Time-constrained and network-aware containers scheduling in GPU era

Leonardo Rosa Rodrigues[a], Guilherme Piêgas Koslovski[a,*], Marcelo Pasin[b], Maurício Aronne Pillon[a], Omir Correa Alves Junior[a], Charles Christian Miers[a]

[a] *Graduate Program in Applied Computing – Santa Catarina State University – Brazil*
[b] *University of Neuchâtel and HES-SO – Switzerland*

Abstract

The recent advances on data center management and applications development are reflected by lightweight containers technology and critical Quality-of-Service (QoS) requirements. Tenants encapsulate applications in containers abstracting away details on the infrastructure, and entrust its management framework with the provisioning of network and time QoS requirements. In this paper, we addressed this NP-hard scheduling problem proposing a GPU Accelerated Containers Scheduler (GPUACS). We model the joint allocation of network and containers with QoS requirements as a graph embedding problem. GPUACS innovates by refactoring two Multicriteria Decision Makings (MCDMs) to GPU model, as well as by defining an efficient data structure to speed up the comparison of time-evolving QoS requirements. GPUACS follows a modular and configurable architecture, and the scheduling objective function can be adjusted by selecting the MCDM method and setting the appropriated weights to guide the comparisons. An experimental analysis demonstrated the sensitivity that GPU-tailored MCDM methods have to schedule container requests considering critical time, network, and processing criteria, as well as multiple queuing policies.

*Corresponding author
Email addresses: leonardo.rodrigues@edu.udesc.br (Leonardo Rosa Rodrigues), guilherme.koslovski@udesc.br (Guilherme Piêgas Koslovski), marcelo.pasin@unine.ch (Marcelo Pasin), mauricio.pillon@udesc.br (Maurício Aronne Pillon), omir.alves@udesc.br (Omir Correa Alves Junior), charles.miers@udesc.br (Charles Christian Miers)

Keywords: scheduling, multitenant, multicriteria, GPU, network
2020 MSC: 00-01, 99-00

1. Introduction

The problem of efficiently allocating a set of servers to host a set of jobs is a fundamental challenge in distributed systems. Such a scheduling problem is well-known in the specialized literature, but the combination of a large number of concurrent jobs (numerous clients, different applications), and the large size of current data centers (DCs) (countless servers) brings a whole new dimension to the problem. Over the last years, several schedulers have been developed for DC resources, allocating together servers and networking equipment. They host jobs deployed on real machines, Virtual Machines (VMs) or even containers, according to a variety of scenarios as, for instance, high performance computing and clouds [1, 2, 3].

The large-size of current DCs and the heterogeneity of jobs make it hard for schedulers to simultaneously achieve QoS requirements, high DC utilization, and scheduling delays. Moreover, communication plays a central role in distributed applications. Point-to-point or collective communication are used to distribute tasks and data, synchronize the execution, and monitor the application. In this sense, network requirements must be jointly analyzed with processing ones to improve the performance indicators of DCs and applications. However, any additional network QoS requirement increases the number of comparisons needed to select candidates for hosting communicating jobs. Given the facts, the main objective of the present work is to advance the field on scheduling network-aware containers requests by considering 3-axes together of open challenges on large-scale containerized DCs:

1. **Joint allocation of networks and containers.** Although containers offer a disruptive technology, the complexity of network management in containerized DCs is not softened by the container technology. The joint scheduling of network requirements (*e.g.*, latency, high resource utilization,

and bandwidth sharing) using malleable and lightweight containers is an open challenge [4]. In addition, the network building blocks for containers are based on third-party drivers for virtual network provisioning, which naturally induces to communication overheads [5]. Specifically regarding the scheduling delays (time needed to decide and propagate an allocation), the number of comparisons necessary to select servers for hosting communicating containers is increased when compared to standalone containers scheduling. In fact, in addition to selecting servers, the scheduler must indicate a DC path with QoS guarantees to forward data between the containers.

2. **Time-constrained requests.** Time-critical jobs have specific requirements for scheduling. The allocation and execution of a job after a strict deadline is undesirable as it consumes resources (processing and networking) to perform tasks which are no longer useful. In addition, the heterogeneity of containerized workloads as well as the large-scale topology of distributed DC applications impose a computational barrier for schedulers. On the application-oriented DCs management's perspective, the scheduler runtime influences on DC resource usage and acceptance ratio of requests [6], while for application's perspective can increase the waiting time before executions [2].

3. **DC dimensionality and scheduler scalability.** The joint scheduling of containers and network requirements can be reduced to a series of similar NP-hard problems, such as the multi-way separator problem and the virtual network embedding [7, 8]. The diversity of job requests combined to the high-dimension of contemporaneous DCs are indeed critical factors for a scheduler. In terms of execution time and multi-objective combinatorial analysis, finding an efficient mapping of requests onto DC resources results on a dichotomy (runtime *vs.* quality). Schedulers carrying out a great number of comparisons (candidates DC resources and jobs) tend to explore the search space in wide, consequently finding accurate results. However, they require high execution times. Current DC can achieve

hundreds of thousands of servers, interconnected by multipath network topologies in the same scale, making the wide search impractical due to time and processing constrains.

1.1. Contributions

The adoption of Graphics Processing Units (GPUs) as a high-performance accelerator to schedule virtual infrastructures has been pointed out as a promising approach [9, 3]. Traditional algorithms have been refactored to efficiently explore the Single Instruction Multiple Data (SIMD) parallelism model of GPUs. A SIMD architecture enables multiple parallel comparisons between DC resources candidates and requests components, however, requires the refactoring of existing algorithms to properly exploit a vectorial architecture. Following this line, the present work proposes GPU Accelerated Containers Scheduler (GPUACS). Figure 1 depicts a generic scenario to schedule a container-based application with heterogeneous requirements. An application can be spread in dozens, hundreds, or even thousands blocks, and each block requires a QoS configuration.

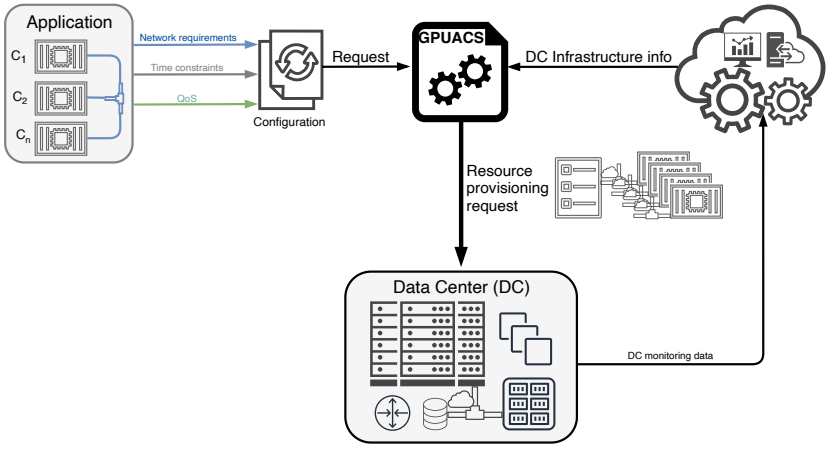

Figure 1: General scenario of GPUACS usage.

Regarding to the three challenges and research opportunities aforementioned,

GPUACS presents two main contributions[1]:

- A scalable scheduler: we model the joint allocation of network and containers with QoS requirements (challenge 1) as a graph embedding problem, in which vertices denote the DC servers (or containers) and edges represent the physical links and paths (or communication requirements). In addition, delay-sensitive requests (challenge 2) inform a fixed constraint regarding the end of execution time. The objective is to find a set of DC resources to host the request which meets into the QoS and time constraints. GPUACS innovated by refactoring two MCDM methods (Technique for Order Preference by Similarity to Ideal Solution (TOPSIS) and Analytic Hierarchy Process (AHP)) to the GPU SIMD programming model, as well as by proposing an Augmented Forest (a set of Augmented Trees) as a data structure to speed up the comparison of time-evolving QoS requirements (challenge 3). MCDM literature contains numerous proposals for ranking and selecting alternatives. In this work, we selected TOPSIS and AHP motivated by the code portability to GPU as well as for the simplicity and objectivity when modelling the problem, offered by both hierarchical methods.

- MCDM configuration and modular architecture: GPUACS follows a modular and configurable architecture to accommodate multiple DCs policies and request priority queues (challenges 2 and 3). Moreover, the scheduler objective function can be adjusted by selecting the MCDM method (or even including a new one) and setting the appropriated weights to guide the comparisons (challenge 1). Finally, GPUACS offers 7 options for ordering the request queues.

GPUACS was assessed in our experimental scalability analysis to schedule multiple large-scale requests on a DC with more than 20,000 servers. On av-

[1]Preliminary results were published at [3]. Specifically, the publication focused on network-aware scheduling of containers.

erage, our decision was computed in less than 3.5 seconds, even considering network and time requirements. A comparison of our results with traditional scheduling approaches (consolidation and spreading) shows GPUACS provides a decrease of the execution delay of requests while not increasing the DC network and server usage. Finally, the configurable aspect of GPUACS MCDM modules soften the impact on schedulers performance (DC resource usage and acceptance ratio) related with queuing policy selection.

1.2. Organization

This paper is organized as follows. Related work is discussed in Section 2, while the problem formulation and the modular architecture of GPUACS are presented in Sections 3 and 4, respectively. An evaluation of GPUACS is presented in Section 5. Finally, Section 6 lists our final remarks and future work.

2. Related work

State-of-the-art on scheduling containers, microservices, and network requirements seems to be still evolving nowadays, and this section reviews some recent proposals on the subject. It also includes a review of other similar proposals for scheduling Virtual Infrastructures (VIs), composed of VMs and virtual networking services. Table 1 summarizes the proposals reviewed, indicating the multicriteria method and the objective function of each one.

