Scheduling Complex Workloads in Large-Scale Distributed Systems: Trends and Challenges

Georgios L. Stavrinides and Helen D. Karatza

Abstract With the advent of the Internet of Things, the ubiquity of mobile computing and the ever-increasing momentum of social networks, big data analytics and cloud computing, numerous aspects of our daily life rely on inevitably complex workloads, processed on distributed interconnected resources that are becoming larger in scale and computational capacity. Complex applications may have different degrees of parallelism and may impose several Quality of Service requirements, such as time constraints and resilience against failures, as well as other objectives, like energy efficiency. These features of the workloads, as well as the inherent characteristics of the computing resources required to process them, present major challenges that require the employment of effective scheduling techniques. In this chapter, a classification of complex workloads is proposed and an overview of the most commonly used approaches for their scheduling in large-scale distributed systems is given. We present novel strategies that have been proposed in the literature and shed light on open challenges and future directions.

Key words: Gang scheduling; Workflow scheduling; Bag-of-Tasks scheduling; Real-time applications; Fault tolerance; Energy efficiency.

1 Introduction

With the rapid pace of technological advances in mobile computing, big data analytics and cloud computing, as well as with the ever-increasing popularity of social

Georgios L. Stavrinides
Department of Informatics, Aristotle University of Thessaloniki, 54124 Thessaloniki, Greece, e-mail: gstavrin@csd.auth.gr

Helen D. Karatza
Department of Informatics, Aristotle University of Thessaloniki, 54124 Thessaloniki, Greece, e-mail: karatza@csd.auth.gr

networks and the advent of recent concepts such as the Internet of Things, many applications we use in our daily life are inevitably becoming more and more complex. Such applications cover a wide spectrum of areas, like healthcare, weather forecasting, environmental monitoring, social interaction, scientific research, industrial manufacturing, telecommunications, multimedia streaming services, financial markets and e-commerce. The complex workloads generated by such applications, are usually processed on interconnected computing resources that are geographically distributed, encompass various heterogeneous components and are becoming larger in scale and computational capacity day by day. Computer clusters, computational grids and clouds are examples of such platforms.

Complex applications may have different degrees of parallelism and may impose several Quality of Service (QoS) requirements, such as time constraints and resilience against failures, as well as other objectives, like energy efficiency. These features of the workloads, as well as the characteristics of the computing resources required to process them, present major challenges that require the employment of effective scheduling techniques. Due to their inherent complexity, the performance of such algorithms is usually evaluated by simulation, rather than by analytical methods. Analytical modeling is difficult and often requires several simplifying assumptions that may have an unpredictable impact on the results.

This chapter is organized as follows: Sect. 2 gives a definition of the scheduling problem in large-scale distributed systems, as well as some of the most important scheduling objectives. In Sect. 3, a classification of complex workloads is proposed, according to their degree of parallelism. An overview of the most widely used strategies for the scheduling of each class of complex applications in large-scale distributed systems is given. Sect. 4 presents other challenges of complex workload scheduling, covering topics such as timeliness, fault tolerance and energy efficiency. Furthermore, novel strategies that have been proposed in the literature are presented in Sect. 5. Finally, Sect. 6 concludes this chapter, shedding light on open challenges and future research directions.

2 Scheduling Problem

In its general form, the scheduling problem in large-scale distributed systems concerns the mapping of a set of application tasks $V = \{n_1, n_2, ..., n_N\}$ to a set of processors $P = \{p_1, p_2, ..., p_Q\}$, in order to complete all tasks under the specified constraints (e.g. complete each task within its deadline) [2, 12]. In this general form, the scheduling problem has been shown to be NP-complete [7].

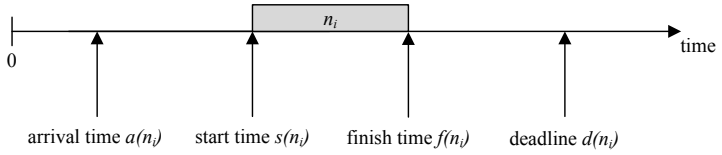

Fig. 1 Typical parameters that characterize a task of an application submitted for execution in a large-scale distributed system.

2.1 Scheduling Objectives

Some of the parameters that characterize a task $n_i \in V$ are shown in Fig. 1. These parameters are:

- *arrival time* $a(n_i)$: it is the time at which the task arrives at the system.
- *start time* $s(n_i)$: it is the time at which the task starts its execution.
- *finish time* $f(n_i)$: it is the time at which the task finishes its execution.
- *deadline* $d(n_i)$: it is the time before which the task should finish its execution.

Based on the above parameters, some of the most commonly used scheduling objectives in large-scale distributed systems are:

(a) To minimize the *average response time* $\overline{R}$ of the tasks $n_i \in V$, where $\overline{R}$ is given by:

$$\overline{R} = \frac{1}{N} \sum_{n_i \in V} R(n_i) \tag{1}$$

where $R(n_i) = f(n_i) - a(n_i)$ and N is the number of tasks in V.

