Using Kubernetes in Academic Environment: Problems and Approaches

(Open Scheduling Problem)

Viktória Spišaková[1], Dalibor Klusáček[2][0000−0001−6434−4433], and Lukáš Hejtmánek[1][0000−0002−2078−8638]

[1] Institute of Computer Science, Masaryk University, Czech Republic
xhejtman,spisakova@ics.muni.cz
[2] CESNET, a.l.e., Prague, Czech Republic
klusacek@cesnet.cz

Abstract. In this work, we discuss our experience when utilizing the Kubernetes orchestrator (K8s) to efficiently allocate resources in a heterogeneous and dynamic academic environment. In the commercial world, the "pay per use" model is a strong regulating factor for efficient resource usage. In the academic environment, resources are usually provided "for free" to the end-users, thus they often lack a clear motivation to plan their use efficiently. In this paper, we show three major sources of inefficiencies. One is the users' requirement to have interactive computing environments, where the users need resources for their application as soon as possible. Users do not appreciate waiting for interactive environments, but constantly keeping some resources available for interactive tasks is inefficient. The second phenomenon is observable in both interactive and batch workloads; users tend to overestimate necessary limits for their computations, thus wasting resources. Finally, Kubernetes does not support fair-sharing functionality (dynamic user priorities) which hampers the efforts when developing a fair scheme for Pod/job scheduling and/or eviction. We discuss various approaches to deal with these problems such as scavenger jobs, placeholder jobs, Kubernetes-specific resource allocation policies, separate clusters, priority classes, and novel hybrid cloud approach. We also show that all these proposals open interesting scheduling-related questions that are hard to answer with existing Kubernetes tools and policies. Last but not least, we provide a real workload trace from our installation to the scheduling community which captures these phenomena.

Keywords: cloud · HPC · scheduling · Kubernetes · resource management.

1 Introduction

In today's world, the usefulness of container-oriented computing is widely recognized. Businesses adopted containers several years ago for their digital services. However, academia started to notice containers as a viable way of supporting research not so long ago.

Historically, demanding computations that process data, produce analyses or deliver results of complex workflows are foundations of research. These actions occur on large high performance computing clusters. Specifically, computations are managed by various scheduling systems because there are more requests on resources than resources themselves. Scheduling methodologies have been in active development for at least 30 years and internal design is finely calibrated to provide a variety of important features such as granular resource selection, placement control or topology and affinity as well as fairness. Scheduling systems can have different optimization goals. Some systems focus on maximizing throughput and resource utilization while users' jobs have to wait in queues. Other systems want to avoid starvation, focusing on low latency, thus running new jobs as quickly as possible which in turn often results in decreased resource utilization. Furthermore, resources are often allocated such that overall fairness among users, groups and/or projects is guaranteed.

Resources for scientific computations in academia are in majority offered for free as a result of financing by national governments that earmark funds for research and education. For this reason, academic resource providers need proper scheduling mechanisms as it is the only way to regulate access to resources. This is in contrast to commercial world where access to resources is paid by users which imposes strong and efficient access regulation. This explains why container orchestrators or cloud-management frameworks do not typically provide truly sophisticated HPC-like schedulers to regulate resource access. Especially, a well-performing scheduler in the commercial world equals high profit and vice versa.

A crucial difference between HPC scheduler logic and container orchestrator is that in the HPC world, submitted jobs typically act as a "finite" computations that start, compute and then finish, whereas container orchestrators (e.g., Kubernetes[3]) were developed to accommodate continuous services and long-running stateless applications [15]. Nevertheless, container orchestrators are actively looking for a way of implementing HPC jobs concept. For example, recent versions of Kubernetes (`v.1.21`) feature `Indexed Jobs`[4] that allow static work partition among the workers of a parallel job. The introduction of such resource marks efforts of Kubernetes developers and community to migrate more of HPC and batch workloads into the platform. None of these "extensions" enforces job runtime limit so as of now, container orchestrators do not forbid endless jobs which makes HPC-like scheduling almost impossible.

Besides potentially endless jobs, there is another category of waiting-sensitive workloads — interactive jobs, i.e., jobs that do not run in batch/background but users interactively work with them. This kind of jobs imposes problems to schedulers even in standard HPC installations used in academia.

Another complication for moving HPC workloads to container platform lies in the modus operandi of this platform. Kubernetes orchestrator (K8s) works with

[3] https://kubernetes.io
[4] https://kubernetes.io/blog/2021/04/19/introducing-indexed-jobs/

the assumption that every `Pod`[5] can be terminated and restarted. In contrast, typical HPC workload does not expect any interruptions once it has started and terminates only when it has either finished or has reached its allocated walltime limit.