Guo *et al.* proposed the containers scheduling based on neighbourhood division algorithms to provide load balance as well as to reduce the distances between containers [10]. The experimental analysis comprised a homogeneous DC with serves interconnected by a Fat-tree [11] topology ($k = 18$, 1458 servers), comparing the proposal with a traditional Particle Swarm Optimization (PSO) [12]. In turn, Havet *et al.* proposed a framework to monitor and schedule containers on DCs, termed GenPack [13]. To reduce the energy consumption, GenPack uses a generational garbage collection and active monitoring techniques. The experimental analysis compared GenPack versus the traditional Docker Swarm

using a sample of Google Borg [6] traces as request input. The DC was composed by 13 servers, and the analysis pointed out the efficiency of GenPack for energy reduction.

The load balance of microservices (hosted by containers) on multitenant DC was investigated by Guerrero *et al.* [14]. In order to guide the decision making, the scheduler considered the containers workload and the network bandwidth usage. The experimental analysis indicates superior results from Kubernetes policies when allocating requests on a heterogeneous DC composed of 400 servers. Hu *et al.* [15] modelled the containers scheduling as a minimum-cost flow problem. The experimental analysis considered a DC composed of 30 heterogeneous servers. The default Kubernetes and Swarm schedulers were used as baseline for comparison, and the proposal reduced the average slowdown up to 30%.

Regarding VIs, Souza *et al.* investigated the allocation of Sofware Defined Networking (SDN)-based DC resources to host multi-tenant requests [16, 17]. The problem was formulated as a Mixed Integer Linear Programming (MILP), comprising latency and bandwidth requirements together with VMs demands. However, due to the problem complexity nature, solving exact models became computationally unfeasible. Then, the authors relaxed the problem's integer constraints, resulting in a linear program, in which a heuristic was introduced. Although the results highlighted the reduction on average latency, as aimed, they indicate a clear scalability limit of linear programs and heuristics for scheduling DC resources to host VIs.

In an investigation regarding priority queues, AHP and PSO was proposed by Alla *et al.* [18]. Their scheduler has a three-phase pipelined execution: *(i)* the requests are dynamically ordered on a priority queue; *(ii)* requests are send to AHP for ranking and latter forwarded to PSO; finally, *(iii)* PSO takes the decision. The experimental analysis was conducted on a DC composed of 60 servers, and 30 requests were submitted, comparing the resulting makespan with a traditional First Come First Served (FCFS). Following a similar track, Paswar [19] proposed a VM two-phase scheduler based on PSO and TOPSIS. The algorithm's first phase applies TOPSIS to rank the candidates (DC servers),

while the second phase uses PSO over the previously ranked servers.

Nesi *et al.* proposed a GPU-accelerated scheduling framework demonstrating that is possible to scale up the problem through the use of parallel programming techniques [20, 9]. Their framework is composed of clustering and graphs algorithms, and supports simulations with contemporary DC network topologies (*e.g.*, Fat-Tree, Bcube, and Dcell.). A performance analysis shows speedups of more than $6,000$, when comparing with traditional CPU graph embedding algorithms.

Carstan *et al.* revisited the easy backfilling algorithm investigating the influence of different queueing policies [2]. It analyzed queuing policies as FCFS, lower-time estimate, smallest estimated area first, and less resources first, with an easy backfilling approach. In short, the policy based on promoting smallest estimated areas reduced the average slowdown while ensuring no starvation in requests.

Table 1 summarizes the related work presented above. Most proposals are implemented using CPUs, facing a scalability barrier due to the natural problem complexity. Although the problem of joint allocation of VMs and network requirements has been addressed in the literature, there seems to be a lack of solutions and research addressing the joint scheduling of containers and their network interconnections. The present work is positioned on these research opportunities, achieving results on DCs equipped with more than $24,000$ servers (as discussed in Section 5). Furthermore, GPUACS uses MCDM methods accelerated by GPU to make the selection of the most appropriated server to support a request analysing multiple requirements, including the network's perspective. A GPU-tailored implementation of MCDM methods emerges as a promising approach to allocate resources on a large-scale DC, hosting time-critical requests composed of a dynamic set of criteria, QoS constraints, and optimization trade-offs.

Ref.	Algorithms/Model	Resources	Network	# Servers
[18]	Priority queues with AHP and PSO	VM	no	60
[14]	Genetic algorithms	container / VM	yes	400
[10]	Neighborhood division	container	yes	1,458
[13]	Generational garbage collection and active monitoring	container	no	13
[19]	TOPSIS and PSO	VM	no	10
[16]	MILP	VM	yes	128
[20] [9]	GPU-accelerated algorithms	VM	no	16,000
[2]	Easy backfilling and priority queues	VM	no	144
[15]	Minimum-cost flow problem	container	yes	30

Table 1: Summary of related work.

3. Problem formulation

Our formulation of the problem starts using a traditional weighted undirected graph embedding. Following this formulation, we argue the DC administrator should be able to dynamically configure the scheduler, instead of defining a specific fixed objective function. Finally, we consider using different queueing policies.

3.1. Weighted undirected graph embedding problem

The allocation of DC resources to host containers and network requirements considering QoS and time constraints is modelled as a graph embedding prob-

lem. Table 2 summarizes the notation and symbols used along this work. The DC is represented by a weighted undirected graph $G^s = (N^s, E^s)$, where N^s denotes the servers and E^s is the set of links connecting all servers. The capacities vector of a server $u \in N^s$ is given by $c^s_u[r]$ for capacities $r \in R$, while bw^s_{uv} denotes the available bandwidth between servers u and v. Following this line, $Req(N^c, E^c)$ represents a request composed of a set of containers N^c and their interconnection details (E^c). The containers are organized in pods, represented by pod_i.[2] A set of container's QoS requirements is informed by the tenant, as well as the target bandwidth between a pair of containers. Moreover, the QoS requirements are specified as intervals of minimum and maximum capacities. In other words, $\forall r \in R; \left[c^{min}[r], c^{max}[r]\right]$ and $\forall e \in E^c; \left[bw^{min}[e], bw^{max}[e]\right]$ for containers and bandwidth, respectively. Finally, each request informs a set of time constraints for containers. For a given container $i \in N^c$, the estimated processing time ($proc_i$, termed walltime) and an execution deadline ($deadline_i$) are specified. The first information is mandatory, while the second is optional.

The scheduling of a DC resource to host a container is given by a map $\mathcal{M}^s(i)$, while $\mathcal{M}^e(ij)$ follows the same principle for container's interconnections. A map is valid only if all QoS requirements and time constraints are provisioned and guaranteed by the DC infrastructure. Alongside defining a DC server for hosting the container, the provisioned capacity for each $r \in R$ must be allocated, respecting the minimum and maximum intervals specified by the application's owner during all reservation period (walltime). In this sense, $c^a_{iu}[r]$ represents the capacities allocated on server u for hosting a container i, while bw^a_{ijuv} denotes the bandwidth reserved to support the communication between containers i and j atop the physical link $uv \in E^s$. If containers i and j have the same pod identificator associated with, they must be hosted by the same server; in other words, $\mathcal{M}^s(i) == \mathcal{M}^s(j)$. Regarding the time constraints, each container's request has a submission time (sub_i) associated with by the DC or-

[2]The organization in pods allow for modelling the same grouping method offered by Kubernetes, which is currently a very popular container manager system.

Notation	Description
$G^s(N^s, E^s)$	data center (DC) graph composed of N^s servers and E^s links
$c^s_u[r]$	Array of resources capacities for a server $u \in N^s$
bw^s_{uv}	Bandwidth capacity between servers u and v, $uv \in E^s$
$Req(N^c, E^c)$	Request composed of N^c containers and E^c links
$c^{min}_i[r]$, $c^{max}_i[r]$	Minimum and maximum QoS requirements for $i \in N^c$
bw^{min}_{ij}, bw^{max}_{ij}	Minimum and maximum bandwidth requirements for $ij \in E^c$
pod_i	Each container $i \in N^c$ belongs to a pod
$c^a_{iu}[r]$	QoS resources capacities from $u \in N^s$ allocated to $i \in N^c$
bw^a_{ijuv}	Bandwidth capacity from $uv \in E^s$ allocated to $ij \in E^c$
$proc_i$	Estimated processing time (walltime) of container $i \in N^c$
$deadline_i$	Execution deadline of $i \in N^c$

Table 2: Symbols and notation used to represent the DC and requests; i and j denote containers while u and v represent DC servers.

chestrator. When a request is not instantly mapped at time sub_i, the waiting time is accounted by the scheduler ($wait_i$). Thus, a request map is valid only if $sub_i + wait_i + proc_i \leq deadline_i$.

Finally, the decision for selecting an appropriated mapping must be guided by the providers' own objectives. DC consolidation, decrease average makespan, decrease energy-consumption, improve Quality-of-Experience (QoE), and increase reliability are examples of objective functions. We argue such information is critical, and should be refined by the infrastructure administrator. Each dimension composing the objective function (*i.e.*, cost, energy consumption, and network usage) should be configurable with specific weights. In this sense, we propose in the following a configurable architecture, based on MCDM methods.

3.2. MCDM methods

Among the existing methods to solve MCDM, we selected Analytic Hierarchy Process (AHP) [21] and Technique for Order Preference by Similarity to Ideal Solution (TOPSIS) [22], chosen due to their multidimensional analysis, being able to work with several servers simultaneously. Also, AHP and TOPSIS provide structured methods to decompose the problem and to consider trade-offs in conflicting criteria. Specifically regarding to the network bandwidth requirements, the sum of all bandwidth capacity bw^s_{uv} with source on u is accounted and included on c^s_u.

AHP is a MCDM algorithm which hierarchically decomposes the problem to reduce the complexity, and performs a pairwise comparison in order to rank all alternatives. In short, the hierarchical organization is composed of three main levels. The objective function of the problem is noted at the top of the hierarchy, while the set of criteria is placed in the second level. Finally, the third level represents all the viable alternatives to solve the problem. In turn, TOPSIS is based in the shortest Euclidean Distance from the alternative to the ideal solution, and handles a large number of criteria and alternatives while requiring a small number of qualitative inputs when compared to AHP.