(b) To minimize the *makespan* (i.e. total execution time) M of the tasks $n_i \in V$, where M is defined as:

$$M = \max_{n_i \in V} \{f(n_i)\} - \min_{n_i \in V} \{s(n_i)\} \tag{2}$$

(c) To maximize the *task guarantee ratio* TGR of the tasks $n_i \in V$, where TGR is given by:

$$TGR = \frac{1}{N} \sum_{n_i \in V} guar(n_i) \tag{3}$$

where

$$guar(n_i) = \begin{cases} 1 & \text{if } f(n_i) \leq d(n_i) \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

(d) To minimize the *average tardiness* $\overline{T}$ of the tasks $n_i \in V$, where $\overline{T}$ is defined as:

$$\overline{T} = \frac{1}{N} \sum_{n_i \in V} T(n_i) \tag{5}$$

where

$$T(n_i) = \begin{cases} f(n_i) - d(n_i) & \text{if } f(n_i) > d(n_i) \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

3 Complex Workloads in Distributed Systems

The applications scheduled for execution in large-scale distributed systems, typically consist of numerous component tasks. At the one end of the spectrum, the tasks require frequent communication with each other during their execution. At the other end of the spectrum, the component tasks do not require any communication and are completely independent. Between these two ends, is the case where communication is required between the component tasks of an application, but only before or after their execution. Consequently, complex workloads in large-scale distributed systems can be classified into the following categories:

- *fine-grained parallel applications*,
- *coarse-grained parallel applications* and
- *embarrassingly parallel applications*.

In the following paragraphs, each class of complex applications is presented in more detail and their corresponding, most widely used scheduling heuristics are analyzed.

3.1 Fine-Grained Parallel Applications

An application features *fine-grained parallelism* when it consists of frequently communicating parallel tasks. A proven and effective way to schedule such applications is *gang scheduling*. According to this approach, the parallel tasks of an application form a *gang* and are scheduled and executed simultaneously on different processors. Hence, all of the tasks of the application start execution at the same time. This way, the risk of a task waiting to communicate with another task that is currently not running is avoided. The task with the largest execution time determines the execution time of the gang. An example of a gang with N parallel tasks is shown in Fig. 2.

Consequently, gang scheduling facilitates the synchronization between the component tasks of a fine-grained parallel application. Without this technique, the synchronization of the component tasks would require more context switches and thus additional overhead. On the other hand, in order to utilize gang scheduling, the number of available processors must be greater than or equal to the number of parallel tasks of an application. Furthermore, due to the requirement that all of the tasks of a gang must start execution at the same time, there may be times at which some of the

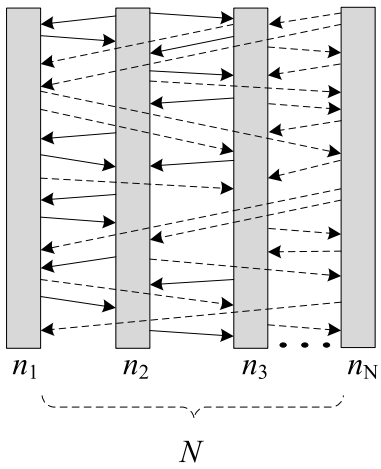

Fig. 2 An example of a fine-grained parallel application. The frequently communicating tasks of the application form a gang of N parallel tasks. The communication between the tasks is depicted with arrows.

processors are idle, even with tasks waiting in their respective queues. Specifically, a task at the head of the queue of an idle processor may be waiting for the other tasks of its gang, which may not be able to start execution at the particular time instant [32]. This situation is depicted in Fig. 3.

3.1.1 Gang Scheduling Policies

The two most widely used gang scheduling policies are the *Adapted First Come First Served (AFCFS)* and *Largest Gang First Served (LGFS)* strategies.

Adapted First Come First Served (AFCFS)

This method is an adapted version of the First Come First Served (FCFS) scheduling heuristic, according to which the gang that arrived first, has the highest priority for execution. A gang starts execution when its tasks are at the head of their assigned queues and the respective processors are idle. When there are not enough idle pro-

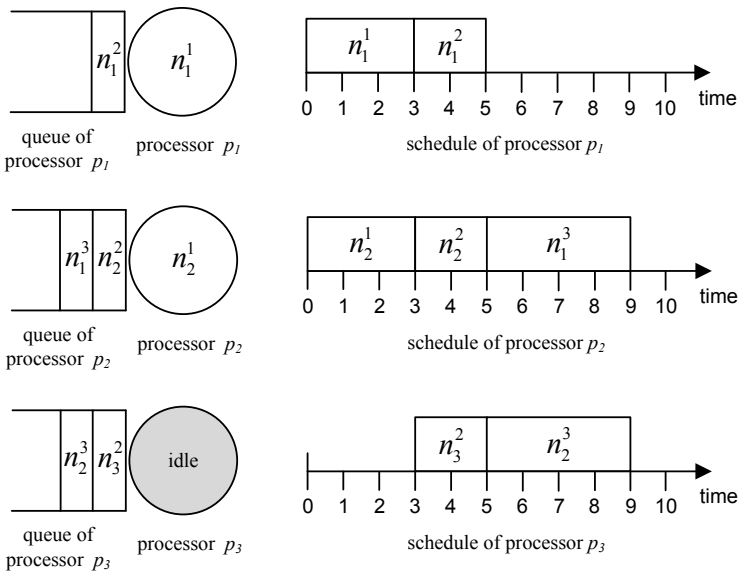

Fig. 3 Example of gang scheduling in a system with three processors p_1, p_2 and p_3. The first gang consists of the tasks n_1^1 and n_2^1, scheduled on processors p_1 and p_2, respectively. The second gang consists of the tasks n_1^2, n_2^2 and n_3^2, scheduled on processors p_1, p_2 and p_3, respectively. The third gang consists of the tasks n_1^3 and n_2^3, scheduled on processors p_2 and p_3, respectively. It can be observed that the processor p_3 remains idle during the execution of the tasks n_1^1 and n_2^1 of the first gang. This is due to the fact that the task n_3^2 at the head of its queue cannot start execution, because according to the gang scheduling technique, it must start execution at the same time as the other tasks of its gang, n_1^2 and n_2^2, which are scheduled on the other processors that are currently busy.

cessors for a gang with a large number of parallel tasks waiting at the front of their assigned queues, a smaller gang with tasks waiting behind those of the larger gang can start execution. This technique is also referred to as *backfilling* [10].

