a single ready-to-execute task with highest primary rank value $rank_u$ [THW02] from each submitted and unfinished workflow application. Until there is at least one ready and unscheduled task in *ready tasks* pool and free available processors, the current task t_{curr} is selected and its quality measure Q (Eq.18) is calculated among all admissible processors ($P_{admissible} \subset P_{free}$). Then, the current task t_{curr} is assigned to processor P_{sel} that has the highest quality measure. Then, the Remaining unconsumed Budget (RB) and Remaining Cheapest Assignment (RCA) are updated for workflow j where task t_{curr} belongs to.

In terms of time complexity, MW-DBS requires the computation of the upward rank ($rank_u$) and Sub-DeadLines (SD) for each task that have complexity $O(n.p)$, where p is the number of available resources and n is the number of tasks in the workflow application. In the processor selection phase, to find and assign a suitable processor for the current task, the complexity is $O(n.p)$ for calculating FT and $Cost$ for current task among all processors, plus $O(p)$ for calculating the Quality measure. The total time is $O(n.p + n(n.p + p))$, where the total algorithm complexity is of the order $O(n^2.p)$.

Algorithm 1 MW-DBS algorithm

1: **for all** $t_{i,j} \in$ *Ready Tasks* pool **do**
2: Assign a priority rank $rank_D(t_{i,j})$
3: **end for**
4: $P_{free} \leftarrow$ free processors $\acute{p} \in P$
5: **while** $(Ready\ Tasks \neq \phi\ \&\ P_{free} \neq \phi)$ **do**
6: $t_{curr} \leftarrow$ task with highest $rank_D$
7: **for all** $\acute{p} \in P_{admissible}$ **do**
8: calculate Quality measure $Q(t_{curr}, \acute{p})$
9: **end for**
10: $P_{sel} \leftarrow$ Processor $\acute{p}$ with highest Q
11: Assign current task t_{curr} to Processor P_{sel}
12: Update RB_j and RCA_J
13: $P_{free} \leftarrow P_{free} - P_{sel}$
14: Remove Task t_{curr} from *Ready Tasks* pool
15: **end while**

5 Experimental Results

This section presents performance comparisons of the MW-DBS algorithm. We implemented modified versions of FDWS[AB14b], MIN-MIN and MAX-MIN. The MIN-MIN and MAX-MIN algorithms have been studied extensively in the literature [MAS+99], and therefore we implemented an online version of these algorithms for our problem. We use the version of FDWS algorithm proposed in [AB14b], namely FDWS2.

In the modified versions of these three algorithms, instead of considering all processors to compute the finish time of current task, processors are filtered ($P_{admissible}$) based on the cost limitation value defined by Eq.13 and the processor that allows the lowest finish time among all affordable processors is selected.

5.1 Simulation platform

We resorted to simulation to evaluate the algorithms discussed in the previous sections. Simulation allows us to perform a statistically significant number of experiments for a wide range of application configurations in a reasonable amount of time. We used the SIMGRID toolkit[1] [CGL+14] 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 heterogeneous computing system, such as the grid platform considered in this work.

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,

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

but each communication flow is limited by the bandwidth of the traversed route and communications using a common network link have to share bandwidth. In this experiments, we connected all processor over one shared bandwidth.

We consider three sites that comprise multiple clusters and having different CPU power composition. The Rennes site has normal distribution of CPU power among its clusters, the Sophia site contains a higher number of low speed processors, while Lille site has a higher number of fast processors.

Table 13 provides the name of each site, along with the set of clusters that compose the site. For each cluster, it presents the total number of processors ($\#CPU_{Total}$), processing speed expressed in GFlop/s and processor cost. $\#CPU_{used}$ shows the number of processors used from each cluster for 8 and 16 processors configurations.

Site	Cluster	#CPU Total	#CPU used		Power (GFlop/s)	Cost ($)
rennes	paradent	64	3	7	21.496e9	0.61$
	paramount	33	2	3	12.910e9	0.31$
	parapide	25	1	2	30.130e9	1.00$
	parapluie	40	2	4	27.391e9	0.87$
sophia	helios	56	3	6	7.732E9	0.16$
	sol	50	3	5	8.939e9	0.19$
	suno	45	2	5	23.530e9	0.70$
lille	chicon	26	2	4	8.962e9	0.19$
	chimint	20	2	4	23.531e9	0.70$
	chinqchint	46	4	8	22.270e9	0.64$

Table 13: Description of the Grid5000 clusters from which the platforms used in the experiments were derived

5.2 Budget and deadline parameters

To evaluate the scheduling algorithm of the proposed framework, we need to define values for deadline and budget constraints for each individual application. For each tested site these parameters are computed independently of the number of CPUs on that site.

To specify the deadline parameter, we define the min_D and max_D as the lowest and highest execution time of the application, as shown in Eq.19 and Eq.20.

$$Min_D = \sum_{t_i \in CP} \left(TR_{min}(t_i) \right) \tag{19}$$

$$Max_D = \sum_{t_i \in CP} \left(TR_{max}(t_i) \right) \tag{20}$$

where CP is the critical path. Please note that both of these values are defined based on an infinite number of CPUs. For a bounded number of resources, the minimum and maximum execution

times may be very optimistic and not reachable. The processing time range is defined based on the critical path, in order to specify a deadline based on the lower bound of the makespan.