AHP and TOPSIS are both guided by a weighting vector to define the importance of each criteria. In other words, a vector $W = \{\alpha_0, \alpha_0, ..., \alpha_{|R|-1}\}$ enables the configuration based on the DC administrator's perspective, in which $\sum_{i \in R} \alpha_i = 1$. Table 3 presents three weight configuration representing distinct objective functions traditionally used in the specialized literature (discussed in Section 2). Four criteria can be configured (CPU, RAM, DC fragmentation, and bandwidth). CPU and RAM guide the DC servers selection, while bandwidth is used to find an appropriated DC path. The fragmentation's metric accounts the ratio of active DC resources by the total number of resources.

The flat configuration applies the same importance level for all criteria. The focus of clustering configuration is to consolidate containers atop DC servers as well as to reduce the use of physical links. In this sense, the fragmentation metric receives a weight configuration of 50%. Finally, the network QoS config-

uration gives more importance to network bandwidth configuration guiding the scheduler's decision.

Objective	**CPU**	**RAM**	**DC fragmentation**	**Bandwidth**
Flat	0.25	0.25	0.25	0.25
Consolidation	0.17	0.17	0.50	0.16
Network QoS	0.17	0.17	0.16	0.50

Table 3: Weighting configurations for AHP and TOPSIS.

We reinforce, the weights assigned to each criterion (Table 3) can be adjusted by the administrator to your specific needs. New configurations can be easily incorporated to allow comparison with other works as well as to test new scenarios.

3.3. Queueing policies

Assigning priorities is not trivial, and their choice have a direct impact on performance and DC usage [2]. A DC administrator can select a queueing policy (ϕ) to order the requests before submitting them to the scheduler. We extended traditional queueing policies in order to cope with container requests using intervals (minimum and maximum values). A DC administrator can select the appropriated one given the allocation objective and the scheduling algorithm (Section 5.5). Table 4 contains the queueing policies considered in this work.

FCFS considers only the submission time (sub_i) for ordering the requests, while Smallest Estimated Processing Time First (SPF) orders by estimating the execution time of each request (ascending order). In turn, Smallest Minimum Resource Requirement First (SQFMin) and Smallest Maximum Resource Requirement First (SQFMax) consider the sum of minimum and maximum requested resources, respectively (ascending order). In addition to resource capacity, Smallest Minimum Resource Area First (SAFMin) and Smallest Maximum Resource Area First (SAFMax) analyse the execution time, while Smallest Re-

Policy	Description	Objective
FCFS	First Come First Served	$\phi = sub_i$
SPF	Smallest Estimated Processing Time First	$\phi = proc_i$
SQFMin	Smallest Mininum Resource Requirement First	$\phi = c_i^{min}$
SQFMax	Smallest Maximum Resource Requirement First	$\phi = c_i^{max}$
SAFMin	Smallest Minimum Resource Area First	$\phi = proc_i \times c_i^{min}$
SAFMax	Smallest Maximum Resource Area First	$\phi = proc_i \times c_i^{max}$
SDAF	Smallest Resource Area First	$\phi = proc_i \times (c_i^{max} - c_i^{min})$

Table 4: Queueing policies.

source Area First (SDAF) is based on range of capacities. Section 5 presents how the choice of a queueing policy has an impact on scheduler performance.

The problem formulation as a graph embedding problem with weights (requirements) on vertices and edges can be reduced to a series of similar NP-hard class problems, such as the multi-way separator problem and the virtual network embedding [8]. MCDM algorithms appear as candidates to deal with multiple criteria (QoS, time, and queuing policies), but it run into the complexity limitations of the problem class. The complexity of computing solutions usually implies high response time, in which the solution is no longer needed. Thus, alternatives based on GPU allow to improve the state-of-the-art with results plausible to be used in real world scenarios.

4. GPU Accelerated Containers Scheduler (GPUACS)

The scheduler's architecture, main modules, and prototyping algorithms are depicted in Figure 2. The figure indicates the core architecture's modules (white color) as well as algorithms selected for implementing a GPU-accelerated prototype (gray). Following the problem formulation (Section 3), the architecture supports 7 queueing policies for ordering the requests, while the core of the scheduler comprises MCDM methods, multicriteria weighs, clustering algorithms and analysis of time constraints. The main core components are detailed in the following sections. It is worthwhile to note that GPUACS proposes a modular and configurable architecture. Although the discussion indicates a set of previous selected algorithms for composing the GPUACS prototype, other choices are easily adaptable.

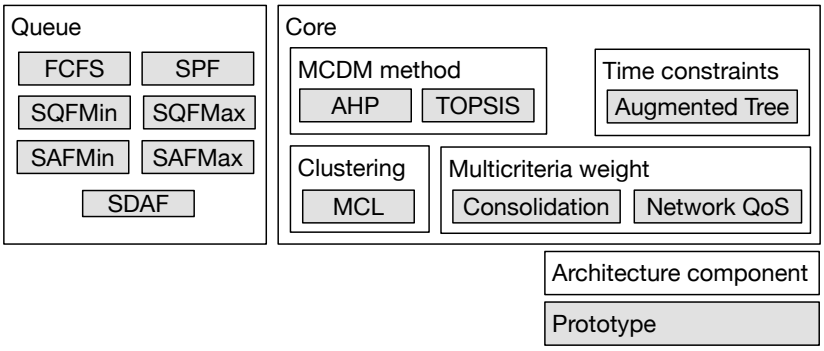

Figure 2: Scheduler main modules and prototyping algorithms.

4.1. DC clustering

Concerning the scalability goal of GPUACS, a data cluster technique is used to decrease the number of comparisons when analyzing DC candidates and request's components. Among the classical algorithms for clustering data and graphs, we selected Markov Cluster Algorithm (MCL) [23] for composing the prototype. MCL is a clustering algorithm specific to graphs based on edges flow and Markov chain. The GPU implementation [9, 20] drastically reduced

the execution time of traditional scheduler's algorithms enabling the analysis of large-scale DC topologies.

Initially, GPUACS creates groups of servers based on computing and networking characteristics. Latter, the groups are given as inputs for the MCDM module as well as the request's requirements. The MCL clustering algorithm has a direct impact on scheduler speedup on large-scale topologies, while for small-scale clustering induces a processing overhead (the analysis is detailed in Section 5). In this context, GPUACS prototype uses a threshold (DC servers ≥ 1024) to verify if MCL must be activated.

4.2. GPU-tailored AHP

The AHP algorithm (described in Section 3.2) was restructured for Single Instruction Multiple Data (SIMD) execution. In short, all matrices NxN were converted to vectors (N^2), and shared-memory techniques were applied, requiring adaptations in the AHP traditional algorithm. First, two vectors (M_1 and M_2) are populated combining criteria and alternatives (second and third levels of AHP hierarchy) while applying the weights defined in Table 3. In other words, $M_1[v] = W[v]; \forall v \in R$ while $M_2[v \times |N^s| + u] = c_u^s[v]; \forall u \in N^s; \forall v \in R$.

Following, a pairwise comparison is executed for all hierarchy levels to selected the appropriated candidate server ($x \in N^s$). For $M_1[v \times |R| + u] > 0$, $M_1[v \times |R| + u] - M_1[x \times |R| + u]$ is selected; if $M_1[v \times |R| + u] \leq 0$, $\frac{1}{M_1[v \times |R| + u] - M_1[x \times |R| + u]}$ is selected; and 1 otherwise. The same rationale is applied for M_2 with index $v \times |N^s|^2 + x \times |N^s| + u$. Finally, both vectors are normalized.

Successively, the algorithm calculates the local ranking of each element in the hierarchy (composing $L1$ and $L2$) as given by Equations (1) and (2), $\forall u, v \in R; \forall y, w \in N^s$. Finally, the global priority (PG) is account for all servers as given by $PG[v] = \sum_{x \in N^s} P_1[v] \times P_2[v \times |N^s| + x]$.

$$L_1[v \times |R| + u] = \frac{\sum_{x \in R} M_1[v \times |R| + x]}{|R|} \quad (1)$$

$$L_2[v \times |N^s| + w] = \frac{\sum_{x \in N^s} M_2[v \times |N^s|^2 + y \times |N^s| + x]}{|N^s|} \quad (2)$$

Concerning the GPU implementation, 5 kernels were implemented to compute the AHP, managed by a centralized CPU thread. Details on GPU memory usage are provided in Section 4.6.

4.3. GPU-tailored TOPSIS

The GPU kernels of TOPSIS follow the data structure from AHP (N^2 vectors), nevertheless the TOPSIS algorithm enables the use of other parallel techniques such as parallel reductions and avoidance of memory conflicts. In order to implement TOPSIS for GPU execution, 11 kernels were developed, coordinated by a CPU thread.

TOPSIS requires 5 steps to rank the DC servers. Initially, an evaluation vector M correlates DC resources (N^s) and scheduling criteria (R), in which $M[v \times |N^s| + u] = c_u^s[v]; \forall u \in N^s; \forall v \in R$, which is later normalized. The next step is the application of the weight configuration over the M values, $M[v \times |N^s| + u] = M[v \times |N^s| + u] \times W[v]; \forall u \in N^s; \forall v \in R$.

Following, two vectors are composed with the minimum and maximum values for each criteria, obtained from M. Specifically, A^- comprises the minimum values while A^+ denotes the maximum. Both vectors are used to account the Euclidian Distance from M, composing Ed^- e Ed^+, respectively. Finally, the proximity coefficient vector is calculated for all servers, denoted by $Rank[u] = \frac{Ed^-[u]}{Ed^+[u] + Ed^-[u]}; \forall u \in N^s$, and then the result vector is ordered in descending order, indicating the selected candidates [24]. The prototype implementation is described in Section 4.6.