The major drawback of this scheduling policy is that it tends to favor smaller gangs, which leads to greater response times for larger gangs. In order to overcome this issue, various techniques have been proposed in the literature, such as the employment of a *bypass count* parameter [17] and the utilization of *task migrations* [22]. The first method, counts for each gang the number of gangs that bypassed it, due to an insufficient number of idle processors. When the bypass count of a gang reaches a specified threshold, it gets the highest priority for execution. According to the second method, the tasks of a gang are candidate for migration only if at least one of them is at the head of its assigned queue and the respective processor is idle. The tasks that are migrated, are placed at the head of their newly assigned queues. In order to avoid the starvation of the other tasks, there is a limit on the number of migrated tasks a queue can accept.

Largest Gang First Served (LGFS)

According to this scheduling strategy, the tasks in the processor queues are sorted in descending order of gang size (i.e. number of tasks) of their respective gang. Thus, tasks that belong to larger gangs have higher priority than tasks that belong to smaller gangs. Whenever a processor becomes idle, the scheduler searches the queues starting from the head of each queue and the first gang with tasks that can start execution occupies the processors [11]. Clearly, this strategy tends to favor applications with a high degree of parallelism (i.e. large gangs), at the expense of smaller gangs. However, this is sometimes desirable and may lead to a better system performance, compared to the AFCFS policy.

3.2 Coarse-Grained Parallel Applications

In case an application exhibits *coarse-grained parallelism*, its component tasks do not require any communication with each other during processing, but only before or after their execution. That is, the component tasks have precedence constraints among them, in such a way that the output data of a task are used as input by other tasks. A component task can only start execution when its predecessor tasks have completed. A task without any parent tasks is called an *entry task*, whereas a task without any child tasks is called an *exit task*.

Such an application is often called a *workflow application* and can be represented by a *Directed Acyclic Graph (DAG)* or *task graph*, $G = (V, E)$, where V and E are the sets of the nodes and the edges of the graph, respectively [27, 29, 30]. Each node represents a component task, whereas a directed edge between two tasks represents the data that must be transmitted from the first task to the other. Each node has a weight that represents the computational cost of its corresponding task. Each edge between two tasks has a weight that denotes the communication cost that is incurred when transferring data from the first task to the other.

The *level* of a task in the graph is equal to the length of the longest path from the particular task to an exit task in the graph. The length of a path is the sum of the computational and communication costs of all of the nodes and edges, respectively, along the path. The *critical path* of the graph is the longest path from an entry task to an exit task in the graph. An example of a workflow application is illustrated in Fig. 4.

3.2.1 Workflow Scheduling Approaches

Workflow applications require a scheduling strategy that should take into account the precedence constraints among their component tasks. The workflow scheduling heuristics are classified into the following general categories:

- *list scheduling algorithms,*

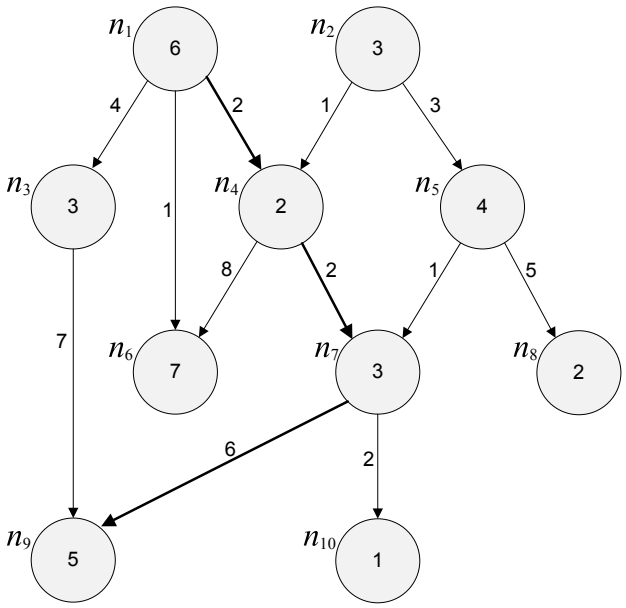

Fig. 4 An example of a coarse-grained parallel application (workflow application), represented as a Directed Acyclic Graph (DAG). The number in each node denotes the computational cost of the represented task. The number on each edge denotes the communication cost between the two tasks that it connects. The critical path of the DAG is depicted with thick arrows.

- *clustering algorithms,*
- *task duplication algorithms* and
- *guided random search algorithms*.

These techniques are analyzed in the following paragraphs.

List Scheduling Algorithms

A list scheduling algorithm consists of two phases: (a) a *task selection phase* and (b) a *processor selection phase*. In the first phase, the tasks are prioritized based on specific criteria and are arranged in a list according to their priority. The task with the highest priority is selected first for scheduling. During the second phase, the selected task is scheduled to the processor that minimizes a specific cost function, such as the estimated start time of the task [34]. List scheduling algorithms are the most commonly used among the workflow scheduling heuristics, because they are generally simpler, more practical, easier to implement and they usually outperform other techniques, incurring less scheduling overhead [37].