In the same way, to specify a budget constrain, we need to estimate the maximum and the minimum cost to obtain a range of feasible budgets to execute the application. We defined the Min_B and Max_B as absolute lowest and highest possible costs for executing the application, that are calculated by summing the maximum and the minimum execution costs for each task, as shown in Eq.21 and 22, respectively.

$$Min_B = \sum_{t_i \in T} \left(Cost_{min}(t_i) \right) \tag{21}$$

$$Max_B = \sum_{t_i \in T} \left(Cost_{max}(t_i) \right) \tag{22}$$

With these highest and lowest bound values, we define a deadline (D) and a budget (B) constraints for the workflow application j as described in Eq(23) and Eq(24):

$$\mathrm{D}_j = min_D + \alpha_D \times \left(max_D - min_D \right) \tag{23}$$

$$\mathrm{B}_j = min_B + \alpha_B \times \left(max_B - min_B \right) \tag{24}$$

A lower deadline parameter α_D results in a lower deadline D for the application, which would require resources with higher capacity and cost, therefore making it difficult to meet its time constraint with low budget. Thus, to define range for budget and deadline parameters, in order to adopt them to a more realistic situation, α_D is selected in the range $[0 \ldots 1]$. If $\alpha_D < 0.5$, then the budget parameter α_B is selected in the range of $[0.5 \ldots 1]$, otherwise the range is $[0 \ldots 0.5]$.

5.3 Workflow structure

Workflow applications can be categorized as *Synthetic* and *Real World* applications. In order to generate dynamic and concurrent scenarios, we consider 50 workflows in each scenario, that arrive with time intervals that range from 5%, 15% and 30% of the deadline of the previous one, i.e., a new workflow is inserted when the corresponding percentage of the deadline from the last workflow has elapsed. The total number of scenarios is 5000 and each scenario is tested for 9 different combinations of deadline and budget parameters.

5.3.1 Synthetic workflow applications

The Synthetic workflow category was created by the synthetic DAG generation program[2]. The computational complexity of a task is modelled 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)

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

where a is picked randomly between 2^6 and 2^9. As a result, different tasks exhibit different communication/computation ratios.

The DAG generator program defines the DAG shape based on four parameters: *width*, *regularity*, *density*, and *jumps*. In our experiment, for synthetic DAG generation, it was considered a number of tasks in the range $n = [40 \ldots 120]$, a fat of 0.1, 0.4, 0.8, a $regularity$ and $density$ of 0.2 and 0.8, and a $jump$ in the range $[1 \ldots 4]$.

5.3.2 Real world workflow applications

To evaluate the algorithms on a standard and real set of workflow applications, a set of workflows were generated using the code developed in Pegasus toolkit[3]. Four well know structures were chosen [JCD+13], namely: Montage, CyberShake, Epigenomics and LIGO. For each workflow type, we generated 300 DAGs with a number of tasks randomly taken in the range $[30 \ldots 100]$.

5.4 Performance metrics

To evaluate the framework, we consider two distinct metrics to evaluate performance improvement in both perspectives, i.e. *users* and *service providers*.

5.4.1 Planning successful rate

The metric to evaluate a dynamic scheduler of independent workflows, must represent the individual successful rate of finding a valid schedule map for each workflow application in the scenario, in order to measure the QoS experienced. We consider the Planning Successful Rate (PSR), as expressed by Eq (25):

$$\text{PSR} = 100 \times \frac{\textit{Successful Planning}}{\textit{Total Number in experiment}} \tag{25}$$

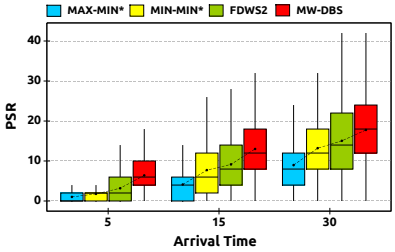
The PSR metric represented the QoS experienced by the users related to the finish time of each user application.

5.4.2 Profit

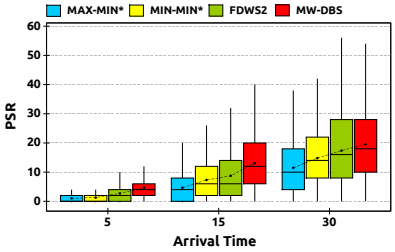
For the providers, the critical parameter is profit, which is defined as the total cost of successful workflow applications. Commonly, a higher number of successful applications may not lead to higher profits. To calculate the profit metric for an algorithm, first we define the total cost of the successful applications as $Total_{cost}(alg)$. Then, the profit metric for each algorithm is calculated as the ratio of the total cost achieved by the algorithm and the maximum total cost among all algorithms ($max(Total_{cost})$) for each scenario.

$$\text{Profit}_{alg} = \frac{Total_{cost}(alg)}{max(Total_{cost})} \tag{26}$$

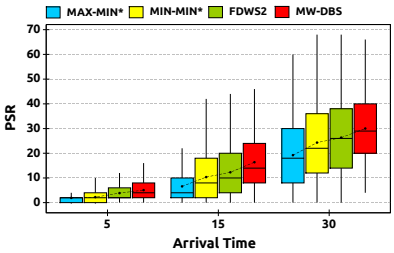
[3] https://confluence.pegasus.isi.edu/



(a) lille

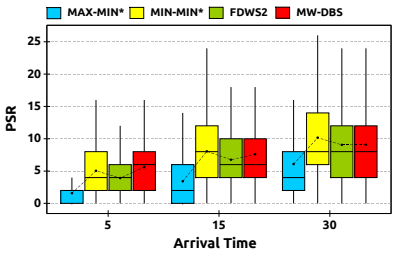


(b) rennes

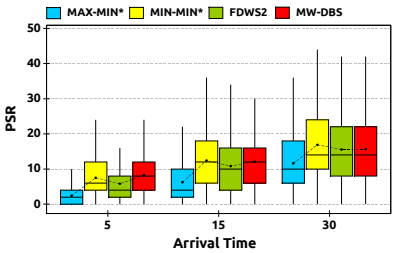


(c) sophia

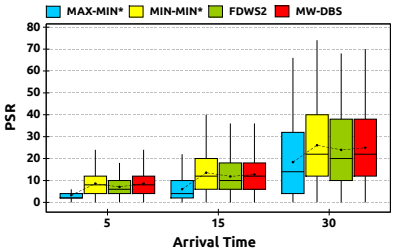
Figure 2: PSR values for synthetic workflow applications and different interval arrival time



(a) lille



(b) rennes



(c) sophia

Figure 3: PSR values for Real World workflow applications and different interval arrival time

5.5 Results and Discussion

In this section, we compare the MW-DBS algorithm with FDWS2[AB14b] and the modified versions of MIN-MIN and MAX-MIN , called MIN-MIN* and MAX-MIN* in the figures. We consider a low number of processors compared to the number of DAGs to analyze the behavior of the algorithms in a higher concurrent environment. The maximum load configuration is observed for 8 processors and 50 DAGs. Also, the filled circle connected by dot lines in the figures represent the average values.