4.4. GPUACS core algorithms

The core algorithms of GPUACS are summarized by Algorithms 1, 2 and 3. Algorithm 1 presents a general view of the scheduler management, while Algorithms 2 and 3 detail the containers and network allocation, respectively.

Algorithm 1: GPUACS core management module.

Input: G^s, Req
Output: $\mathcal{M}^s, \mathcal{M}^e$

```
1  begin
2      ranking_method ← select_MCDM()
3      groups ← resources_cluster(G^s)
4      for ∀pod ∈ Req do
5          M^t ← GPUACS_containers(N^s, pod, groups, ranking_method)
6          if (M^t == ∅) then
7              postpone(Req)
8              return (∅, ∅)
9          end
10         M^s ← M^s + M^t
11     end
12     M^t ← GPUACS_network(G^s, Req, M^s)
13     if (M^t == ∅) then
14         postpone(Req)
15         return (∅, ∅)
16     end
17     update_DC(G^s, E^s, M^s, M^e)
18     return (M^s, M^e)
19 end
```

In general, Algorithm 1 sequentially analyses all pods (groups of containers) from a given request Req, guided by the configuration previously defined (MCDM method and resources clustering). Indeed, the main rationale indicates an initial selection of DC candidates for hosting containers (detailed by Algorithm 2) followed by the search of DC paths with network QoS assurance (Algorithm 3). Specifically, Algorithm 1 identifies the MCDM option (line 2), and applies the MCL clustering method to create groups of DC servers (line 3). A tenant can organize the containers in pods, defining all containers from a given pod must be allocated atop the same DC server (line 4). When the scheduling is not correctly accounted (lines 6 and 13), then the request is postponed and

empty set is returned. It is worthwhile to mention, the postponing of requests is a DC administrator choice, and when it is not selected, the request is immediately rejected by GPUACS. Once containers are scheduled, the network QoS requirements are ensured by GPUACS (line 12). Thus, when a request is fully analyzed and scheduled, the DC is updated (line 17) decreasing the available capacity of resources (severs and network).

In order to schedule containers, the Algorithm 2 receives the set of DC servers, the pod to be analyzed, the groups of servers (if available), and the MCDM ranking method. In short, Algorithm 2 schedules containers atop DC groups of servers or individual servers, based on the parameterization. The predefined MCDM algorithm is used to rank and compare all options, assuring the QoS requirements. When groups are used (line 2), a small fraction of DC resources is clustered and ranked by the MCDM method (line 3). Otherwise, the entire DC is analyzed (line 13). Moreover, GPUACS must be aware of QoS constraints (lines 5 and 14) considering the minimum and maximum requirements. The function $QoS\_min\_max()$ aims at maximizing the resources capacity for all containers of a given pod that share a DC server. However, to reduce the number of comparisons, capacities values are gradually reduced by 10% from the maximum capacity specified by the request. Finally, when all DC candidates are analyzed and a suitable server is not found, an empty set is returned (line 20).

Afterwards, Algorithm 3 prepares the network paths for hosting the communication requirements between two containers. It is worthwhile to mention that even containers from the same pod (which are scheduler for the same DC server) must have their communication requirements guaranteed by GPUACS. In short, the goal of Algorithm 3 is to guarantee that the DC has enough capacity to provide the QoS requirements. All containers with communication requirements are processed in pairs (lines 2 and 3). A QoS-guaranteed DC path must be available between the servers selected for hosting containers i and j (line 4). The same rationale for containers minimum and maximum requirements is applied for bandwidth requirements (when need, the bandwidth

Algorithm 2: GPUACS containers module.

```
   Input: N^s, pod_i, groups, ranking_method
   Output: M^s
 1 begin
 2     if (groups ≠ ∅) then
 3         for ∀g ∈ ranking_method(groups) do
 4             for ∀u ∈ ranking_method({u ∈ g}) do
 5                 if QoS_min_max(u, pod_i) then
 6                     M^s(i) = u
 7                     return M^s
 8                 end
 9             end
10         end
11     end
12     else
13         for ∀u ∈ ranking_method(N^s) do
14             if QoS_min_max(u, pod_i) then
15                 M^s(i) = u
16                 return M^s
17             end
18         end
19     end
20     return ∅
21 end
```

requirement is gradually decrease until reaching the minimum value). If a path P is available, the communication mapping is updated (line 9), otherwise, an empty set is returned.

Regarding to the GPU accelerations, the first step is observed on Algorithm 1 at line 4. All pods are parallel processed by GPUACS. In addition, the *ranking_method*() is executed in parallel for all DC candidates at lines $3, 4, 5, 13$ and 14 of Algorithm 2. In turn, the *widest_path*() from Algorithm 3 uses the GPU SIMD architecture to analyze all paths from a DC. This function is a modified version of Dijkstra's algorithm to find a path between servers for hosting containers i and j with QoS requirements. The *widest_path*() GPU kernel executes multiple parallel threads with different pairs of source and destination DC servers. In order to reduce the GPU memory use, the algorithm was structured

Algorithm 3: GPUACS network module.

```
Input: G^s, Req, M^s
Output: M^e
begin
    for ∀i ∈ N^c do
        for ∀j ∈ N^c|ij ∈ E^c do
            P ← widest_path(M^s(i), M^s(j), bw_ij^min, bw_ij^max)
            if P = ∅ then
                return ∅
            end
            else
                M^e(ij) ← uv|∀uv ∈ P
            end
        end
    end
    return M^e
end
```

as an undirected graph. In other words, the algorithm stores $\frac{N \times (N-1)}{2}$, and given two servers u and v, where $u < v$, the links $u \rightarrow v$ and $v \rightarrow u$ are equal.

The functions related with verification of QoS requirements, $QoS\_min\_max()$ and $widest\_path()$, must analyze the reservation and capacity profiles of each server and links. A profile is used to represent the reservations over time, as well as the residual (available) capacity of each resource (*e.g.*, RAM, CPU, and bandwidth). This hard constraints impose a computational challenge for GPUACS, as discussed in the following section.

Finally, regarding the algorithm complexity, the Algorithm 1 has $\mathcal{O}(1)$ operations, except the grouping technique ($\mathcal{O}(|N^s| * |R|^2|)$ for MCL), while Algorithms 2 and 3 are dependent on the MCDM selection. Specifically, the Algorithm 2 is detailed as $\mathcal{O}(|pods| * |N^s| * MCDM)$, and Algorithm 3 is summarized as $\mathcal{O}(|N^s| * |E^s| * (|N^s| * log|N^s| + |E^s|))$.

4.5. Time-constrained scheduling

GPUACS supports two ways of working with time constraints, with and without requests re-submission. The DC administrator is responsible for select-

ing the desired configuration, which can be changed at any time. In short, the method without re-submission immediately rejects requests when any criteria (QoS or time) can not be guaranteed, while the method with re-submission postpones requests for future reanalysis.

For speeding up the processing of time-constrained requests, GPUACS relays on a data structure termed Augmented Tree (AT). The structure is inspired on traditional binary and Adelson-Velsky and Landis Tree (AVL) trees and optimized to process time intervals. Different from traditional AVL, the AT uses time intervals as keys and enables duplicated keys.

A set of $|R|$ trees are created for each DC resource. Specifically, each DC server and link resource $r \in R$ (*i.e.*, CPU, RAM, and bandwidth) has a dedicated tree, and the set of all trees compose an Augmented Forest (AF). A node from the tree is composed of a tuple $< start, end, allocated, left\_node, right\_node >$, where *start* and *end* denote the period under analysis, and *allocated* indicates the physical capacity already reserved for hosting containers. Thus, $left\_node$ and $right\_node$ are pointers to control the tree structure.

The insertion, removal, and balancing operations are similar to the AVL structure, while the availability of capacity at a given time instant is verified as overlapping intervals. DC resources min and max values are defined by the administrator, while for a request for $i \in N^c$, the min is given by the submission time (sub_i) and the max by the estimated ($sub_i + proc_i$) or rigid ($deadline_i$) execution time.

4.6. Prototype implementation and target hardware

A proof-of-concept implementation of GPUACS was developed with C++ and NVIDIA CUDA toolkit 10.2, compiled with GCC 8.3. The code was optimized (memory limits and bandwidth) to a GPU NVIDIA RTX $2080TI$ (11GB), hosted by a server equipped with Intel $i7-8700K$, 32GB RAM DDR4 (3200MHz). Specifically, the block size follows the GPU warp configuration, and all matrices-based kernels used single global matrices, to reduce the number of data movement between threads and GPU memory, as well as to fit

large DC and request structures. Eventually, shared memory is used to transfer data between threads from a single block. The implementation decision (global memory) reduces the speedup improvement in order to obtain scalability. GPU kernels are instantiated up to 3 dimensions, and pinned memory is used to accelerate data movement from host to GPU device. Finally, an implementation of a discrete event simulator go along employing the GPUACS prototype.

5. Evaluation of GPUACS performance

We evaluate GPUACS regarding the three main challenges addressed by the present work: joint containers and network requirements allocation; time-constrained requests; and DC dimensionality and scheduler's scalability. Figure 3 summarizes the evaluation details (comprising goals, metrics, and algorithms) and is used along this section to guide the discussion.