One of the simplest list scheduling policies is the *Highest Level First (HLF)* [1]. According to this method, the task prioritization phase is based on the level of each task. In the processor selection phase, the selected task is scheduled to the processor that can provide it with the earliest start time. An improved version of the HLF strategy is the *Insertion Scheduling Heuristic (ISH)* [13] and it is based on the observation that idle time slots may form in the schedule of a processor (schedule gaps), due to the data dependencies among the tasks. The task selection phase of this technique is based on HLF. However, during the processor selection phase, a task may be inserted into a schedule gap, as long as it does not delay the execution of the succeeding task in the schedule and provided that it cannot start earlier on any other processor. An alternative version of ISH, adapted for heterogeneous systems, is the *Heterogeneous Earliest Finish Time (HEFT)* policy [37]. According to this approach, for the calculation of the level of each task, the average computational and communication costs of the tasks and edges, respectively, are used.

Clustering Algorithms

The main idea of clustering algorithms is the minimization of the communication cost between the tasks of a DAG, by grouping heavily communicating tasks into the same cluster and assigning all of the tasks in the cluster to the same processor. A clustering algorithm is an iterative process. At first, each task is an independent cluster. At each iteration, previous clusters are refined by merging some of them, according to specific criteria. At the end of the process, a cluster merging step is needed, so that the number of clusters is equal to the number of processors. Subsequently, a cluster mapping step is required, in order to map each cluster to a processor. Finally, a task ordering step is performed, in order to determine the execution order of tasks on each processor [9].

One of the most popular clustering techniques is the *Dominant Sequence Clustering (DSC)* algorithm [40]. This method is based on the observation that the makespan of a DAG is determined by the longest path in the scheduled task graph and not by its critical path, which is calculated before the scheduling of the tasks of the DAG. The longest path in the scheduled DAG is called the *dominant sequence (DS)*. According to the DSC algorithm, the tasks in a DAG are clustered in such a way, so that the dominant sequence of the graph is minimized.

Task Duplication Algorithms

In this category of workflow scheduling heuristics, the main concept is to utilize idle resource time by duplicating predecessor tasks in a DAG, so that the makespan of the particular DAG is minimized. The various duplication-based algorithms differentiate with each other, according to the criteria used for the selection of the tasks for duplication. One of the major drawbacks of task duplication algorithms, is that they usually have higher complexity than the other DAG scheduling techniques.

One of the most well-known duplication algorithms is the *Duplication Scheduling Heuristic (DSH)* [13]. According to this approach, the tasks in a DAG are prioritized according to their level. At each scheduling step, the task with the highest level is selected and is allocated to the processor that can provide it with the earliest start time. In order to calculate the earliest possible start time of the selected task on each processor, first its start time is calculated without duplication of any predecessor tasks. Subsequently, the *duplication time slot* is determined, which is the time period between the finish time of the last scheduled task on the particular processor and the start time of the currently examined task. The algorithm then tries to duplicate the predecessors of the task into the duplication time slot in a recursive manner, starting from the parent task from which the data arrives the latest, until either the slot cannot accommodate other predecessor tasks or the start time of the examined task is not improved.

Guided Random Search Algorithms

A guided random search algorithm is an iterative process of finding the best schedule for a DAG, based on specific criteria. At each step, the previously generated schedule is improved, by utilizing random parameters for the generation of the new schedule. This iterative process terminates according to a predefined condition. These algorithms, even though they generally generate schedules of good quality, however, they incur a much higher scheduling overhead than the other workflow scheduling methods. The most commonly used algorithms of this category are *genetic algorithms*, according to which each new schedule is generated by applying evolutionary techniques from nature, known as *fitness functions* [8].

Simulated Annealing (SA) is another example of a guided random search metaheuristic. This technique emulates the physical process of annealing in metallurgy, which involves the heating and the controlled, slow cooling of metals, in order to form a crystallized structure without any defects [20]. In SA, a temperature variable is used in order to simulate this heating process. Initially, it is set at a high value and as the algorithm runs, it is allowed to slowly cool down. While the value of the temperature variable is high, the algorithm is allowed to accept solutions that are worse than the current one, with higher frequency. As the value of the temperature variable is decreased, so is the chance of accepting worse solutions. Therefore, the algorithm gradually focuses on an area of the search space in which hopefully a near-optimal solution can be found.

3.3 Embarrassingly Parallel Applications

An application is regarded as *embarrassingly parallel* when its component tasks are independent, do not communicate with each other and can be executed in any order. Due to these characteristics, such applications are also called *Bag-of-Tasks*

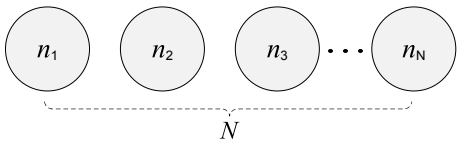

Fig. 5 An embarrassingly parallel application, consisting of N independent parallel tasks. Such applications are commonly referred to as Bag-of-Tasks (BoT) applications.

(BoT) applications. Due to the independence between their tasks, BoT applications are well suited for execution on widely distributed resources, such as computational grids, where communication can become a bottleneck for more tightly-coupled parallel applications, such as gangs and DAGs [39]. An example of a BoT application is depicted in Fig. 5.

3.3.1 Scheduling BoT Applications

The most widely used strategies for scheduling BoT applications are: (a) *Min-Min*, (b) *Max-Min* and (c) *Sufferage*. All of these policies focus on minimizing the makespan of the scheduled BoT application.

Min-Min

This heuristic is an iterative process, consisting of two steps. In the first step, the *minimum completion time (MCT)* of each unassigned task is calculated, over all of the processors in the system. In the second step, the task with the minimum MCT is assigned to the corresponding processor. At each iteration of the algorithm, the MCT of each unassigned task is determined taking into account the current load of the processors, as resulted by the scheduling decision of the previous iteration [39].