In this paper, we discuss our experience and problems observed when utilizing the Kubernetes container orchestrator in academia. We aim to provide Kubernetes platform that will be fully competitive with traditional HPC infrastructure as well as other types of workload. We see this as challenging and interesting task. The benefits of containerized computations are obvious and are becoming very popular among scientists. The challenge is to come up with a solution that allows "free of charge" computing while being robust and self-regulating. On the other hand, we do not aim to "mimic" HPC-like system including complex HPC scheduler on top of Kubernetes infrastructure, and vice versa Kubernetes infrastructure on top of HPC infrastructure.

The rest of this paper is organized as follows. First, in Section 2 we define major scheduling challenges that arise due to the intended use of Kubernetes infrastructure in academic environment. Next, Section 3 presents current scheduling capabilities of Kubernetes. In Section 4, we propose several ideas on how to solve our scheduling and resource allocation issues. We also discuss the details of our infrastructure setup and provide basic data about the workload trace from our installation (Section 5). Finally, we introduce the related work in Section 6 and conclude the paper.

2 Scheduling Challenges

Concept of shared, multi-tenant infrastructure is usually adopted in academic environment. In such environment, proper scheduling is important otherwise vast inefficiencies can appear and lead to resource (money) wasting while irritating the users. We discuss those major obstacles that threaten the success and efficiency of the infrastructure. From infrastructure administrator point of view, we distinguished three major domains of inefficiencies, all coupled with scheduling.

2.1 Endless Computing with Limited Resources

In a standard HPC batch system, each job has a maximum allowed lifetime — the so called walltime limit [17]. Achieving time limit for a workload is not always simple, e.g., in interactive workloads or other long-running services. Therefore, the runtime of a container/Pod is unbounded and unknown, in general [10]. This means that the system (and its users) do not expect the workload will terminate after predefined time. Since the academia is not using pay-per-use model, we clearly lack a clear motivation for the users to terminate their containers/Pods once they are not needed anymore.

[5] Pods are the smallest deployable units of computing that you can create and manage in Kubernetes.

At the same time, academia budget is fixed (as users use it for free), i.e., we cannot simply buy another cluster whenever the demand is approaching the available capacity. The absence of the pay-per-use model together with no clear resource-reclaiming policy will inevitably cause another obvious problem — existing resources will be allocated to the users without considering overall fairness. This is in great contrast with common batch HPC installations, where fairness is one of the major optimization goals and is usually enforced by the well known fair-sharing approach [11].

Job starvation is tightly connected to the absence of walltime limit. Users can submit jobs demanding resources that cannot be satisfied because all nodes are occupied by jobs without finite walltime. The problem is more visible with standard Kubernetes scheduler as it is not able to reserve a node for a large job. Without reservation, large job is endlessly preempted by smaller jobs. There is an ongoing work[6,7] regarding the problem but it is not present in currently used Kubernetes versions.

2.2 Interactive Computing

In HPC environment, batch jobs often do not start immediately but reside in a queue and wait until a cluster has enough free resources to execute the job. However, we are witnessing rising popularity of user interest in working with graphical interfaces rather than command line. In the past, we deployed a web portal *Open OnDemand*[8] as the first step of improving interactive applications' accessibility. Figure 1 shows the growing demand of GUI applications in CERIT-SC, supporting the argument that users seek more comfortable ways of working with graphical interfaces, i.e, interactive use-cases will not likely disappear.

Interactive jobs comprise both interactive CLI jobs, i.e., jobs running purely from command line, and more importantly interactive GUI jobs. The two types should be scheduled and started as soon as possible because they require user interaction to run, e.g., selecting options or filling in password in the GUI. If this is the case, the user has to wait but the job may start after such a long time that the user is not available anymore, thus the job is only blocking resources. A common solution to the *waiting-user* problem is to keep some resources unoccupied so interactive jobs start nearly immediately, thus keeping users active and responsive. On the other hand, resources are likewise blocked needlessly so both situations impose ineffective usage of resources.

One solution suggesting itself is suspending non-interactive jobs in order to accommodate interactive ones. Suspending a job would mean to free its resources and thus letting interactive job run. After interactive job is completed, dormer non-interactive job resumes. Unfortunately, neither HPC nor Kubernetes allow to suspend running jobs which makes interactive computing problematic both in HPC and Kubernetes.