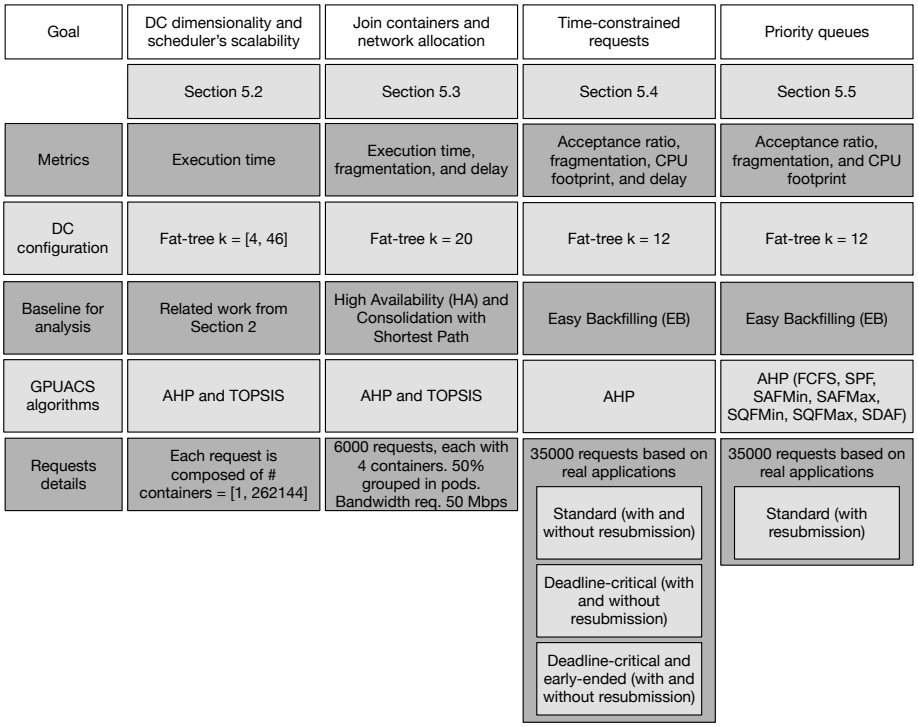

Figure 3: Summary of evaluation protocol, metrics, configurations and algorithms.

Each column from Figure 3 denotes the parameterization for a specific evaluation goal. Initially, large-scale DC topologies and containers requests are analyzed to highlight the performance improvement (execution time) from GPU-tailored implementations (Section 5.2). Following, the network requirements (bandwidth and latency - represented by pods) are introduced to deepen the analysis (Section 5.3). A set of time-constrained requests are prepared and submitted comparing the performance of GPUACS AHP with the traditional Easy Backfilling algorithm (Section 5.4). Finally, the impact of priority queues is individually discussed in Section 5.5. It is worthwhile to mention that GPUACS is compared with distinct algorithms, previously proposed by the specialized literature, and multiples metrics are quantified to corroborate the innovations claimed by GPUACS, as well as to point out eventual limitations.

5.1. DC topology and experimental protocol

The evaluation considers a DC composed of homogeneous servers equipped with 24 cores, 256GB RAM and interconnected by a Fat-Tree topology [11] (Clos-based) and bandwidth capacity of 1Gbps for all links. In a Fat-Tree, the number of servers is given by $k^3/4$, where k is used to guide the topology organization in terms of switches, network links, and servers. In order to compose the experimental protocol, five metrics were selected to analyze the performance of GPUACS regarding the contemporaneous scheduling challenges.

- *Execution time*: The scheduler's scalability is accounted by the execution time required to process large-scale requests and DC configurations. The analysis aims at finding the upper-bound limit of GPUACS executed on a single GPU hardware.

- *Delay*: The container's requests submitted to GPUACS may experience a delay on starting time due to DC occupation (given by Fragmentation and Footprint metrics). This metric gives insights on efficiency of queuing policies and time-constrained jobs.

- DC *fragmentation* and *footprint*: The fragmentation of DC servers (links) is given by the ratio of active servers (links) by the total number of servers (links). A DC resource is considered active when is hosting at least a container (or used to delivery QoS communication requirements). In turn, the DC footprint accounts the resources (*e.g.*, CPU, RAM, and network bandwidth) used to host a set of containers requests. Combined, the metrics give a perspective on joint containers and network requirements allocation as well as DC consolidation techniques.

- *Acceptance Ratio (AR)*: GPUACS only schedule a request if all QoS requirements can be provisioned. Eventually, requests are postponed (analyzed by the delay metric), or definitely rejected. In this sense, the acceptance ratio is the proportion of requests accepted to be scheduled.

For each experiment set, the number of servers composing the DC and the requests configurations were adapted to represent the analysis goal. Finally, for each scenario, 10 executions were performed.

5.2. Evaluation of execution time and scalability

The first set of experiments is related with DC dimensionality and scalability challenges. The scalability of a scheduler is quantified by the number of server-container pairs that can be analyzed in an acceptable computational time. In this sense, the scalability of GPUACS is measured in terms of execution time to process requests atop small- and large-scale topologies. Specifically, two scheduling dimensions are jointly analyzed: the number of containers composing a request and the number of servers in a DC.

The first dimension, number of containers composing a request, varies between 1 and $262,144$ containers, while the second dimension follows a Fat-Tree topology configuration with k between 4 and 46. The configuration is limited ($262,144$ containers in a single request and $24,334$ servers composing a DC) by GPU memory size (detailed in Section 4.6). As all containers follow an homogeneous configuration with low QoS requirements, and all requests are accepted

by GPUACS. In this sense, the goal of this experiment is to quantify the GPU execution time for processing requests.

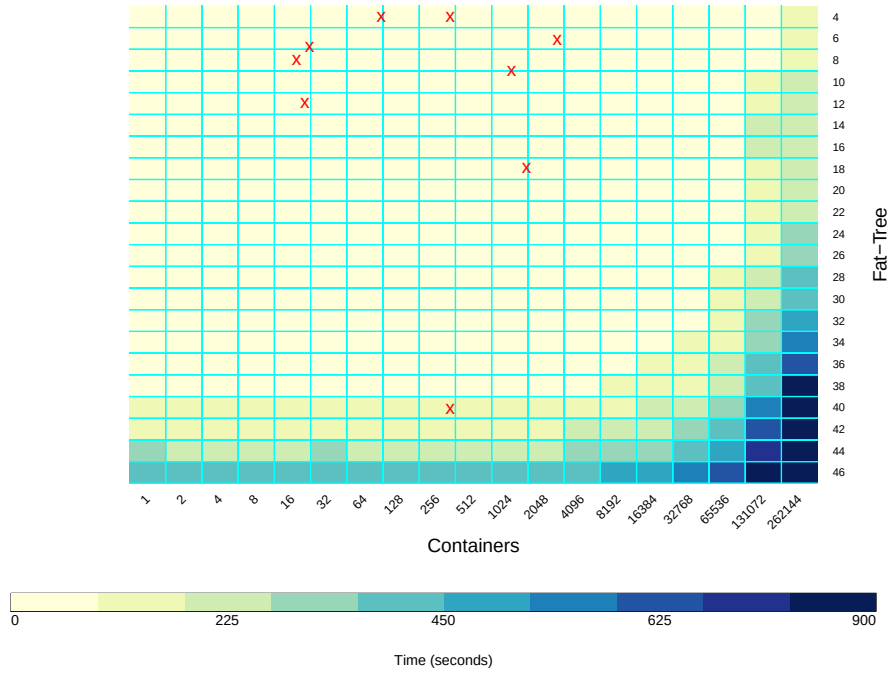

Figure 4: Processing time of GPUACS MCDM AHP. The red x symbols represent the configuration used in experimental evaluation of related work (Section 2).

Figures 4 and 5 present results for AHP and TOPSIS, respectively. The results are presented as heat-maps. The number of containers composing a request evolves on the x-axis, while the Fat-Tree configuration k increases on y-axis. Each scenario was executed 10 times, and the heat-map presents the distribution of the execution time using a color scale. Arbitrary, the execution time of GPUACS was limited to 900 seconds.

Before starting the discussion about GPU-accelerated results, it is important to highlight the limits from CPU-tailored AHP and TOPSIS algorithms.

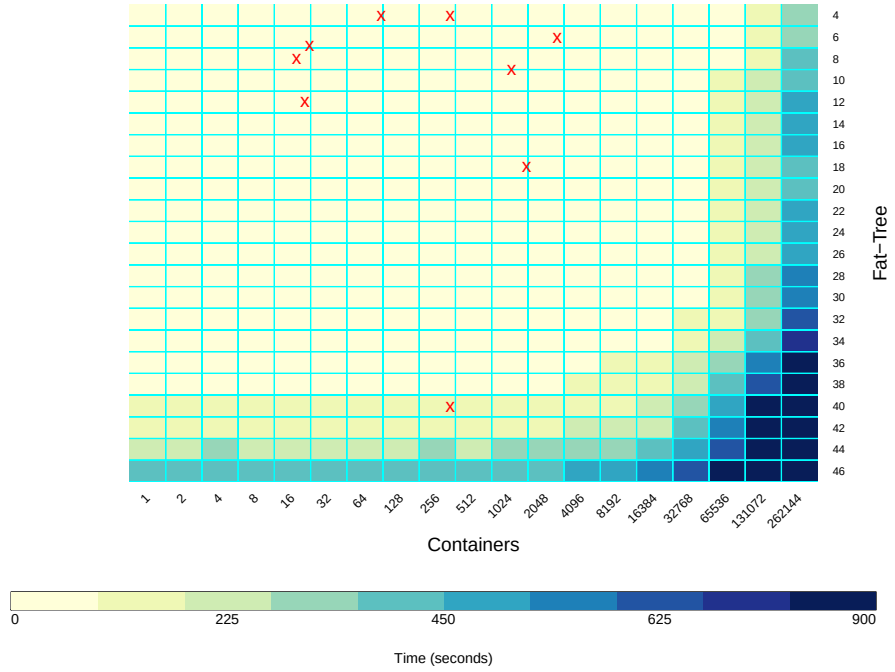

Figure 5: Processing time of GPUACS MCDM TOPSIS. The red x symbols represent the configuration used in experimental evaluation of related work (Section 2).

CPU-only experimentation atop the hardware described in Section 4.6 founds the scalability barrier for both MCDM methods with $2,048$ containers scheduled on a $k = 16$ Fat-Tree topology ($1,024$ servers). In addition, the red symbols represent the configuration used in experimental evaluation of related work (Section 2). It is evident that GPUACS deals with large-scale dimensions: $128x$ for requests and $23x$ for DC servers.