Max-Min

This strategy differs from the Min-Min policy, in that the task with the maximum (instead of the minimum) MCT is assigned to the corresponding processor in the second step of the scheduling process. Consequently, in cases where the application consists of a large number of tasks with small execution times and a few tasks with large execution times, the Max-Min heuristic is likely to give a smaller makespan than the Min-Min algorithm, since it schedules the tasks with larger execution times at earlier iterations [35].

Sufferage

This algorithm is a two-step iterative process, like the Min-Min and Max-Min heuristics. However, in this case, in addition to the MCT of each task, its second MCT is also calculated during the first step of the process. Subsequently, the *sufferage value* of each task is determined, by subtracting its MCT from its second MCT. In the second step, the task with the largest sufferage value is assigned to the processor that can provide it with the MCT. That is, this heuristic is based on the idea that the highest priority for scheduling should be given to the task that would suffer the most (in terms of completion time) if it is not assigned to the processor that can provide it with the MCT [16].

4 Other Challenges

In addition to the challenges imposed by their degree of parallelism, complex applications in large-scale distributed systems may also have various QoS requirements, such as timeliness and fault tolerance, as well as other objectives, like energy efficiency. These requirements are usually specified in a *Service Level Agreement (SLA)*, which is a contract between the user that submits the application for execution and the provider of the infrastructure that the application is executed on. In the following paragraphs, representative examples for each case are given.

4.1 Scheduling Complex Applications with Time Constraints

The most common QoS requirement that complex applications may impose, is to finish execution within a strict time constraint. Such applications are regarded as *real-time*, since they have a deadline that must be met [2]. Two of the most widely used policies for the scheduling of real-time complex applications are the *Earliest Deadline First (EDF)* and the *Least Laxity First (LLF)* algorithms [15, 19]. According to the EDF strategy, the component task with the highest priority for execution is the one with the earliest deadline. On the other hand, according to the LLF policy, the task with the highest priority is the one with the minimum *laxity*. The laxity of a task at a specific time instant, is defined as the difference between its deadline and its finish time. That is, it is the maximum amount of time that the particular task can delay its execution and still not miss its deadline.

A heuristic for the scheduling of real-time workflow applications in distributed systems, is the *Least Space-Time First (LSTF)* policy [5], which takes into account both the precedence and the time constraints among the tasks. Specifically, according to this method, the task with the highest priority for scheduling is the one with the minimum value of the *space-time* parameter. The space-time parameter of a task at a specific time instant, is defined as the difference between the deadline of the

DAG and the level of the particular task. Even though this algorithm outperforms other scheduling policies, such as EDF, LLF and HLF described earlier, in the sense that it minimizes the maximum tardiness of the tasks, however, it exhibits poorer performance at guaranteeing deadlines.

4.1.1 Approximate Computations

Based on the observation that it is often more desirable for a real-time application to produce an approximate result by its deadline, than to produce a precise result late, the technique of *approximate computations* has been proposed [14]. According to this method, a real-time application is allowed to return intermediate, approximate results of poorer, but still acceptable quality, when the deadline of the application cannot be met. Approximate computations can be utilized especially in the case of applications with *monotone* component tasks, where the quality of a task's results is improved as more time is spent to produce them (e.g. statistical estimation and video processing tasks). Each monotone task typically consists of a *mandatory part*, followed by an *optional part*. In order for a task to return an acceptable result, its mandatory part must be completed. The optional part refines the result produced by the mandatory part. Consequently, the approximate computations technique provides scheduling flexibility, by trading off precision for timeliness, since it allows the scheduler to terminate a task that has completed its mandatory part at any time, depending on the workload conditions of the system [25, 26].

4.2 Fault-Tolerant Scheduling of Complex Workloads

Fault tolerant scheduling in large-scale distributed systems, such as clouds, is usually achieved through *application-directed checkpointing*, which in contrast to system-directed checkpointing, is more practical, easier to implement and system-independent [21]. According to this approach, each application is responsible for checkpointing its own progress periodically, at regular intervals during its execution. In complex applications in particular, each component task periodically stores its state and intermediate data on persistent storage, creating a local checkpoint. The set of the local checkpoints (one from each task) that form a consistent application state, constitute a consistent global checkpoint.

When a failure occurs, the application is rolled back and resumes execution from its last consistent global checkpoint. Checkpointing is a reactive failure management technique, where recovery measures are taken after the occurrence of a failure. As opposed to proactive failure management approaches, where prevention measures are taken before the occurrence of a failure (e.g. task migrations), reactive management is simpler to implement, since it does not require any complex failure prediction methods.

4.3 Energy Efficient Scheduling of Complex Applications

There is a growing focus on *green computing* from both the academia and the industry, in an attempt to minimize the carbon footprint of data centers and increase the energy efficiency of applications. Typically, in most computing systems the processor consumes the greatest amount of energy compared to other components [38]. In embedded systems, as well as in large-scale virtualized platforms such as the cloud, a technique that is frequently used in order to meet the energy constraints is the *Dynamic Voltage and Frequency Scaling (DVFS)* method. This technique allows the dynamic adjustment of the supply voltage and operating frequency (i.e. speed) of a processor, based on the workload conditions, in an attempt to reduce the energy consumption of the processor [12, 36].

A heuristic frequently used with DVFS, is the *slack reclamation* technique [4]. This method is based on the fact that the actual execution time of tasks is sometimes much shorter than their estimated worst case execution time. The difference between the actual and the worst case execution time of a task is called *slack time*. At runtime, the scheduler tries to reclaim the slack time due to the early completion of a task, by selecting an unprocessed task to be executed at a slower processor speed via DVFS and thus save energy.