Figure 2 and 3 show the average PSR obtained on three different sites with CPU configurations per site equal to 8 and 16 processors, for different arrival times, and for synthetic and real world applications. In general, for both cases, the MW-DBS algorithm shows better performance in all sites. Also, the MAX-MIN* algorithm yielded poorer results than the MIN-MIN* algorithm in most cases.

The main advantage of MW-DBS algorithm occurred for low time intervals, that shows significant performance improvement rather than other algorithms. Increasing the time intervals between the DAGs arrival times reduces the concurrency, and thus, the improvements are less significant.

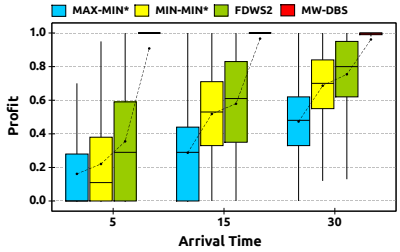
In another view, comparing results of two different workflow datasets, the PSR value of synthetic workflow applications is higher than for real world workflows. The reason can be explained based on workflow structure. Real world workflows have higher parallelism and are more balanced than synthetic ones, so the defined user time constraint value which is calculated based on the critical path (Eq.23) may not be achieved, i.e, the MW-DBS algorithm scheduled some tasks of the workflow and after some time it realizes that the deadline cannot be met resulting in a failed workflow. An alternative strategy is to have a pre-scheduling method to remove unfeasible workflows, but it would increase the require time to make scheduling decisions.

From the service provider's viewpoint, the major key is how much revenue is made. Figures. 4 and 5 shows the profit value of MW-DBS algorithm over other compared algorithms. As it is shown, the proposed scheduling algorithm presents higher performance for profit in addition of higher user satisfaction, higher PSR, in the majority of the cases.

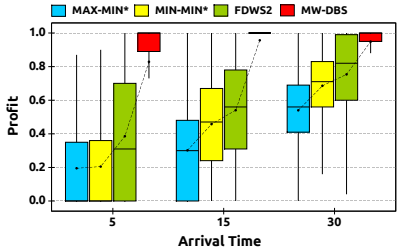
The alternative to the framework that reserves a set of processors to each job, presents a PSR and profit near zero for the high concurrent conditions considered in the experiment, and therefore, are not presented graphically.

6 Conclusion

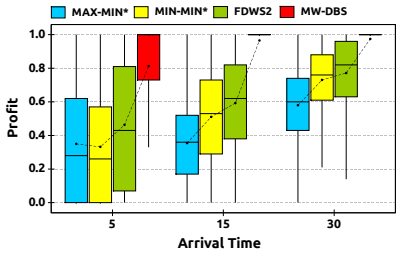
In this paper, we have presented a framework for dynamic scheduling of concurrent workflows with two conflicting QoS requirements. To the best of our knowledge, there is no previous research that deal with multiple workflow scheduling that are submitted at different moments in time and that are based on the two conflicting QoS parameters, namely, time and cost constraint at the same time. The main advantage of the proposed framework is the heuristic scheduler with low complexity, which make it suitable for usage in real grid infrastructures. In terms of result



(a) lille

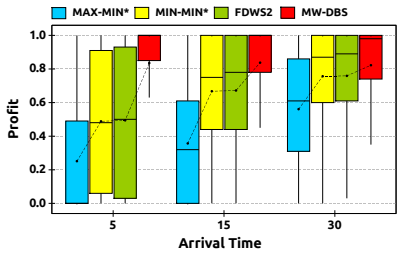


(b) rennes

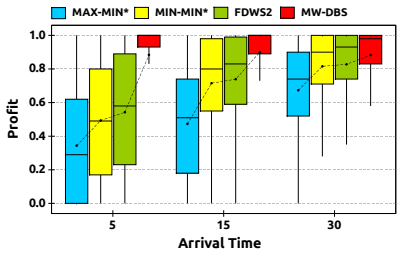


(c) sophia

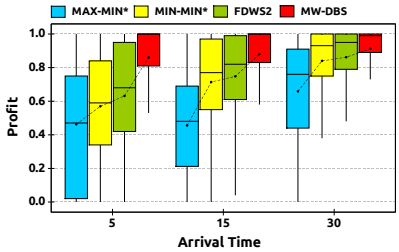
Figure 4: Profit values for synthetic workflow applications for different interval arrival time



(a) lille



(b) rennes



(c) sophia

Figure 5: Profit values for Real World workflow applications for different interval arrival time

accuracy, we used a simulator with a realistic model of the computing platform and with shared links, as occurs in a common grid infrastructure.

The scheduler algorithm, MW-DBS, maps each workflow to a heterogeneous system constrained to a user-defined budget and time. The MW-DBS algorithm obtains better performances in almost all presented cases, synthetic and real world applications, specially for lower arrival time. The increase on the PSR implies a higher revenue for the provider that charges based on the successful completed jobs.

In future work, we intend to extend the algorithm to consider more QoS parameter such as energy, reliability and fault tolerance in grid environments.

7 Acknowledgment

This work was supported in part by the Fundação para a Ciência e Tecnologia, PhD Grant FCT - DFRH - SFRH/BD/80061/2011.

References

[AB12a] Hamid Arabnejad and Jorge Barbosa. Fairness resource sharing for dynamic workflow scheduling on heterogeneous systems. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 633–639. IEEE, 2012.

[AB12b] Hamid Arabnejad and Jorge G Barbosa. Performance evaluation of list based scheduling on heterogeneous systems. In *Euro-Par 2011: Parallel Processing Workshops*, pages 440–449. Springer, 2012.