The light-color area from Figures 4 and 5 indicate that both AHP and TOPSIS methods offered by GPUACS scheduled requests atop large-scale topologies up to few seconds. Specifically, AHP method outperforms TOPSIS as dimensionality increases, as observed for Fat-Tree configurations larger than

$k = 36$ and requests composed with more than $16,384$ containers. Finally, with regarding the scalability challenge, it is evident that GPUACS stands out compared to specialized literature (revised in Section 2) regarding the scheduler's scalability.

5.3. Evaluation of network-aware scheduling

This evaluation set investigates the performance of GPUACS for allocating a set of requests with containers and network requirements. Network-related QoS requirements are key aspects that must be considered when scheduling containers atop a DC. However, as previously motivated in Section 3 the introduction of network requirements exacerbates the number of comparisons needed to find a suitable allocation. In this sense, after demonstrating the scalability of GPUACS (Section 5.2), the purpose of this experiment is to analyze the quality of the allocations. Specifically, $6,000$ requests are submitted to be scheduled atop a $k = 20$ Fat-Tree based DC. Each request is composed of 4 containers with a running time up to 250 events (uniformly distributed), while simulations run until all requests processing are complete. Up to 50% of containers from a single request are grouped in pods (latency-sensitive containers), while the bandwidth requirement between a pair of containers is configured up to 50 Mbps (a heavy network requirement).

Two algorithms were selected as baseline for comparisons aiming individually for consolidation or High Availability (HA) (spreading) of containers on DC. Both algorithms are offered by containers orchestration frameworks as default schedulers (*e.g.*, Kubernetes and Docker Swarm). Using GPUACS, the AHP and TOPSIS weighting tables were configured with the network QoS objective function described in Table 3 (high importance level for communication requirements). It is worthwhile to mention, HA and consolidation algorithms were slightly improved to select a viable network path with QoS requirements (shortest path).

Results are summarized by Table 5 and Figure 6. Initially, Table 5 corroborates the results from Section 5.2 highlighting the average runtime for each

algorithm. Both GPUACS algorithms (AHP and TOPSIS) executed in less than 7 seconds. On this scenario, TOPSIS runtime outperformed all comparing algorithms with an acceleration of approximately $22x$ in the foremost case.

Algorithm	**MCDM configuration**	**Runtime (s)**
Consolidation	-	79.38
HA	-	47.80
AHP	Network QoS	6.90
TOPSIS	Network QoS	3.48

Table 5: Average runtime for AHP, TOPSIS, Consolidation and HA algorithms.

Figure 6 deepens the discussion by summarizing the impact of network and server fragmentation on requests delays. Each time a request is not allocated by the scheduler (lack of DC resources to guarantee QoS), it is delayed for next discrete simulation event, and the results demonstrate the impact of DC fragmentation on requests delays. A consolidation algorithm tends to decrease servers fragmentation and consequently the network fragmentation (a shortest path algorithm), while a HA approach tends to spread resources atop the DC, activating multiple servers (the steps on Figure 6) and composing longer network paths. The consolidation algorithm sequently activate the servers, while the HA can allocate a DC server for hosting a single container. Thus, the consolidation algorithm delays multiple requests even for an under-utilized DC (lower fragmentation), in which the HA gradually increases the delay.

Both GPUACS algorithms decreased the requests delays even for a fragmented DC (using almost all servers). The network QoS impact is observed on network fragmentation. While consolidation and HA algorithms have well-defined profiles, AHP and TOPSIS are malleable approaches. In short, GPUACS accelerate the network fragmentation (between 30% and 40%) to guarantee the QoS requirements. While TOPSIS delayed some requests with lower fragmentation values, the AHP algorithm keeps allocations without delay until DC

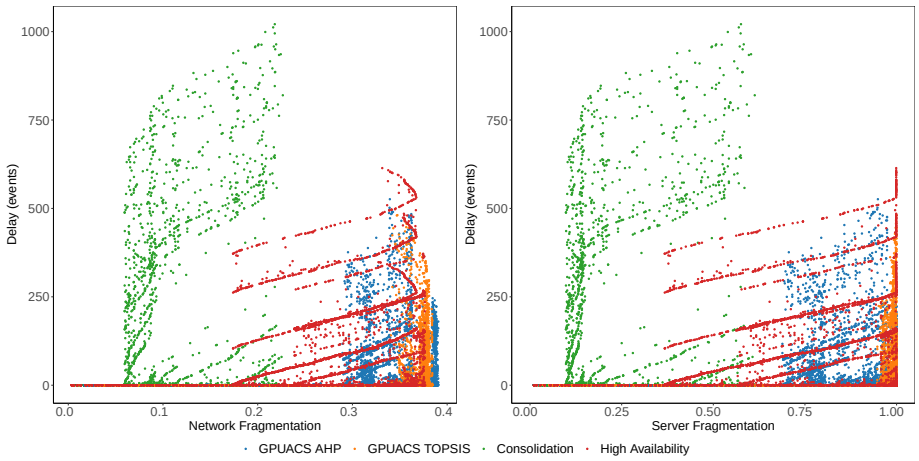

Figure 6: The impact of server and link fragmentation on requests delay.

resources are saturated. The MCDM methods outperformed the traditional consolidation and high availability algorithms guided by the network QoS requirements. Specifically, a distributed application scheduled with GPUACS will receive network QoS guarantees during all runtime, acting on a critical container's management challenge [5] and improving the user's perspective.

5.4. Evaluation of time-constrained requests

This set of experiments investigates the application of GPUACS to schedule time-constrained jobs. As indicated by the previous analysis, AHP outperformed TOPSIS for the scenarios analyzed (Sections 5.2 and 5.3), and is selected as the GPUACS MCDM for the evaluation of time-constrained requests, configured with Network QoS weights (from Table 3). As detailed in Figure 3, this scenario comprises a deep analysis comparing GPUACS with a well-know scheduling algorithm and three requests configurations. GPUACS is compared with Easy Backfilling (EB), a traditional algorithm in the literature already applied in commercial schedulers (*e.g.,* IBM, CTC SP2, HPC2N, KTH, LLNL Thunder) [2]. In order to extend ours analysis, two objective functions are given for EB, consolidation and HA (Section 5.3). Aiming a fair comparison regarding the network QoS requirements, the shortest path algorithm was applied after the

execution of EB. Moreover, two modes of requests management are discussed: with and without requests resubmissions (Section 4.5). In short, the scenario using resubmission re-queues unallocated requests to be later reprocessed by the scheduler (when DC is ready to guarantee the QoS requirements).

The containers requests are based on CPU, RAM, and network bandwidth configurations extracted from real applications [25, 26]. CPU and RAM are normalized and selected in intervals, $[0.1, 1]$ and [25MB, 1GB], respectively, while bandwidth is defined by a strict minimum QoS configuration selected from 0.16Mbps, 5.09Mbps, 0.71Mbps, 4.08Mbps, 2.95Mbps, and without connectivity. A set of $35,000$ requests composed of 4 containers were uniformly generated based on those previous selected values and intervals. Each request executes up to 200 simulation events, and all scenarios are analyzed using the same interval of simulation. Finally, the DC is constructed as $k = 12$ Fat-Tree.

The time-constrained analysis is decomposed into three scenarios: *(i)* Standard requests: this scenario analyzes the traditional requests as explained in Section 3.1. Each request specifies a walltime that is perfectly executed, without preemption or early terminations (*i.e.*, process errors or anticipated successful executions). Moreover, requests are deadliness. *(ii)* Deadline-critical requests: each request specifies a hard execution deadline. Besides executing a QoS-aware allocation, the schedulers (GPUACS and EB) must respect the walltime and deadline information, which are precisely specified on the request. *(iii)* Deadline-critical and early-ended requests: this scenario represents a complex execution composed of hard deadlines and inaccurate walltime prediction. It is worthwhile to mention, the walltime is a user-specified information that only indicates to the orchestrator an estimation on execution time.

The results are summarized by Table 6 and Figures 7- 9. While Table 6 informs the Acceptance Ratio (AR) for all scenarios, the network fragmentation, CPU footprint and scheduling delay (events) are summarized by Cumulative Distribution Function (CDF) graphs. In the following, each workload scenario is individually analyzed.

Requests	Resubmission	Algorithm	AR
Standard	Without	EB HA	52.40%
		EB Consolidation	55.14%
		GPUACS AHP	97.05%
	With	EB HA	100.00%
		EB Consolidation	100.00%
		GPUACS AHP	100.00%
Deadline-critical	Without	EB HA	69.58%
		EB Consolidation	56.49%
		GPUACS AHP	98.68%
	With	EB HA	20.55%
		EB Consolidation	29.23%
		GPUACS AHP	100.00%
Deadline-critical and early-ended	Without	EB HA	45.69%
		EB Consolidation	59.15%
		GPUACS AHP	97.43%
	With	EB HA	32.10%
		EB Consolidation	100.00%
		GPUACS AHP	100.00%

Table 6: Summary of EB and GPUACS acceptance ratio (AR).

5.4.1. Standard requests

The execution without resubmission of requests represent a traditional batch allocation. In this sense, the Acceptance Ratio (AR) metric is used to quantify requests that were successfully scheduled. Results from Table 6 indicate GPUACS AHP allocated almost all requests, while EB achieved 55.14% with consolidation technique. In addition, EB and GPUACS AHP allocated all requests when resubmission is allowed. The impact of both methods are deeply

illustrated by Figure 7.

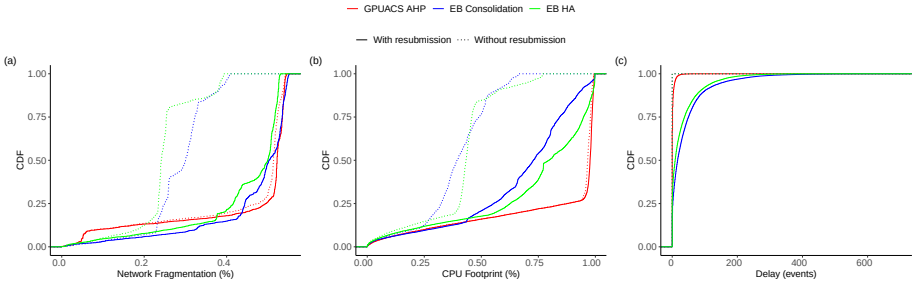

Figure 7: Results for EB and GPUACS AHP for standard requests.