An energy-efficient scheduling strategy for real-time BoT applications in the cloud utilizing DVFS, is the *Cloud-Aware Energy-Efficient Scheduling (CAEES)* algorithm [3]. At each scheduling step, this method attempts to reduce the total energy consumption of the hosts, by selecting the most suitable virtual machine (VM) for the execution of each task, in an energy-wise manner. Specifically, the algorithm tries to schedule a task by examining specific criteria, starting from the best solution and gradually going to the worst solution: (a) the task is scheduled to a VM in use, without requiring an increase in its frequency, (b) the task is scheduled to a VM in use, but its operating frequency needs to be increased, (c) the task is scheduled to an idle VM, but there is at least one other VM on the same host that is not idle (i.e. the host is not idle) and (d) the task is scheduled to an idle VM on an idle host.

5 Recent Novel Ideas and Research Trends

In an attempt to provide even more effective scheduling solutions for complex workloads in large-scale distributed systems, recent novel approaches have been proposed in the literature. A prominent research trend is the utilization of approximate computations in combination with other techniques, in order to achieve better scheduling performance, in terms of timeliness, resilience against failures and energy conservation. For example, approximate computations can be combined with:

- bin packing techniques, in order to enhance timeliness,
- checkpointing, in an attempt to improve fault tolerance and
- DVFS, for better energy efficiency.

5.1 Approximate Computations with Bin Packing

The traditional *bin packing* problem concerns the packing of a set of objects into a set of bins, using as few bins as possible [6]. The most commonly used bin packing techniques are: (a) *First Fit (FF)*, where the object is placed into the first bin where it fits, (b) *Best Fit (BF)*, where the object is placed into the bin where it fits and leaves the minimum unused space possible and (c) *Worst Fit (WF)*, where the object is placed into the bin where it fits and leaves the maximum unused space possible.

In an attempt to improve the timeliness of real-time workflow applications in a heterogeneous distributed system, a novel list scheduling heuristic has been proposed, which utilizes schedule gaps with a technique that combines approximate computations with the FF, BF and WF bin packing policies [28, 31]. Another characteristic of the proposed approach, is that it takes into account the effects of error propagation among the tasks of an application in case of partially completed tasks. The task prioritization is based on EDF. Once a task is selected by the scheduler, it is allocated to the processor that can provide it with the earliest estimated start time. In order to calculate the estimated start time of the task on the particular processor, schedule gaps are exploited with a technique that allows only a fraction of the task to be inserted into an idle time slot. The fraction of the task to be inserted into a schedule gap must be at least equal to the mandatory part of the task. Moreover, its potential output error must not exceed the input error limit of its child tasks.

The placement of the partial task into a schedule gap is performed using a modified version of either the FF, BF or WF bin packing policy:

- *First Fit with Approximate Computations (FF_AC)*: the task is placed into the first schedule gap where at least its minimum possible computational cost fits.
- *Best Fit with Approximate Computations (BF_AC)*: the task is placed into the schedule gap where its maximum possible computational cost fits, leaving the minimum unused time possible.
- *Worst Fit with Approximate Computations (WF_AC)*: the task is placed into the schedule gap where its minimum possible computational cost fits, leaving the maximum unused time possible.

In contrast to this approach, the other list scheduling heuristics presented earlier, ISH and HEFT, essentially use FF in order to utilize idle time slots. More importantly, with the incorporation of approximate computations, this approach is more flexible, allowing only a fraction of a task to be inserted into a schedule gap when the task does not completely fit into it. An example of scheduling tasks with the proposed heuristics (EDF_FF_AC, EDF_BF_AC and EDF_WF_AC), compared to the baseline EDF policy, is illustrated in Fig. 6. The parameters of the tasks used in the example are shown in Table 1.

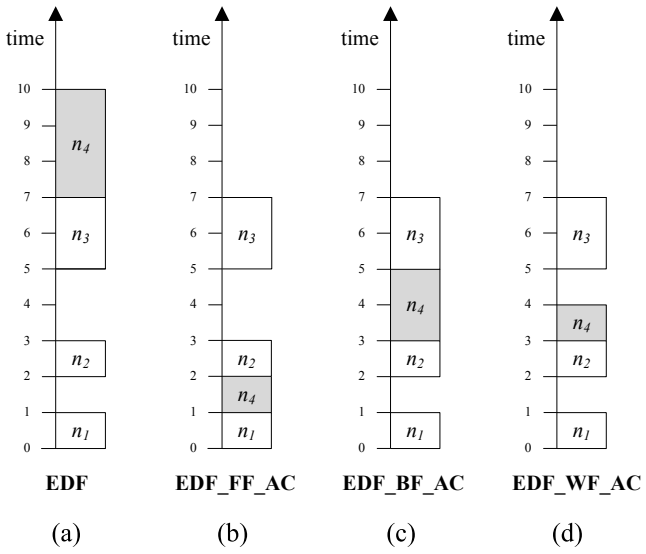

Fig. 6 An example of scheduling tasks with the strategies described in Subsection 5.1. A task n_4 is scheduled according to one of the policies: (a) EDF (baseline algorithm), (b) EDF_FF_AC, (c) EDF_BF_AC and (d) EDF_WF_AC. The parameters of the tasks used in the example are shown in Table 1.

Table 1 The parameters of the tasks used in the example of Fig. 6. For each task, d is its deadline, t_{data} is the time at which its required input data will be available, c is its computational cost and c_{min} is its minimum computational cost when approximate computations are utilized.