[AB14a] Hamid Arabnejad and Jorge G Barbosa. A budget constrained scheduling algorithm for workflow applications. *Journal of Grid Computing*, 12(4):665–679, 2014.

[AB14b] Hamid Arabnejad and Jorge G Barbosa. Budget constrained scheduling strategies for on-line workflow applications. In *Computational Science and Its Applications–ICCSA 2014*, pages 532–545. Springer, 2014.

[AB14c] Hamid Arabnejad and Jorge G Barbosa. List scheduling algorithm for heterogeneous systems by an optimistic cost table. *Parallel and Distributed Systems, IEEE Transactions on*, 25(3):682–694, 2014.

[AB15] Hamid Arabnejad and Jorge G. Barbosa. Multi-workflow qos-constrained scheduling for utility computing. In *18th IEEE Int. Conference on Computational Science and Engineering (CSE)*, pages 1–8. IEEE, 2015.

[ABK98] Abdel Krim Amoura, Evripidis Bampis, and Jean-Claude Konig. Scheduling algorithms for parallel gaussian elimination with communication costs. *Parallel and Distributed Systems, IEEE Transactions on*, 9(7):679–686, 1998.

[ABS14] Hamid Arabnejad, Jorge Barbosa, and Frédéric Suter. Fair resource sharing for dynamic scheduling of workflows on heterogeneous systems. *High-Performance Computing on Complex Environments*, pages 145–167, 2014.

[Ama] Amazon. @ONLINE. http://aws.amazon.com/ec2.

[ANE12] Saeid Abrishami, Mahmoud Naghibzadeh, and Dick HJ Epema. Cost-driven scheduling of grid workflows using partial critical paths. *Parallel and Distributed Systems, IEEE Transactions on*, 23(8):1400–1414, 2012.

[ANE13] Saeid Abrishami, Mahmoud Naghibzadeh, and Dick HJ Epema. Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds. *Future Generation Computer Systems*, 29(1):158–169, 2013.

[ASMH00] Shady Ali, Howard Jay Siegel, Muthucumaru Maheswaran, and Debra Hensgen. Task execution time modeling for heterogeneous computing systems. In *Heterogeneous Computing Workshop, 2000.(HCW 2000) Proceedings. 9th*, pages 185–199. IEEE, 2000.

[BB99] Mark Bakery and Rajkumar Buyyaz. Cluster computing at a glance. *High Performance Cluster Computing: Architectures and Systems*, 1:3–47, 1999.

[BBD+07] Duncan A Brown, Patrick R Brady, Alexander Dietz, Junwei Cao, Ben Johnson, and John McNabb. A case study on the use of workflow technologies for scientific analysis: Gravitational wave data analysis. In *Workflows for e-Science*, pages 39–59. Springer, 2007.

[BCD+08] Shishir Bharathi, Ann Chervenak, Ewa Deelman, Gaurang Mehta, Mei-Hui Su, and Karan Vahi. Characterization of scientific workflows. In *Workflows in Support of Large-Scale Science, 2008. WORKS 2008. Third Workshop on*, pages 1–10. IEEE, 2008.

[BGL+04] GB Berriman, JC Good, AC Laity, A Bergou, J Jacob, DS Katz, E Deelman, C Kesselman, G Singh, M-H Su, et al. Montage: a grid enabled image mosaic service for the national virtual observatory. In *Astronomical Data Analysis Software and Systems (ADASS) XIII*, volume 314, page 593, 2004.

[BKK+11] Eun-Kyu Byun, Yang-Suk Kee, Jin-Soo Kim, Ewa Deelman, and Seungryoul Maeng. Bts: Resource capacity estimate for time-targeted science workflows. *Journal of Parallel and Distributed Computing*, 71(6):848–862, 2011.

[BM02] Ladislau Bölöni and Dan C Marinescu. Robust scheduling of metaprograms. *Journal of Scheduling*, 5(5):395–412, 2002.

[BM07] Luiz F Bittencourt and Edmundo RM Madeira. Fulfilling task dependence gaps for workflow scheduling on grids. In *Signal-Image Technologies and Internet-Based System, 2007. SITIS'07. Third International IEEE Conference on*, pages 468–475. IEEE, 2007.

[BM08] Jorge Barbosa and António P Monteiro. A list scheduling algorithm for scheduling multi-user jobs on clusters. In *High Performance Computing for Computational Science-VECPAR 2008*, pages 123–136. Springer, 2008.

[BM10] L.F. Bittencourt and E. Madeira. Towards the scheduling of multiple workflows on computational grids. *Journal of Grid Computing*, 8:419–441, 2010.

[BM11a] Jorge G Barbosa and Belmiro Moreira. Dynamic scheduling of a batch of parallel task jobs on heterogeneous clusters. *Parallel Computing*, 37(8):428–438, 2011.

[BM11b] Luiz Fernando Bittencourt and Edmundo Roberto Mauro Madeira. Hcoc: a cost optimization algorithm for workflow scheduling in hybrid clouds. *Journal of Internet Services and Applications*, 2(3):207–227, 2011.

[Boc14] Klavdiya Bochenina. A comparative study of scheduling algorithms for the multiple deadline-constrained workflows in heterogeneous computing systems with time windows. *Procedia Computer Science*, 29:509–522, 2014.

[BR+04] Cristina Boeres, Vinod EF Rebello, et al. A cluster-based strategy for scheduling task on heterogeneous processors. In *Computer Architecture and High Performance Computing, 2004. SBAC-PAD 2004. 16th Symposium on*, pages 214–221. IEEE, 2004.

[BSB+01] Tracy D Braun, Howard Jay Siegel, Noah Beck, Ladislau L Bölöni, Muthucumaru Maheswaran, Albert I Reuther, James P Robertson, Mitchell D Theys, Bin Yao, Debra Hensgen, et al. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed computing*, 61(6):810–837, 2001.