[6] https://github.com/kubernetes/kubernetes/issues/86373
[7] https://github.com/kubernetes/kubernetes/issues/100347
[8] https://openondemand.org

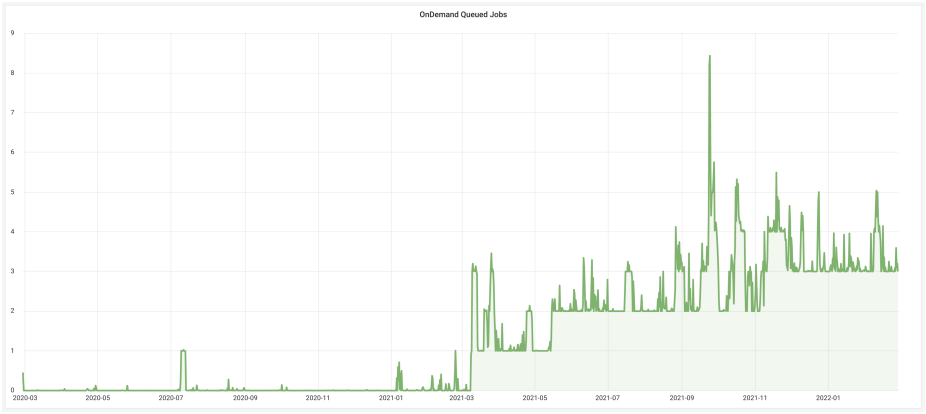

Fig. 1. Time series graph depicting rising popularity of applications spawned from OnDemand portal.

2.3 Overestimation

Specifying precise job resource request is a key precondition for effective job scheduling. This is a well understood requirement in HPC world but container orchestrators are not notably strict about the necessity to specify job resources. Even if users specify job resources, there are no guarantees how exact the specification is. As a result, users tend to significantly overestimate resource requests which has several reasons.

First reason is the sheer obliviousness to the concept and the logic behind resource specification — some users can not envision abstract units as RAM GB or units of CPU so they are not capable of setting sensible value. Second reason is their fear of job exceeding allocated resources (causing job termination) which leads to specifying substantially more resources than needed in order to avoid the situation. Last but not least, computations are sometimes characterized by dynamic variation when most of the time, resource utilization is low but for a short time period, perhaps for more complex part of computation, resource consumption spikes. This can result in specifying fundamentally more resources than needed, although the correct practice could probably be to split the job into several units with tailored requirements.

User-induced overestimation causes very low real cluster usage. Here, resource oversubscription (allocating more resources than the physical capacity) is a crucial enhancement to existing systems [2]. According to our experience the *overestimation problem* is more coupled with K8s workloads than HPC where the actual usage-to-request ratio is quite good[9].

As we show in the Figure 2, users are prone to significantly overestimate Pods' resource requests which in turn leads to inefficient use of those resources. In the

[9] In our system, HPC workloads typically utilize more than 80% of requested CPU resources.

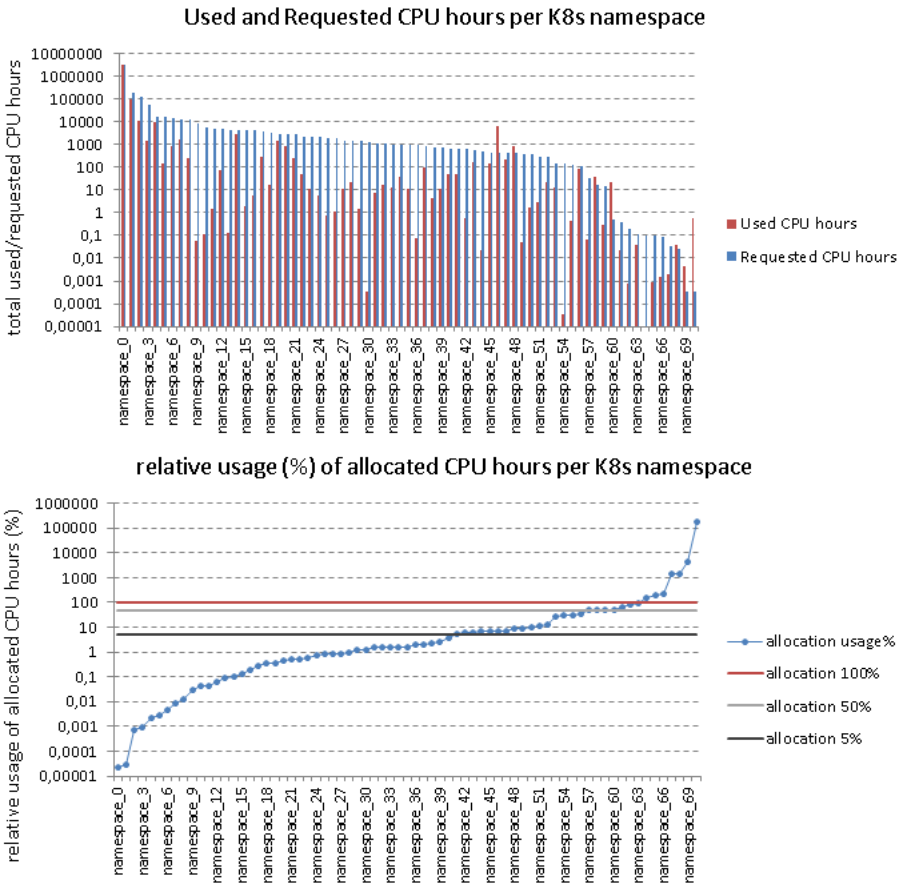

Fig. 2. The comparison of absolute (top) and relative (bottom) usage of requested CPU resources in CERIT-SC Kubernetes (K8s) cluster during the last 90 days. Y-axis in log. scale.