The network fragmentation, CPU footprint, and delay are given summarized by Figures 7(a), Figures 7(b), and Figures 7(c), respectively. Regardless the objective function (consolidation or HA), EB has lower AR values, and consequently reduce the DC usage, as observed by network fragmentation and CPU footprints. In turn, GPUACS AHP has competitive AR values with and without resubmission, consequently increasing the DC usage. It is worthwhile to mention that even for high AR values (with and without resubmission), GPUACS AHP drastically reduce the requests delay. EB with HA obtained the minimum total delay, however it allocated approximately half of the requests.

5.4.2. Deadline-critical requests

A strict time limit requirement is forced for each container uniformly selected in the $[proc_i, proc_i + 0.1 \times proc_i]$ interval, for deadline-critical requests. On this scenario, Table 6 shows only GPUACS AHP accepted all requests during the simulation interval, while EB HA achieved 69.58% and EB consolidation obtained 29.23% without and with requests resubmission, respectively. The efficiency of MCDM is evident in this scenario. When analyzing multiple options (requests and DC resources) simultaneously, a better use of DC is obtained, as indicated by Figures 8(a), 8(b), and 8(c). Moreover, the main reason for EB rejection of requests is the inability to jointly analyze containers and network QoS requirements. Even selecting appropriated DC servers to host containers, EB

with a shortest path approach fails to found DC paths given the QoS objectives.

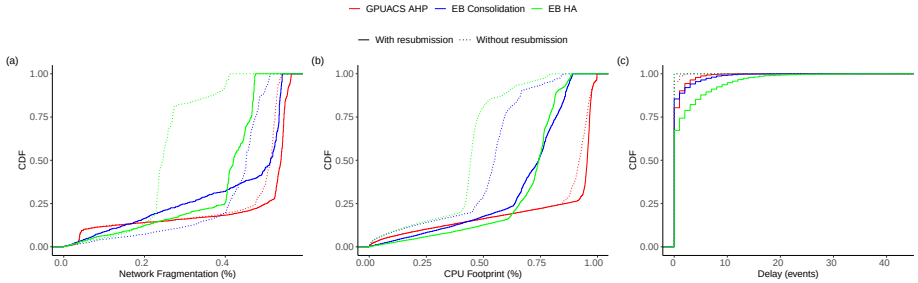

Figure 8: Results for EB and GPUACS AHP for deadline-critical requests.

It is worthwhile to mention the lightly increase on requests delay for GPUACS AHP when comparing both submission methods. Even increasing on 2.57% on AR, the resubmission of requests increased the scheduling delay (Figure 8(c)).

5.4.3. Deadline-critical and early-ended requests

Besides the deadline configuration induced on Subsection 5.4.2, an early end for each container is uniformly selected in the $[0.7 \times proc_i, proc_i]$ interval. An interruption on tasks execution can be originated from different reasons: the task successfully ended, the process was interrupted by an error, the data which would be processed is not available, among others. This scenario brings the problems from real multitenant DCs [27].

The interruption of process execution opens the opportunity to schedule more requests, as observed by the AR values when compared with only deadline-critical scenario (from Table 6). In this sense, EB consolidation and GPUACS AHP allocated all requests when resubmission is allowed. Without resubmission of requests, GPUACS AHP rejected only 2.57% of requests, while EB Consolidation allocated 59.15% at most. The quality of EB and GPUACS AHP allocations is evidenced by Figure 9. Although both algorithms allocated all requests, GPUACS AHP decreases the total delay (Figure 9(a)). However, GPUACS AHP requires more CPU usage (footprint from Figure 9(b)) and more network resources (the fragmentation from Figure 9(a)). This scenario makes evident

the dichotomy between execution delay and DC resources usage. Finally, while the current experiment is based on Network QoS weighting scheme, GPUACS MCDM methods can be configured following the DC administration objectives.

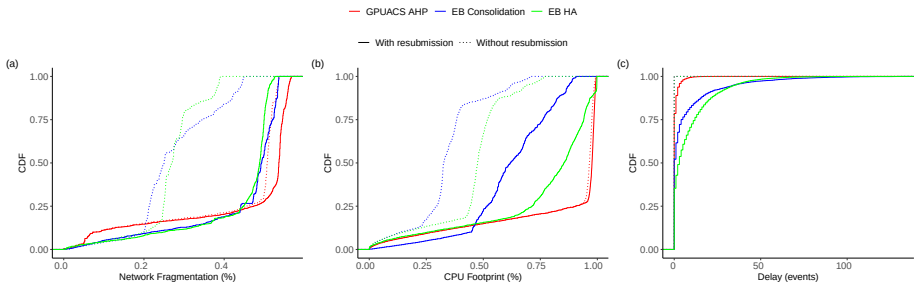

Figure 9: Results for EB and GPUACS AHP for deadline-critical and early-ended requests.

5.5. Priority queues evaluation

For complementing the discussion on containers requests allocation, we investigate the performance of GPUACS AHP when different queueing policies are applied for ordering the requests (the policies are presented in Section 3.3 and the notation is summarized in Table 4). The set of standard requests were submitted for the seven queueing scenarios, without the resubmission method (and consequently without the delay metric).

Table 7 and Figure 10 summarize the Acceptance Ratio, network fragmentation and CPU footprint, respectively. While GPUACS AHP demonstrate a low variation on AR scheduling almost all requests, EB, regardless of the objective function (HA or consolidation), varies from 52.40% for FCFS up to 76.03% for SPF. This fact is justified by the nature of the MCDM algorithm that extensively compares candidates and criteria. In short, regardless the queuing policy, SPF is the preferred option for EB algorithm, with SQFMin emerging as the indicated option for GPUACS AHP.

Figure 10(a) summarizes the network fragmentation while Figure 10(b) indicates the CPU footprint for all algorithms and queue policies. Regardless the policy, GPUACS AHP tends to quickly increase the network usage balancing

Queuing policy	Algorithm	AR
FCFS	EB HA	52.40%
	EB Consolidation	52.40%
	GPUACS AHP	97.05%
SPF	EB HA	76.03%
	EB Consolidation	67.53%
	GPUACS AHP	96.25%
SAFMin	EB HA	60.33%
	EB Consolidation	55.77%
	GPUACS AHP	96.25%
SAFMax	EB HA	60.33%
	EB Consolidation	55.77%
	GPUACS AHP	96.25%
SQFMin	EB HA	56.02%
	EB Consolidation	57.61%
	GPUACS AHP	97.15%
SQFMax	EB HA	56.02%
	EB Consolidation	57.63%
	GPUACS AHP	97.15%
SDAF	EB HA	60.33%
	EB Consolidation	55.77%
	GPUACS AHP	96.25%

Table 7: Summary of GPUACS AHP and EB acceptance ratio with different queueing policies.

the load atop multiple links (high fragmentation). The network fragmentation is lower for both EB algorithms. However, Table 7 indicates that fewer requests are scheduled. The same rationale is applied for CPU footprint. For improv-

ing the AR, the MCDM of GPUACS gradually increases the CPU usage. In conclusion, while the queue policy has a large impact on EB performance [2], the large number of comparisons performed by the MCDM AHP offers better results, regardless the queue policy.

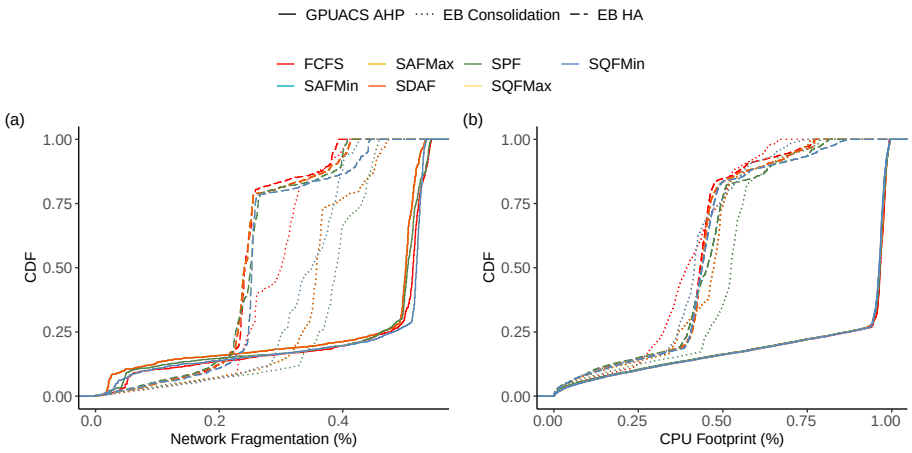

Figure 10: Results for EB and GPUACS AHP with different queueing policies.

5.6. Discussion and Limitations

In order to deepen the discussion on the experimental evaluation, we present the key aspects of GPUACS as well as some identified limitations. Initially, the importance of developing a modular architecture (Figure 2) for GPUACS proved to be appropriate in a perspective of the potential configuration options for the scheduling algorithms (MCDM, EB, or any future implementation), objective functions (*e.g.*, MCDM weighting parameters, consolidation, high availability), and priority queues. A specific analysis on GPUACS prototypes based on AHP and TOPSIS indicated that the challenges previously identified (jointly containers and network scheduling, time-constrained requests, and DC dimensionality and scheduler scalability) were covered paving the road for further discussions. The scalability achieved by GPU implementations (Section 5.2) appeared as a useful alternative to implement centralized schedulers. However, to obtain optimal results, the AHP and TOPSIS codes must be adapted and pruned to fit

the GPU memory specification.