[BSM+01] Peter Blaha, Karlheinz Schwarz, GKH Madsen, Dieter Kvasnicka, and Joachim• Luitz. Wien2k: An augmented plane wave plus local orbitals program for calculating crystal properties, 2001.

[BSM10] Luiz F Bittencourt, Rizos Sakellariou, and Edmundo RM Madeira. Dag scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 27–34. IEEE, 2010.

[BVB08] James Broberg, Srikumar Venugopal, and Rajkumar Buyya. Market-oriented grids and utility computing: The state-of-the-art and future directions. *Journal of Grid Computing*, 6(3):255–276, 2008.

[CB76] Edward Grady Coffman and John L Bruno. *Computer and job-shop scheduling theory*. John Wiley & Sons, 1976.

[CB14] Rodrigo N Calheiros and Rajkumar Buyya. Meeting deadlines of scientific workflows in public clouds with tasks replication. *Parallel and Distributed Systems, IEEE Transactions on*, 25(7):1787–1796, 2014.

[CCD+05] 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*, pages 99–106, November 2005.

[CDS10] Henri Casanova, Frédéric Desprez, and Frédéric Suter. On cluster resource allocation for multiple parallel task graphs. *Journal of Parallel and Distributed Computing*, 70(12):1193–1203, 2010.

[Čer85] Vladimír Černỳ. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of optimization theory and applications*, 45(1):41–51, 1985.

[CFFK01] Karl Czajkowski, Steven Fitzgerald, Ian Foster, and Carl Kesselman. Grid information services for distributed resource sharing. In *High Performance Distributed Computing, 2001. Proceedings. 10th IEEE International Symposium on*, pages 181–194. IEEE, 2001.

[CFK+98] Karl Czajkowski, Ian Foster, Nick Karonis, Carl Kesselman, Stuart Martin, Warren Smith, and Steven Tuecke. A resource management architecture for metacomputing systems. In *Job Scheduling Strategies for Parallel Processing*, pages 62–82. Springer, 1998.

[CGL+14] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917, 2014.

[CJ01] Bertrand Cirou and Emmanuel Jeannot. Triplet: a clustering scheduling algorithm for heterogeneous systems. In *Parallel Processing Workshops, 2001. International Conference on*, pages 231–236. IEEE, 2001.

[CJSZ08] Louis-Claude Canon, Emmanuel Jeannot, Rizos Sakellariou, and Wei Zheng. Comparative evaluation of the robustness of dag scheduling heuristics. In *Grid Computing*, pages 73–84. Springer, 2008.

[CJW+08] Haijun Cao, Hai Jin, Xiaoxin Wu, Song Wu, and Xuanhua Shi. Dagmap: Efficient scheduling for dag grid workflow job. In *Proceedings of the 2008 9th IEEE/ACM International Conference on Grid Computing*, pages 17–24. IEEE Computer Society, 2008.

[CJW+10] Haijun Cao, Hai Jin, Xiaoxin Wu, Song Wu, and Xuanhua Shi. Dagmap: efficient and dependable scheduling of dag workflow job in grid. *The Journal of supercomputing*, 51(2):201–223, 2010.

[CLCG13] Zhicheng Cai, Xiaoping Li, Long Chen, and Jatinder ND Gupta. Bi-direction adjust heuristic for workflow scheduling in clouds. In *Parallel and Distributed Systems (ICPADS), 2013 International Conference on*, pages 94–101. IEEE, 2013.

[CLQ08] Henri Casanova, Arnaud Legrand, and Martin Quinson. Simgrid: A generic framework for large-scale distributed experiments. In *Computer Modeling and Simulation, 2008. UKSIM 2008. Tenth International Conference on*, pages 126–131. IEEE, 2008.

[CR92] Yeh-Ching Chung and Sanjay Ranka. Applications and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed memory multiprocessors. In *Supercomputing'92., Proceedings*, pages 512–521. IEEE, 1992.

[CSM+04] Jorge Cardoso, Amit Sheth, John Miller, Jonathan Arnold, and Krys Kochut. Quality of service for workflows and web service processes. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(3):281–308, 2004.

[CZ09] Wei-Neng Chen and Jun Zhang. An ant colony optimization approach to a grid workflow scheduling problem with various qos requirements. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 39(1):29–43, 2009.

[CZ12] Wei-Neng Chen and Jun Zhang. A set-based discrete pso for cloud workflow scheduling with user-defined qos constraints. In *Systems, Man, and Cybernetics (SMC), 2012 IEEE International Conference on*, pages 773–778. IEEE, 2012.

[DAYA02] Muhammad K Dhodhi, Imtiaz Ahmad, Anwar Yatama, and Ishfaq Ahmad. An integrated technique for task matching and scheduling onto distributed heterogeneous computing systems. *Journal of parallel and distributed computing*, 62(9):1338–1361, 2002.

[DBG+03] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, Kent Blackburn, Albert Lazzarini, Adam Arbree, Richard Cavanaugh, et al. Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing*, 1(1):25–39, 2003.

[DCG99] Marco Dorigo, Gianni Di Caro, and Luca Maria Gambardella. Ant algorithms for discrete optimization. *Artificial life*, 5(2):137–172, 1999.

[DFP12] Juan J Durillo, Hamid Mohammadi Fard, and Radu Prodan. Moheft: A multi-objective list-based method for workflow scheduling. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 185–192. IEEE, 2012.

[DG97] Marco Dorigo and Luca Maria Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *Evolutionary Computation, IEEE Transactions on*, 1(1):53–66, 1997.

[DK08] Mohammad I Daoud and Nawwaf Kharma. A high performance algorithm for static task scheduling in heterogeneous distributed computing systems. *Journal of Parallel and distributed computing*, 68(4):399–409, 2008.

[DNP14] Juan J Durillo, Vlad Nae, and Radu Prodan. Multi-objective energy-efficient workflow scheduling using list-based heuristics. *Future Generation Computer Systems*, 36:221–236, 2014.