upper part of Figure 2 we show the total used CPU hours and the requested (allocated) CPU hours per K8s namespace. Clearly, there are huge differences both in the amount of used resources as well as in the allocations (Y-axis is in *log* scale). When we normalize these allocations into percents (see Figure 2 bottom), we can see how poorly those allocated resources are being used. 57.7% of namespaces uses less than 5% of allocated resources. The fact that some namespaces use (way) more than 100% of allocated resources is caused by the Kubernetes allowing Pods to have their CPU limits greater than their guaranteed allocations.

2.4 Problem Summary and Scheduling Objectives

To sum up the challenges that we want to address let us briefly recap the scheduling problem. In ideal scenario, we want to provide a service that will allow immediate start of users' workloads to guarantee interactive-like experience. At the same time, we need to provide this service for free while having a fixed budget (i.e., fixed size of infrastructure). Also, we want to minimize resource wastage. Since these requirements are somehow contradictory, we need to develop a reasonable compromise.

First of all, we need to ensure unused resources will not be wasted but rather used by some suitable (lower priority) workloads. Thankfully, this can be solved quite easily (see Section 4.2). The complicated part is how to guarantee quick start for new workloads when, e.g., the infrastructure is full. So far, we foresee several possible directions. The first possibility is to terminate some of those currently executing workloads or guarantee that (at least some) of those executing workload will complete in reasonable time. However, in order to solve this we need to address major problems. Simply put, we need to find a mechanism to select (and assign) priority to users' workloads which in turn will help us to decide which workloads to stop and/or allow to run.

The complication is that this priority mechanism can not be static in general. It is not sufficient to assign each user a fixed priority (or a static share) and keep it intact. This would (in a long run) lead to inefficiencies and/or unfairness. The reasons are obvious — just like in HPC, the "priority" of a given user (tenant) may change in time (for various reasons). Similarly, the size of a "share" that a given user can obtain depends, e.g., on the current number of active users and their resource usage.

In other words, Kubernetes is an orchestrator, that is very good in keeping the infrastructure in some "desired state". However, it does not provide automated mechanisms to *dynamically adjust* this "desired state" with respect to the changing situation in the system. Obviously, we need such component to address the aforementioned chalenges. In the following section we will discuss some of the tools that Kubernetes provides to enable complex scheduling policies. As we will demonstrate, it currently lacks the ability to fully fulfill our needs. Therefore, in Section 4 we will provide a proposal how to, at least partially, solve it.

3 Scheduling in Kubernetes

As we have already presented, we are searching for a solution that would allow us to fairly execute various types of workloads — batch, low latency interactive, bursty as well as long-running services. Due to the nature of the infrastructure (free of charge, limited resources) we will need to to develop a rather robust scheduling policies. In this section we will mention some of the key components of Kubernetes that may help us to achieve this goal.

Prevailing version (`1.22`) of container orchestrator Kubernetes offers several general concepts that can be utilized together to build more complex scheduling

policies. Many available features can usually be found in other scheduling systems as well, therefore we will not discuss them thoroughly but we will rather focus on their usefulness when dealing with our scheduling problem.

3.1 Pods and Jobs

In Kubernetes, the basic unit of scheduling is a Pod. It is the smallest deployable unit of computing and contains one or more containers. A Job in Kubernetes is a higher level of abstraction than a Pod. A Job creates one or more Pods and will (try) to execute these Pods until a specified number of them successfully terminate. The important feature of Job is that a *deadline* can be specified and Jobs can be cleaned up by CronJobs, i.e., deleted from the system after their completion. Therefore they can be used for HPC-like jobs with known maximum allowed runtime. Jobs are thus useful building blocks to prevent the "endless computing" scenario mentioned in Section 2.1.