Task	d	t_{data}	c	c_{min}
n_1	2	0	1	1
n_2	4	2	1	1
n_3	9	5	2	1
n_4	10	1	3	1

5.2 Approximate Computations with Checkpointing

In an attempt to improve resilience against transient software failures in a SaaS cloud, where real-time fine-grained parallel applications are scheduled and executed, the approximate computations technique has been combined with application-directed checkpointing [23, 24, 33]. Specifically, gang scheduling is employed, where the prioritization of the component tasks is according to the EDF policy. In addition to application-directed checkpointing, fault tolerance is enhanced by the use of approximate computations in either a restricted manner or a more holistic

approach. In the first case, an application may provide approximate results when it has completed its parallel mandatory part and (a) its deadline is reached, (b) a failure occurred and its last generated checkpoint stored results corresponding to computational work greater than or equal to its mandatory part or (c) another notified application must start execution immediately (i.e. there is time to execute only the mandatory part of the other application before its deadline). According to the second approach, all applications are scheduled to complete only their mandatory part. That is, in this case all applications give approximate results.

5.3 Approximate Computations with DVFS

In order to enhance energy efficiency, a heuristic that combines approximate computations with DVFS has been proposed, for the scheduling of periodic real-time tasks [18]. According to this approach, the tasks are scheduled according to the Mandatory-First Earliest Deadline (MFED) policy, while the supply voltage and processor frequency are scaled according to the Cycle-Conserving Real-Time DVFS (CC-RT-DVFS) technique. MFED is a policy according to which the mandatory parts of the tasks have always higher priority than the optional parts. The mandatory part with the earliest deadline has the highest priority for execution. CC-RT-DVFS is essentially a dynamic slack reclamation technique, which utilizes the slack time that occurs due to the early completion of a mandatory part, for the scheduling of the optional part of the task at a lower processor speed, utilizing DVFS. Thus, in this strategy there is a trade-off not only between result precision and timeliness, but also between result precision and energy savings.

6 Conclusions

In this chapter, a classification of complex workloads was proposed and an overview of the most commonly used heuristics for their scheduling in large-scale distributed systems was given. Other challenges of complex applications were covered, such as timeliness, resilience against failures and energy efficiency. Furthermore, recent novel ideas and research trends were presented.

Scheduling complex workloads in large-scale distributed systems remains an active research area, with many open challenges. As the workloads tend to get more complex and computationally demanding, more effective scheduling heuristics must be employed. In addition to the timeliness, fault tolerance and energy efficiency objectives, security and data awareness are drawing an ever-increasing interest from both the industry and the research community. Hence, efforts towards this direction are expected to be intensified in the near future.

Acknowledgements The second author of this chapter, Helen D. Karatza, has been invited as a trainer to the cHiPSet Training School 2016 *"New Trends in Modeling and Simulation in HPC Systems"*, held in Bucharest, Romania, 21-23 September 2016, and has been supported by the IC1406 Horizon 2020 grant.

References

[1] Adam TL, Chandy KM, Dickson JR (1974) A comparison of list schedules for parallel processing systems. Commun ACM 17(12):685–690
[2] Buttazzo GC (2011) Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, 3rd edn. Springer
[3] Calheiros RN, Buyya R (2014) Energy-efficient scheduling of urgent bag-of-tasks applications in clouds through dvfs. In: Proc. 6th IEEE Int. Conf. Cloud Comput. Technol. Sci. (CloudCom'14), pp 342–349
[4] Chen JJ, Yang CY, Kuo TW (2006) Slack reclamation for real-time task scheduling over dynamic voltage scaling multiprocessors. In: Proc. 2006 IEEE Int. Conf. Sens. Netw. Ubiquitous Trust. Comput. (SUTC'06), pp 358–365
[5] Cheng BC, Stoyenko AD, Marlowe TJ, Baruah SK (1997) Lstf: A new scheduling policy for complex real-time tasks in multiple processor systems. Automatica 33(5):921–926
[6] Coffman Jr EG, Csirik J, Galambos G, Martello S, Vigo D (2013) Bin packing approximation algorithms: survey and classification, Springer, pp 455–531
[7] Garey MR, Johnson DS (1979) Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company
[8] Gkoutioudi KZ, Karatza HD (2012) Multi-criteria job scheduling in grid using an accelerated genetic algorithm. J Grid Comput 10(2):311–323
[9] Jiang HJ, Huang KC, Chang HY, Gu DS, Shih PJ (2011) Scheduling concurrent workflows in hpc cloud through exploiting schedule gaps. In: Proc. 11th Int. Conf. Algorithms Archit. Parallel Process. (ICA3PP'11), pp 282–293
[10] Karatza HD (2008) The impact of critical sporadic jobs on gang scheduling performance in distributed systems. Simul: Trans Soc Model Simul Int 84(2–3):89–102
[11] Karatza HD (2014) Scheduling jobs with different characteristics in distributed systems. In: Proc. 2014 Int. Conf. Comput. Inf. Telecommun. Syst. (CITS'14), pp 1–5
[12] Kolodziej J (2012) Evolutionary Hierarchical Multi-Criteria Metaheuristics for Scheduling in Large-Scale Grid Systems. Springer
[13] Kruatrachue B, Lewis TG (1987) Duplication scheduling heuristic, a new precedence task scheduler for parallel systems. Tech. Rep. 87-60-3, Oregon State University, Corvallis, OR
[14] Lin KJ, Natarajan S, Liu JWS (1987) Imprecise results: utilizing partial computations in real-time systems. In: Proc. 8th IEEE Real-Time Syst. Symp. (RTSS'87), pp 210–217