The joint containers and network requirements allocation is efficiently addressed by MCDM. A simple introduction of the bandwidth and latency-sensitive (expressed as pods) requirements increased the allocation ratio. However, an evident limit of any MCDM configuration is the arbitrary and subjective configuration of weighting parameters. Thus, for GPUACS AHP and TOPSIS prototypes, the weighting configuration were empirically defined, but it is essential to emphasize they must be systematically refined to meet the objectives of the DC administrator. It is worthwhile to mention that the GPUACS prototype was based on two consolidated MCDM methods, AHP and TOPSIS, however, the modular architecture is prepared to be extended. In addition, the selection of an appropriated queuing policy is essential for traditional algorithms (*i.e*, EB with consolidation, or HA objective functions), while it is smoothed and simplified by the time-constrained and MCDM analysis performed by GPUACS.

Finally, GPUACS was evaluated considering five metrics: execution time, scheduling delay, DC fragmentation and footprint, and acceptance ratio. Although the analysis of data for all metrics has indicated that GPUACS improved the scheduling in the perspective of DC administrator and users, a future evaluation should consider the QoE of distributed applications (application-oriented metrics).

6. Considerations and future work

Usually, containers orchestrators apply simple scheduling algorithms to optimize the performance and to soften (or avoid) the scalability barriers. Most algorithms analyze few allocation criteria, and usually optimize only one objective function due to the complexity of the problem (NP-Hard). It is a fact there is a dichotomy between the response time to perform the allocation and the approximation of the optimal solution. Moreover, the network requirements are not considered as critical resources by most orchestrators, with a lack of schedulers performing the joint scheduling of containers, communication resources,

and time-based constraints.

Based on this scenario, we addressed relevant aspects through the discussion of the use of the multicriteria approach accelerated by GPUs to carry out container scheduling taking into account network and time constraints. Specifically, we proposed and evaluated GPU Accelerated Containers Scheduler (GPUACS), a new approach to schedule requests atop large-scale DCs.

Our experimental analyzes demonstrated the sensitivity that GPU-tailored MCDM methods have to schedule containers requests considering relevant criteria identified on the problem formulation. As future work, we envisage the implementation of a load balancing module, allowing to optimize the resources already allocated in the DC, as well as a fault tolerante implementation, allowing the scheduler to recover from serious failures. Finally, a future implementation with multiple GPUs requires adaptions to fully explore the memory bandwidth.

Acknowledgments

The research leading to the results presented here has received funding from UDESC and FAPESC, and the European Unions Horizon 2020 research and innovation programme under the LEGaTO Project (legato-project.eu), grant agreement No 780681

References

[1] D. A. Lifka, The anl/ibm sp scheduling system, in: D. G. Feitelson, L. Rudolph (Eds.), Job Scheduling Strategies for Parallel Processing, Springer Berlin Heidelberg, Berlin, Heidelberg, 1995, pp. 295–303.

[2] D. Carastan-Santos, R. Y. De Camargo, D. Trystram, S. Zrigui, One can only gain by replacing easy backfilling: A simple scheduling policies case study, in: 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), 2019, pp. 1–10.

[3] L. Rosa Rodrigues, M. Pasin, O. Correa Alves Jr., C. C. Miers, M. Pillon, P. Felber, G. Koslovski, Network-Aware container scheduling in Multi-Tenant data center, in: 2019 IEEE Global Communications Conference: Selected Areas in Communications: Cloud & Fog/Edge Computing, Networking and Storage (Globecom2019 SAC CCNS), Waikoloa, USA, 2019.

[4] Z. Wang, H. Li, Z. Li, X. Sun, J. Rao, H. Che, H. Jiang, Pigeon: An effective distributed, hierarchical datacenter job scheduler, in: Proceedings of the ACM Symposium on Cloud Computing, SoCC 19, Association for Computing Machinery, New York, NY, USA, 2019, p. 246258. doi:10.1145/3357223.3362728.
URL https://doi.org/10.1145/3357223.3362728

[5] K. Suo, Y. Zhao, W. Chen, J. Rao, An analysis and empirical study of container networks, in: IEEE INFOCOM 2018 - IEEE Conference on Computer Communications, 2018, pp. 189–197.

[6] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, J. Wilkes, Large-scale cluster management at google with borg, in: Proceedings of the Tenth European Conference on Computer Systems, EuroSys 15, Association for Computing Machinery, New York, NY, USA, 2015. doi:10.1145/2741948.2741964.
URL https://doi.org/10.1145/2741948.2741964

[7] M. Rost, C. Fuerst, S. Schmid, Beyond the stars: Revisiting virtual cluster embeddings, SIGCOMM Comput. Commun. Rev. 45 (3) (2015) 1218. doi:10.1145/2805789.2805792.
URL https://doi.org/10.1145/2805789.2805792

[8] M. Rost, E. DÖhne, S. Schmid, Parametrized complexity of virtual network embeddings: Dynamic & linear programming approximations, SIGCOMM Comput. Commun. Rev. 49 (1) (2019) 3–10.

[9] L. L. Nesi, M. A. Pillon, M. D. de Assuno, C. C. Miers, G. P. Koslovski,

Tackling virtual infrastructure allocation in cloud data centers: a gpu-accelerated framework, in: 2018 14th International Conference on Network and Service Management (CNSM), 2018, pp. 191–197.

[10] Y. Guo, W. Yao, A container scheduling strategy based on neighborhood division in micro service, in: NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium, 2018, pp. 1–6.

[11] M. Al-Fares, A. Loukissas, A. Vahdat, A scalable, commodity data center network architecture, SIGCOMM Comput. Commun. Rev. 38 (4) (2008) 63–74. doi:10.1145/1402946.1402967.
URL http://doi.acm.org/10.1145/1402946.1402967

[12] M. Sheikhan, N. Mohammadi, Time series prediction using pso-optimized neural network and hybrid feature selection algorithm for ieee load data, Neural computing and applications 23 (3-4) (2013) 1185–1194.

[13] A. Havet, V. Schiavoni, P. Felber, M. Colmant, R. Rouvoy, C. Fetzer, Genpack: A generational scheduler for cloud data centers, in: IEEE Int. Conf. on Cloud Engineering (IC2E), IEEE, Vancouver, BC, Canada, 2017, pp. 95–104.

[14] C. Guerrero, I. Lera, C. Juiz, Genetic algorithm for multi-objective optimization of container allocation in cloud architecture, Journal of Grid Computing 16 (1) (2018) 113–135.

[15] Y. Hu, H. Zhou, C. de Laat, Z. Zhao, Concurrent container scheduling on heterogeneous clusters with multi-resource constraints, Future Generation Computer Systems 102 (2020) 562–573.

[16] F. R. de Souza, C. C. Miers, A. Fiorese, G. P. Koslovski, Qos-aware virtual infrastructures allocation on sdn-based clouds, in: 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), IEEE, Madrid, Spain, 2017, pp. 120–129.

[17] F. R. de Souza, C. C. Miers, A. Fiorese, M. D. de Assunção, G. P. Koslovski, Qvia-sdn: Towards qos-aware virtual infrastructure allocation on sdn-based clouds, Journal of Grid Computingdoi:10.1007/s10723-019-09479-x. URL https://doi.org/10.1007/s10723-019-09479-x

[18] H. Ben Alla, S. Ben Alla, A. Ezzati, A. Touhafi, An efficient dynamic priority-queue algorithm based on ahp and pso for task scheduling in cloud computing, in: A. Abraham, A. Haqiq, A. M. Alimi, G. Mezzour, N. Rokbani, A. K. Muda (Eds.), Proceedings of the 16th International Conference on Hybrid Intelligent Systems (HIS 2016), Springer International Publishing, Cham, 2017, pp. 134–143.

[19] N. Panwar, S. Negi, M. M. S. Rauthan, K. S. Vaisla, Topsis–pso inspired non-preemptive tasks scheduling algorithm in cloud environment, Cluster Computing 22 (4) (2019) 1379–1396.

[20] L. L. Nesi, M. A. Pillon, M. D. d. Assuno, G. P. Koslovski, Gpu-accelerated algorithms for allocating virtual infrastructure in cloud data centers, in: 2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), 2018, pp. 364–365.

[21] T. L. Saaty, L. G. Vargas, Models, methods, concepts & applications of the analytic hierarchy process, Vol. 175, Springer Science & Business Media, 2012.

[22] C.-L. Hwang, K. Yoon, Methods for multiple attribute decision making, in: Multiple attribute decision making, Springer, 1981, pp. 58–191.

[23] S. M. Van Dongen, Graph clustering by flow simulation, Ph.D. thesis, University of Utrecht, Utrecht, Holanda (2001).

[24] C.-T. Chen, Extensions of the topsis for group decision-making under fuzzy environment, Fuzzy sets and systems 114 (1) (2000) 1–9.

[25] A. H. Marcondes, G. Diel, F. R. de Souza, P. R. Vieira, A. Fiorese, G. P. Koslovski, Executing distributed applications on sdn-based data center: A

study with nas parallel benchmark, in: 2016 7th International Conference on the Network of the Future (NOF), IEEE, 2016, pp. 1–3.

[26] M. Imdoukh, I. Ahmad, M. Alfailakawi, Optimizing scheduling decisions of container management tool using many-objective genetic algorithm, Concurrency and Computation: Practice and Experience (2019) e5536.

[27] L. C. Casagrande, G. P. Koslovski, C. C. Miers, M. A. Pillon, Deepscheduling: Grid computing job scheduler based on deep reinforcement learning, in: L. Barolli, F. Amato, F. Moscato, T. Enokido, M. Takizawa (Eds.), Advanced Information Networking and Applications - Proceedings of the 34th International Conference on Advanced Information Networking and Applications, AINA-2020, Caserta, Italy, 15-17 April [canceled because of the COVID-19 crisis], Vol. 1151 of Advances in Intelligent Systems and Computing, Springer, 2020, pp. 1032–1044. doi:10.1007/978-3-030-44041-1_89.

URL https://doi.org/10.1007/978-3-030-44041-1_89