3.2 Resource Requests and Limits

Kubernetes uses two types of resource allocation for each container — request and limit — that can be applied to CPU and memory. Request represents guaranteed resources that will be allocated to a container whereas limit is the upper bound of the resources. Standard Kubernetes scheduler makes resource allocation based on requests meaning the scheduler ensures that for both CPU and memory, the sum of their requests (respectively) of all containers scheduled on a node is less than its capacity. CPU limit is a hard upper bound on amount CPU time a container can use. Pod resource request/limit is the sum of the resource requests/limits of that type for each container in the Pod. However, containers share total CPU time and if all containers need more CPU time than their request but less than limit (i.e., several containers use more than they requested), performance degradation can be observed but container runtimes do not terminate jobs or containers for excessive CPU usage.[10]

On the other hand, memory limit imposes strict regime — if container exceeds the limit, the system kernel terminates the process that attempted the allocation and it is likely that the Pod will be evicted if memory shortage appears on the physical node. As scheduler allocates resources solely according to requests, it can happen that a node is short of resources if many containers exceed request resources. In such a case, container eviction starts and some Pods are terminated and moved to another node. However, there are no checkpoints and Pods are basically restarted.

Apparently, requests and limits can be used as a building block to accommodate bursty workloads with generally low momentary CPU utilization (by setting low requests and generous limits). The problem is that requests and limits cannot be modified for running Pods, neither can running Pods be migrated. To change existing limit or move it somewhere else a Pod must be restarted.

[10] https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/

It is worth mentioning that no other resources can be strictly limited (or requested) in Kubernetes scheduler, e.g. network bandwidth, GPU or I/O throughput. Technically, *nvidia add-on*[11] enables manipulation with GPU card in Kuberentes in same manner as CPU but eventually, there are multiple ways how GPU can be used in a container without formal request. To conclude, it is important to think about other computational resources and cover them in future discussions because there are many applications that might not be concerned about CPU time but rather stable filesystem connection.

3.3 Priority Classes

Priorities are extensively used in HPC world to indicate users' rights to use resources. Kubernetes offers similar concept called Priority Class[12]. Priority Class demonstrates the importance of a Pod among all other Pods in the cluster or in the pending queue. If a Pod cannot be scheduled due to limited capacity of a cluster, the scheduler attempts to preempt one or more Pods with lower priority in favor of scheduling pending Pods with higher priorities.

Priority classes can be configured as preempting and non-preempting. A workload assigned to non-preempting priority class will stay in the scheduling queue until its resource requests are satisfied. This represents a kind of *silent overtake* when prioritized workload claims resource ahead of others but does not violently terminate other workloads, with the risk of losing their in-progress computation.

Importantly, while a priority class can be changed, added or even removed, this will not impact already running Pods. In other words, the priority of running Pod can not be changed without restarting that Pod.

3.4 Labels, Affinity, Anti-affinity

Kubernetes understands heterogeneous clusters exist and they feature wide variety of node types. This creates many opportunities for fine scheduling when Pods need to run on same node (e.g. due to sharing cache) or oppositely, Pods run on different nodes in order to lower chance of node failure bringing down all workloads.

If there are circumstances when Pods pose their own preferences concerning nodes, Kubernetes has several ways[13] of employing affinity (or anti-affinity) mainly represented by assigning labels and taints to nodes and nodeSelectors to Pods. Importantly, node taints can be used to repel Pods from specific nodes. Moreover, taints can be used to evict Pods and both taints and labels can be changed dynamically. This features can greatly contribute not only to improve Pod's performance but it can be also used to steer the scheduler toward better decisions.

[11] https://kubernetes.io/docs/tasks/manage-gpus/scheduling-gpus/
[12] https://kubernetes.io/docs/concepts/scheduling-eviction/pod-priority-preemption/
[13] https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/

4 Problem Solutions using Kubernetes Building Blocks

This section discusses several possible approaches how to deal with the outlined problems shown in the previous sections. Not all of them are directly connected to scheduling, but they rather present different approaches to running workloads in the Kubernetes platform. Whenever possible, *we try to use legacy K8s functionality* instead of using either some third party solution or proposing new components.

4.1 Separate clusters

The first and by far the easiest solution to assigning resources to multiple competing workloads is creating separate clusters for specific computations (e.g. interactive jobs, HPC jobs, web services). Separate clusters bring the possibility of applying distinct schedulers into each cluster where one might be more suitable than the other for a certain workload type.

Nonetheless, this is merely a naive solution because it brings overhead for users as well as administrators. Users must be familiar with each cluster's structure in order to decide the most appropriate environment for workloads; they have to control progress at multiple places and eventually they spend more time analyzing infrastructure. Furthermore, administrators must handle several clusters, provide maintenance and continuous development. On the other hand, analyzing the most executed workload types brings the opportunity to tailor scheduler to specific requirements of the specific workload classes. Still, within a cluster diverse workloads may co-exist, thus bringing back most of the problems mentioned earlier.