[DÖ05] A. Doğan and F. Özgüner. Bi-objective scheduling algorithms for execution time–reliability trade-off in heterogeneous computing systems. *The Computer Journal*, 48(3):300–314, 2005.

[DPAM02] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Evolutionary Computation, IEEE Transactions on*, 6(2):182–197, 2002.

[DSS+05] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G Bruce Berriman, John Good, et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.

[ERL90] Hesham El-Rewini and Ted G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of parallel and Distributed Computing*, 9(2):138–153, 1990.

[FFP13] Hamid Mohammadi Fard, Thomas Fahringer, and Radu Prodan. Budget-constrained resource provisioning for scientific applications in clouds. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, volume 1, pages 315–322. IEEE, 2013.

[GJ79] Michael R Garey and David S Johnson. Computers and intractability: a guide to the theory of np-completeness. 1979. *San Francisco, LA: Freeman*, 1979.

[Glo89] Fred Glover. Tabu search-part i. *ORSA Journal on computing*, 1(3):190–206, 1989.

[Glo90] Fred Glover. Tabu search—part ii. *ORSA Journal on computing*, 2(1):4–32, 1990.

[Goo] Google. @ONLINE. http://code.google.com/appengine/.

[GS11] Ritu Garg and Awadhesh Kumar Singh. Multi-objective workflow grid scheduling based on discrete particle swarm optimization. In *Swarm, Evolutionary, and Memetic Computing*, pages 183–190. Springer, 2011.

[GS13] Ritu Garg and Awadhesh Kumar Singh. Enhancing the discrete particle swarm optimization based workflow grid scheduling using hierarchical structure. *International Journal of Computer Network and Information Security (IJCNIS)*, 5(6):18, 2013.

[HCTY+12] Adán Hirales-Carbajal, Andrei Tchernykh, Ramin Yahyapour, José Luis González-García, Thomas Röblitz, and Juan Manuel Ramírez-Alcaraz. Multiple workflow scheduling strategies with user run time estimates on a grid. *Journal of Grid Computing*, 10(2):325–346, 2012.

[HHW11] C.C. Hsu, K.C. Huang, and F.J. Wang. Online scheduling of workflow applications in grid environments. *Future Generation Computer Systems*, 27(6):860–870, 2011.

[HJ03] Tarek Hagras and Jan Janecek. A simple scheduling heuristic for heterogeneous computing environments. In *null*, page 104. IEEE, 2003.

[HJ05] Tarek Hagras and Jan Janeček. A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems. *Parallel Computing*, 31(7):653–670, 2005.

[HLP15] T. Hirofuchi, A. Lebre, and L. Pouilloux. Simgrid vm: Virtual machine support for a simulation framework of distributed systems. *Cloud Computing, IEEE Transactions on*, PP(99):1–1, 2015.

[HS06] U. Hönig and W. Schiffmann. A meta-algorithm for scheduling multiple dags in homogeneous system environments. In *Parallel and Distributed Computing and Systems*. ACTA Press, 2006.

[IÖF95] Michael A Iverson, Füsun Özgüner, and Gregory J Follen. Parallelizing existing applications in a distributed heterogeneous environment. In *4TH HETEROGENEOUS COMPUTING WORKSHOP (HCW'95*. Citeseer, 1995.

[IT07] E Ilavarasan and P Thambidurai. Low complexity performance effective task scheduling algorithm for heterogeneous computing environments. *Journal of Computer sciences*, 3(2):94–103, 2007.

[ITM05] E Ilavarasan, P Thambidurai, and R Mahilmannan. High performance task scheduling algorithm for heterogeneous computing system. In *Distributed and Parallel Computing*, pages 193–203. Springer, 2005.

[JCD+13] Gideon Juve, Ann Chervenak, Ewa Deelman, Shishir Bharathi, Gaurang Mehta, and Karan Vahi. Characterizing and profiling scientific workflows. *Future Generation Computer Systems*, 29(3):682–692, 2013.

[JDB+12] Gideon Juve, Ewa Deelman, G Bruce Berriman, Benjamin P Berman, and Philip Maechling. An evaluation of the cost and performance of scientific workflows on amazon ec2. *Journal of Grid Computing*, 10(1):5–21, 2012.

[JHC+11] He-Jhan Jiang, Kuo-Chan Huang, Hsi-Ya Chang, Di-Syuan Gu, and Po-Jen Shih. Scheduling concurrent workflows in hpc cloud through exploiting schedule gaps. In *Algorithms and Architectures for Parallel Processing*, pages 282–293. Springer, 2011.

[KA96] Yu-Kwong Kwok and Lshfaq Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 7(5):506–521, 1996.

[KA99] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, 1999.

[KGV+83] Scott Kirkpatrick, C Daniel Gelatt, Mario P Vecchi, et al. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[Kha12] Minhaj Ahmad Khan. Scheduling for heterogeneous systems using constrained critical paths. *Parallel Computing*, 38(4):175–193, 2012.

[KR12] B Arun Kumar and T Ravichandran. Time and cost optimization algorithm for scheduling multiple workflows in hybrid clouds. *European Journal of Scientific Research*, 89(2):265–275, 2012.

[KS74] Walter H Kohler and Kenneth Steiglitz. Characterization and theoretical comparison of branch-and-bound algorithms for permutation problems. *Journal of the ACM (JACM)*, 21(1):140–156, 1974.

[KSW97] D. Karger, C. Stein, and J. Wein. Scheduling algorithms. *CRC Handbook of Computer Science*, 1997.

[KY94] Dongseung Kim and Byung-Guoen Yi. A two-pass scheduling algorithm for parallel programs. *Parallel Computing*, 20(6):869–885, 1994.

[LCJY09] Ke Liu, Jinjun Chen, Hai Jin, and Yun Yang. A min-min average algorithm for scheduling transaction-intensive grid workflows. In *Proceedings of the Seventh Australasian Symposium on Grid Computing and e-Research-Volume 99*, pages 41–48. Australian Computer Society, Inc., 2009.