[15] Liu CL, Layland JW (1973) Scheduling algorithms for multiprogramming in a hard real-time environment. J ACM 20(1):46–61
[16] Maheswaran M, Ali S, Siegel HJ, Hensgen D, Freund RF (1999) Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. J Parallel Distrib Comput 59(2):107–131
[17] Manickam V, Aravind A (2012) A fair and efficient gang scheduling algorithm for multicore processors. In: Proc. 6th Int. Conf. Inf. Process. (ICIP'12), pp 467–476
[18] Mizotani K, Hatori Y, Kumura Y, Takasu M, Chishiro H, Yamasaki N (2015) An integration of imprecise computation model and real-time voltage and frequency scaling. In: Proc. 30th Int. Conf. Comput. Their Appl. (CATA'15), pp 63–70
[19] Mok AK (1983) Fundamental design problems of distributed systems for the hard real-time environment. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA
[20] Moschakis IA, Karatza HD (2015) Multi-criteria scheduling of bag-of-tasks applications on heterogeneous interlinked clouds with simulated annealing. J Syst Softw 101:1–14
[21] Oldfield RA, Arunagiri S, Teller PJ, Seelam S, Varela MR, Riesen R, Roth PC (2007) Modeling the impact of checkpoints on next-generation systems. In: Proc. 24th IEEE Conf. Mass Storage Syst. Technol. (MSST'07), pp 30–46
[22] Papazachos ZC, Karatza HD (2009) Performance evaluation of gang scheduling in a two-cluster system with migrations. In: Proc. 23rd IEEE Int. Parallel Distrib. Process. Symp. (IPDPS'09), pp 1–8
[23] Stavrinides GL, Karatza HD (2008) Performance evaluation of gang scheduling in distributed real-time systems with possible software faults. In: Proc. 2008 Int. Symp. Perform. Eval. Comp. Telecommun. Syst. (SPECTS'08), pp 1–7
[24] Stavrinides GL, Karatza HD (2009) Fault-tolerant gang scheduling in distributed real-time systems utilizing imprecise computations. Simul: Trans Soc Model Simul Int 85(8):525–536
[25] Stavrinides GL, Karatza HD (2010) Scheduling multiple task graphs with end-to-end deadlines in distributed real-time systems utilizing imprecise computations. J Syst Softw 83(6):1004–1014
[26] Stavrinides GL, Karatza HD (2011) The impact of input error on the scheduling of task graphs with imprecise computations in heterogeneous distributed real-time systems. In: Proc. 18th Int. Conf. Anal. Stoch. Model. Tech. Appl. (ASMTA'11), pp 273–287
[27] Stavrinides GL, Karatza HD (2011) Scheduling multiple task graphs in heterogeneous distributed real-time systems by exploiting schedule holes with bin packing techniques. Simul Model Pract Theory 19(1):540–552
[28] Stavrinides GL, Karatza HD (2012) Scheduling real-time dags in heterogeneous clusters by combining imprecise computations and bin packing techniques for the exploitation of schedule holes. Futur Gener Comput Syst 28(7):977–988

[29] Stavrinides GL, Karatza HD (2014) The impact of resource heterogeneity on the timeliness of hard real-time complex jobs. In: Proc. 7th Int. Conf. PErvasive Technol. Relat. Assist. Environ. (PETRA'14), Workshop Distrib. Sens. Syst. Assist. Environ. (Di-Sensa), pp 65:1–65:8

[30] Stavrinides GL, Karatza HD (2014) Scheduling real-time jobs in distributed systems - simulation and performance analysis. In: Proc. 1st Int. Workshop Sustain. Ultrascale Comput. Syst. (NESUS'14), pp 13–18

[31] Stavrinides GL, Karatza HD (2015) A cost-effective and qos-aware approach to scheduling real-time workflow applications in paas and saas clouds. In: Proc. 3rd Int. Conf. Futur. Internet Things Cloud (FiCloud'15), pp 231–239

[32] Stavrinides GL, Karatza HD (2016) Scheduling different types of applications in a saas cloud. In: Proc. 6th Int. Symp. Bus. Model. Softw. Des. (BMSD'16), pp 144–151

[33] Stavrinides GL, Karatza HD (2016) Scheduling real-time parallel applications in saas clouds in the presence of transient software failures. In: Proc. 2016 Int. Symp. Perform. Eval. Comp. Telecommun. Syst. (SPECTS'16), pp 1–8

[34] Stavrinides GL, Duro FR, Karatza HD, Blas JG, Carretero J (2017) Different aspects of workflow scheduling in large-scale distributed systems. Simul Model Pract Theory 70:120–134

[35] Tabak EK, Cambazoglu BB, Aykanat C (2014) Improving the performance of independent task assignment heuristics minmin, maxmin and sufferage. IEEE Trans Parallel Distrib Syst 25(5):1244–1256

[36] Terzopoulos G, Karatza HD (2014) Bag-of-task scheduling on power-aware clusters using a dvfs-based mechanism. In: Proc. 28th IEEE Int. Parallel & Distrib. Process. Symp. (IPDPS'14), 10th Workshop High-Perform. Power-Aware Comput. (HPPAC'14), pp 833–840

[37] Topcuoglu H, Hariri S, Wu MY (2002) Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Trans Parallel Distrib Syst 13(3):260–274

[38] Valentini GL, Lassonde W, Khan SU, Allah NM, Madani SA, Li J, Zhang L, Wang L, Ghani N, Kolodziej J, Li H, Zomaya AY, Xu CZ, Balaji P, Vishnu A, Pinel F, Pecero JE, Kliazovich D, Bouvry P (2013) An overview of energy efficiency techniques in cluster computing systems. Clust Comput 16(1):3–15

[39] Weng C, Lu X (2005) Heuristic scheduling for bag-of-tasks applications in combination with qos in the computational grid. Futur Gener Comput Syst 21(2):271–280

[40] Yang T, Gerasoulis A (1994) Dsc: scheduling parallel tasks on an unbounded number of processors. IEEE Trans Parallel Distrib Syst 5(9):951–967