4.2 Scavanger Jobs

One of the major problems (see Figure 2) observed in practice is the low CPU utilization of guaranteed allocations. One way how to increase resource utilization without impacting other users is to deploy so called *scavenger jobs*, i.e., jobs that are reasonably short/small and can be easily terminated and later resumed [9]. Scavenger jobs usually run at low priority and if resources are needed, they are terminated. When a resource becomes free again, scavanger jobs are resumed.

In Kubernetes, the administrator can define preemptible classes for Pods. These Pods then act as scavanger jobs; they get started and terminated according to resources' state. As they can be preempted by the scheduler at any time, eligible users will obtain interactive feedback immediately.

We have evaluated this approach using job preemption to deal with inefficiencies of using shared computational infrastructure. It turned out that preemptible scavenger jobs influence Pod allocations that rely on the interplay of requests and limits (see Section 3.2). These concepts are basically contradictory. Scavenger jobs, by their nature, do not leave available resources, so users are unable to utilize more resources than they requested, i.e., use the limit property. Therefore, in the following section we propose a solution for this problem in the form of ad-hoc *placeholder jobs* and we discuss their pros and cons.

4.3 Placeholder Jobs

As we stated above, there are no guarantees of free resources in the range between a Pod's request and its limit. As we observed, adopting scavanger jobs makes all resources above Pod's request almost unusable (as they are occupied). Moreover, in the current version of K8s it is not possible to change the amount of requested resources without container restart[14], so the user (or the scheduler) is unable to deal with this problem by temporarily increasing resource requests.

Until in-place vertical scaler is provided in K8s, there is another possibility to mitigate this problem. Instead of specifying resource requests and limits, we can use a little trick to ensure that there are enough free resources that can be used by user's Pod.

The trick how to obtain free resources (used by scavenger jobs) on a particular node is to create so called placeholder job, i.e., a job that *reserves resources but it does not consume them*. Placeholder job terminates existing scavanger job(s), thus freeing resources for the demanding Pod. Using node affinity, we can easily ensure that the placeholder job runs on the same node as the Pod that needs more resources. This approach works because there is no CPU or memory pinning to particular Pod, so the unused resources (reserved by placeholder) can be consumed by anyone.

Figure 3 illustrates this approach. On the left side we show an idle Pod. Its requested CPUs are low and its (unused) CPU limit is utilized by scavenger jobs. When the load increases (Figure 3 right), a placeholder job with large CPU request is deployed on that node (evicting low priority scavengers) and the the now busy Pod can use its CPUs to the limit.

This solution is not perfect as there are no guarantees that created free resources will be consumed just by the requesting Pod/user. Still, our initial evaluation shows that it is good enough in most situations. We believe that the combination of scavenger and placeholder jobs is currently good solution to keep the utilization high while allowing for quick vertical scaling of selected Pods.

4.4 Unresolved Issues

Native Kubernetes concepts alone are not capable of ensuring fair, efficient, and transparent automated scheduling mechanism. So far, we were able to use some of those building blocks to come up with solutions to several problems. Still, some problems (e.g., fairness) remain unresolved. Let us now discuss some of the directions that can be taken to deal with them.

Hybrid Cloud Approach The concept of hybrid cloud is now appearing everywhere. Nearly all big cloud providers have already tackled the idea and generally, it is considered as a promising way of computing. The definition of hybrid cloud is not entirely transparent because word "hybrid" itself induces some ambiguity

[14] https://github.com/kubernetes/kubernetes/pull/102884

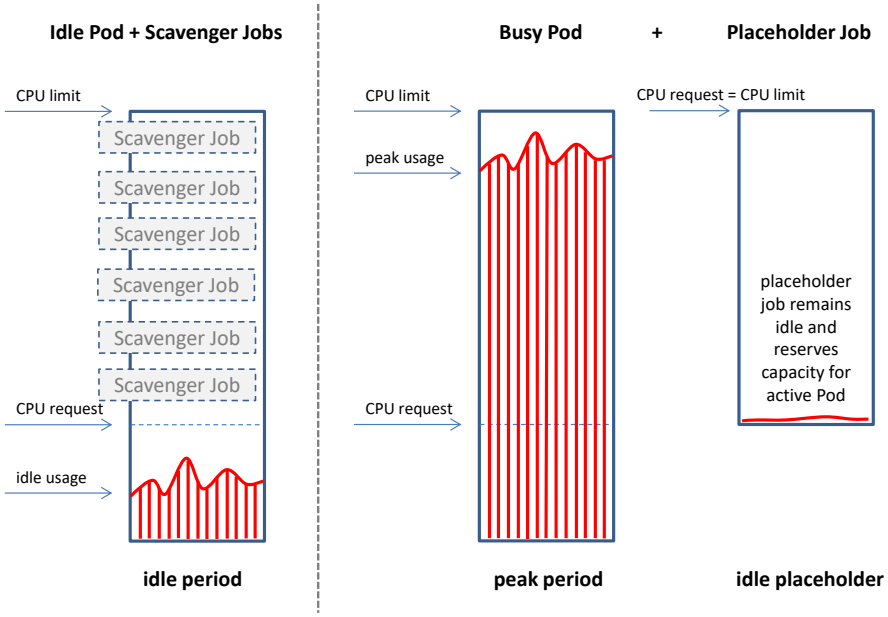

Fig. 3. During idle period scavenger jobs use free resources (left). When Pod's CPU load grows, placeholder job is started thus evicting scavenger jobs and freeing resources for active Pod (right).