[LP+97] Jing-Chiou Liou, Michael Palis, et al. A comparison of general approaches to multiprocessor scheduling. In *Parallel Processing Symposium, 1997. Proceedings., 11th International*, pages 152–156. IEEE, 1997.

[LPX05] GQ Liu, Kim-Leng Poh, and Min Xie. Iterative list scheduling for heterogeneous computing. *Journal of Parallel and Distributed Computing*, 65(5):654–665, 2005.

[LSZ09] Young Choon Lee, Riky Subrata, and Albert Y Zomaya. On the performance of a dual-objective optimization model for workflow applications on grid platforms. *Parallel and Distributed Systems, IEEE Transactions on*, 20(9):1273–1284, 2009.

[MAS+99] Muthucumaru Maheswaran, Shoukat Ali, Howard Jay Siegel, Debra Hensgen, and Richard F Freund. Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 59(2):107–131, 1999.

[MH11] Ming Mao and Marty Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 49. ACM, 2011.

[MJDN15] Maciej Malawski, Gideon Juve, Ewa Deelman, and Jarek Nabrzyski. Algorithms for cost-and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds. *Future Generation Computer Systems*, 48:1–18, 2015.

[NS09] T. N'takpé and F. Suter. Concurrent scheduling of parallel task graphs on multiclusters using constrained resource allocations. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[OH96] Hyunok Oh and Soonhoi Ha. A static scheduling heuristic for heterogeneous processors. In *Euro-Par'96 Parallel Processing*, pages 573–577. Springer, 1996.

[Peg13] Pegasus. Pegasus workflow generator. https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator, 2013.

[PGB+14] Deepak Poola, Saurabh Kumar Garg, Rajkumar Buyya, Yun Yang, and Kotagiri Ramamohanarao. Robust scheduling of scientific workflows with deadline and budget constraints in clouds. In *Advanced Information Networking and Applications (AINA), 2014 IEEE 28th International Conference on*, pages 858–865. IEEE, 2014.

[PK01] Hee-Jun Park and Byung Kook Kim. An optimal scheduling algorithm for minimizing the computing period of cyclic synchronous tasks on multiprocessors. *Journal of Systems and Software*, 56(3):213–229, 2001.

[PSL03] Chintan Patel, Kaustubh Supekar, and Yugyung Lee. A qos oriented framework for adaptive management of web service based workflows. In *Database and Expert Systems Applications*, pages 826–835. Springer, 2003.

[PW10] Radu Prodan and Marek Wieczorek. Bi-criteria scheduling of scientific grid workflows. *Automation Science and Engineering, IEEE Transactions on*, 7(2):364–376, 2010.

[PY79] Christos H. Papadimitriou and Mihalis Yannakakis. Scheduling interval-ordered tasks. *SIAM Journal on Computing*, 8(3):405–409, 1979.

[RVG00] Andrei Rădulescu and Arjan JC Van Gemund. Fast and effective task scheduling in heterogeneous systems. In *Heterogeneous Computing Workshop, 2000.(HCW 2000) Proceedings. 9th*, pages 229–238. IEEE, 2000.

[SJD06] Zhiao Shi, Emmanuel Jeannot, and Jack J Dongarra. Robust task scheduling in nondeterministic heterogeneous computing systems. In *Cluster Computing, 2006 IEEE International Conference on*, pages 1–10. IEEE, 2006.

[SK12] Claudia Szabo and Trent Kroeger. Evolving multi-objective strategies for task allocation of scientific workflows on public clouds. In *Evolutionary Computation (CEC), 2012 IEEE Congress on*, pages 1–8. IEEE, 2012.

[SKD07] Gurmeet Singh, Carl Kesselman, and Ewa Deelman. A provisioning model and its comparison with best-effort for performance-cost optimization in grids. In *Proceedings of the 16th international symposium on High performance distributed computing*, pages 117–126. ACM, 2007.

[SL+93] Gilbert C Sih, Edward Lee, et al. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *Parallel and Distributed Systems, IEEE Transactions on*, 4(2):175–187, 1993.

[SLH+13] Sen Su, Jian Li, Qingjia Huang, Xiao Huang, Kai Shuang, and Jie Wang. Cost-efficient task scheduling for executing large programs in the cloud. *Parallel Computing*, 39(4):177–188, 2013.

[SS04] Oliver Sinnen and Leonel Sousa. List scheduling: extension for contention awareness and evaluation of node priorities for heterogeneous cluster architectures. *Parallel Computing*, 30(1):81–101, 2004.

[STZN13] Shaghayegh Sharif, Javid Taheri, Albert Y Zomaya, and Surya Nepal. Mphc: Preserving privacy for workflow execution in hybrid clouds. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2013 International Conference on*, pages 272–280. IEEE, 2013.

[SZ04] Rizos Sakellariou and Henan Zhao. A hybrid heuristic for dag scheduling on heterogeneous systems. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 111. IEEE, 2004.

[SZTD07] Rizos Sakellariou, Henan Zhao, Eleni Tsiakkouri, and Marios D Dikaiakos. Scheduling workflows with budget constraints. In *Integrated research in GRID computing*, pages 189–202. Springer, 2007.

[THW99] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Task scheduling algorithms for heterogeneous processors. In *Heterogeneous Computing Workshop, 1999.(HCW'99) Proceedings. Eighth*, pages 3–14. IEEE, 1999.

[THW02] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260–274, 2002.

[TKB09] AKM Talukder, Michael Kirley, and Rajkumar Buyya. Multiobjective differential evolution for scheduling workflow applications on global grids. *Concurrency and Computation: Practice and Experience*, 21(13):1742–1756, 2009.

[TULA13] Tanyaporn Tirapat, Orachun Udomkasemsub, Xiaorong Li, and Tiranee Achalakul. Cost optimization for scientific workflow execution on cloud computing. In *Parallel and Distributed Systems (ICPADS), 2013 International Conference on*, pages 663–668. IEEE, 2013.

[Ull75] Jeffrey D. Ullman. Np-complete scheduling problems. *Journal of Computer and System sciences*, 10(3):384–393, 1975.