and room for multiple explanations. However, RedHat has come with a list of actions[15] that hybrid cloud should be able to perform and one of them is to *"be able to move workloads between environments"*. We consider this claim as a perfect definition because joining environments — in our context HPC and container-based environments — could represent possibly viable solution to some of the unresolved scheduling problems we discussed. In the first sections of this article, we have introduced complications connected to moving workloads from HPC to container infrastructure, mainly related to resource requests overestimation and "access control" policies including fairness. We should give it a second thought and admit that for now, some computations perform better in traditional HPC environment, e.g., due to significant resource requests (hundreds of GB of RAM, hundreds of CPUs) or code-optimizations for certain hardware infrastructures. Usually, container-oriented clusters are not composed of large nodes and even if they are, there are limits on the number of containers that can be deployed in a cluster.

Hybrid cloud approach could bring an alternative way of computing where truly HPC workloads would be submitted to HPC environment and other workload types would continue existing as containers. Importantly, HPC schedulers offer better resource division and always allow to fine-tune fairness and job or-

[15] https://www.redhat.com/en/topics/cloud-computing/what-is-hybrid-cloud

dering, so offloading large workloads from Kubernetes clusters into HPC cluster would diminish the current need for major changes in K8s's scheduling abilities. Until Kubernetes community creates a way of dealing with HPC-cloud transformation, it would be sufficient to develop a sort of connector for HPC jobs that would connect the two worlds.

While this "outsourcing" may help us to survive for some time, we still need to properly handle truly container-based workloads. Especially, we need to focus on their life cycle which we will discuss next.

Pod Life-cycle and Resource Draining Once we have Pods running in our system, we must make sure that only "living" Pods will keep their allocations and all leftover workloads will be evicted and their Pods deleted. Here we propose a naive mechanism which uses priority classes, resource requests and limits. We understand the complexity of scheduling, resource allocation and fairness, and we are aware of the simplicity of the following method.

In the beginning, each workload submitted into the Kubernetes has the same priority and it must provide its specification of resource requirements. Once deployed, the system observes and logs user's utilization of resources for a specified period of time, e.g., three to four days. If the observed resource consumption is close to the requests, no action is needed. If the resource consumption is significantly lower than requested, this workload's priority class is decreased and the user is informed about this fact together with data and a set of recommendations on how to improve their resource requests. Whatever is the case, the final decision (action Y/N) is stored in a database for further comparisons. After the first *observation time period* the new period begins and same rules apply. Undoubtedly, priority class can not be lowered forever so a proper mechanism handling ignorant users/workloads must be in place. For example, after, e.g., five decreasing periods, the Pod is evicted and deleted. This simple approach guarantees that long-running idle workloads can be preempted or terminated in order to drain resources for new tasks.

Such simple eviction works out of the box only for workloads resembling long stateful services. It is common that cluster hosts burst Pods, on-demand computations or pipelines with fluctuating resource requests. Therefore, during one observation time period, they might present themselves as *resource-intensive* (fully utilizing requested resources) and the next time they can be *resource-dormant* (utilizing near-to-nothing). One solution to that could be enforcing singularity principle, thus letting *one Pod perform only one task* which naturally breaks all pipelines and compound computations into better manageable units. Assigning resource requests to small individual units is more accurate and should contribute to reducing resource wasting.

Accounting, Monitoring and Control These approaches will require the development of some monitoring framework, i.e., a "controller" with a predefined logic and some recent knowledge about the behavior of workloads and/or users in the system. Such solution is not currently available in K8s. Sure, this can be

done manually by the administrator, but it does not scale very well. Also, some non trivial accounting is necessary when goals like long-term fairness are to be achieved. Again, this requires non trivial data to be recorded, analyzed and used by some internal logic to develop proper scheduling decisions with the desired impact.