[VL09] Pedro Velho and Arnaud Legrand. Accuracy study and improvement of network simulation in the simgrid framework. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, page 13. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009.

[WG90] Min-You Wu and Daniel D Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Transactions on Parallel & Distributed Systems*, 1(3):330–343, 1990.

[WLN+13] Zhangjun Wu, Xiao Liu, Zhiwei Ni, Dong Yuan, and Yun Yang. A market-oriented hierarchical scheduling strategy in cloud workflow systems. *The Journal of Supercomputing*, 63(1):256–293, 2013.

[WLY+15] Chunlin Wu, XINGQIN LIN, Daren Yu, Wei Xu, and Luoqing Li. End-to-end delay minimization for scientific workflows in clouds under budget constraint. *Cloud Computing, IEEE Transactions on*, 3(2):169–181, 2015.

[WPPF08] M. Wieczorek, S. Podlipnig, R. Prodan, and T. Fahringer. Bi-criteria scheduling of scientific workflows for the grid. In *Cluster Computing and the Grid, 2008. CCGRID'08. 8th IEEE International Symposium on*, pages 9–16. IEEE, 2008.

[WWT15] Fuhui Wu, Qingbo Wu, and Yusong Tan. Workflow scheduling in cloud: a survey. *The Journal of Supercomputing*, pages 1–46, 2015.

[XCWB09] Meng Xu, Lizhen Cui, Haiyang Wang, and Yanbing Bi. A multiple qos constrained scheduling strategy of multiple workflows for cloud computing. In *Parallel and Distributed Processing with Applications, 2009 IEEE International Symposium on*, pages 629–634. IEEE, 2009.

[YB06a] J. Yu and R. Buyya. A budget constrained scheduling of workflow applications on utility grids using genetic algorithms. In *Workflows in Support of Large-Scale Science, 2006. WORKS'06. Workshop on*, pages 1–10. IEEE, 2006.

[YB06b] Jia Yu and Rajkumar Buyya. Scheduling scientific workflow applications with deadline and budget constraints using genetic algorithms. *Scientific Programming*, 14(3-4):217–230, 2006.

[YBR08] J. Yu, R. Buyya, and K. Ramamohanarao. Workflow scheduling algorithms for grid computing. *Metaheuristics for scheduling in distributed computing environments*, pages 173–214, 2008.

[YBT05a] Jia Yu, Rajkumar Buyya, and Chen Khong Tham. Cost-based scheduling of scientific workflow applications on utility grids. In *e-Science and Grid Computing, 2005. First International Conference on*, pages 8–pp. IEEE, 2005.

[YBT+05b] Jia Yu, Rajkumar Buyya, Chen Khong Tham, et al. Qos-based scheduling of workflow applications on service grids. In *Proc. of 1st IEEE International Conference on e-Science and Grid Computing*, 2005.

[YKB07] Jia Yu, Michael Kirley, and Rajkumar Buyya. Multi-objective planning for workflow execution on grids. In *Proceedings of the 8th IEEE/ACM International conference on Grid Computing*, pages 10–17. IEEE Computer Society, 2007.

[YLM10] Yonglei Yao, Jingfa Liu, and Li Ma. Efficient cost optimization for workflow scheduling on grids. In *Management and Service Science (MASS), 2010 International Conference on*, pages 1–4. IEEE, 2010.

[YLW07] Yingchun Yuan, XiaoPing Li, and Qian Wang. Dynamic heuristics for time and cost reduction in grid workflows. In *Computer Supported Cooperative Work in Design III*, pages 499–508. Springer, 2007.

[YLWZ09] Yingchun Yuan, Xiaoping Li, Qian Wang, and Xia Zhu. Deadline division-based heuristic for cost optimization in workflow scheduling. *Information Sciences*, 179(15):2562–2575, 2009.

[YRB09] Jia Yu, Kotagiri Ramamohanarao, and Rajkumar Buyya. Deadline/budget-based scheduling of workflows on utility grids. *Market-Oriented Grid and Utility Computing*, pages 427–450, 2009.

[YS07] Zhifeng Yu and Weisong Shi. An adaptive rescheduling strategy for grid workflow applications. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[YS08] Zhifeng Yu and Weisong Shi. A planner-guided scheduling strategy for multiple workflow applications. In *Parallel Processing-Workshops, 2008. ICPP-W'08. International Conference on*, pages 1–8. IEEE, 2008.

[YVB06] Jia Yu, Srikumar Venugopal, and Rajkumar Buyya. A market-oriented grid directory service for publication and discovery of grid service providers and their services. *The Journal of Supercomputing*, 36(1):17–31, 2006.

[ZH14] Amelie Chi Zhou and Bingsheng He. Transformation-based monetary costoptimizations for workflows in the cloud. *Cloud Computing, IEEE Transactions on*, 2(1):85–98, 2014.

[ZS06] Henan Zhao and Rizos Sakellariou. Scheduling multiple dags onto heterogeneous systems. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 14–pp. IEEE, 2006.

[ZS12] Wei Zheng and Rizos Sakellariou. Budget-deadline constrained workflow planning for admission control in market-oriented environments. In *Economics of Grids, Clouds, Systems, and Services*, pages 105–119. Springer, 2012.

[ZS13] Wei Zheng and Rizos Sakellariou. Budget-deadline constrained workflow planning for admission control. *Journal of grid computing*, 11(4):633–651, 2013.

[ZVL12] Lingfang Zeng, Bharadwaj Veeravalli, and Xiaorong Li. Scalestar: Budget conscious scheduling precedence-constrained many-task workflow applications in cloud. In *Advanced Information Networking and Applications (AINA), 2012 IEEE 26th International Conference on*, pages 534–541. IEEE, 2012.

[ZVL15] Lingfang Zeng, Bharadwaj Veeravalli, and Xiaorong Li. Saba: A security-aware and budget-aware workflow scheduling strategy in clouds. *Journal of Parallel and Distributed Computing*, 75:141–151, 2015.