5 Real Workload Trace from CERIT-SC Installation

In this section we provide real data from our K8s installation in CERIT-SC system [1]. It is based on Kubernetes cluster version 1.21 consisting of 20 nodes. Each node is equipped with 128 hyperthreaded cores, 512 GB RAM, one NVIDIA GPU card and 7TB of local SSD storage. In total, the system has 2,560 CPU cores. All of those 20 nodes have worker roles, i.e., they are able to run any Pod. Default limit is 110 Pods per node but it has been increased to 160 Pods per node, so we are able to run up to 3200 Pods on the whole cluster, service jobs are included in this limit.

We collected data of all Pods that were executed on the cluster during the last 90 days (December 2021 – February 2022). The workload trace contains information about more than 60,000 Pods. In the workload trace, each Pod occupies one line that contains following data:

- Pod ID
- namespace ID
- arrival time
- start time
- termination time (either completed or killed)
- CPU request
- CPU limit
- RAM request
- RAM limit
- average, minimum and maximum CPU usage
- average, minimum and maximum RAM usage
- requested GPUs

Since real usage of CPU/Memory resources varies over time and it would not be practical to provide real usage, e.g., for each minute, we (currently) provide data on average, minimum, and maximum real CPU/Memory usage over the runtime of the Pod. Figure 4 shows current distribution of CPU requests and limits (top left) and also illustrates how the values of CPU utilization (avg., min., max.) are spread with respect to Pods' CPU requests. More details can be obtained from the time-series database that records these values periodically. We plan to provide this workload trace into the JSSPP workloads archive [8].

6 Related Work

Aforementioned problems have already been acknowledged by other groups. Even commercial world deals somehow with them as users can opt for enter-

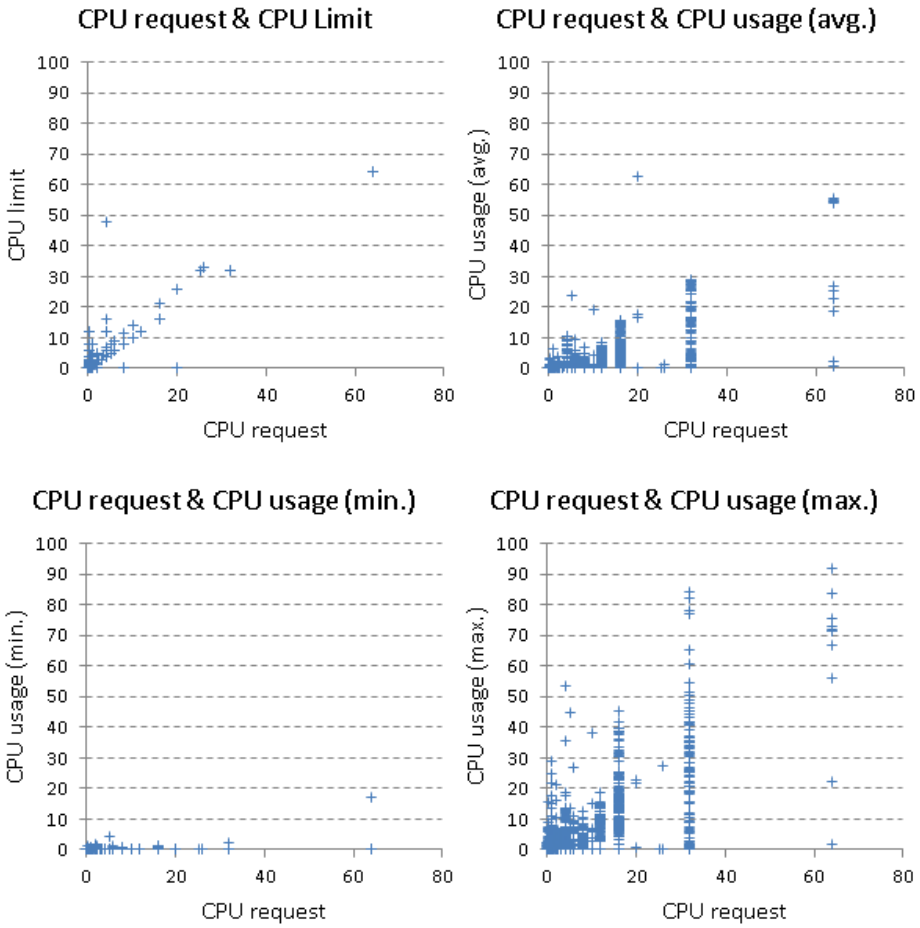

Fig. 4. CPU requests and limits (top left) and the average, minimum and maximum CPU utilization of Pods recorded in the trace. All charts have the same scale.

prise cloud infrastructures which often offer several free deals but in the context of large resource requests, unpaid plans do not provide sensible amount of resources. Enterprise infrastructure can be represented by Amazon AWS[16]. Amazon's paying model is based on per-second billing where user pays only for what he or she uses. Amazon currently offers five plans of reserving computational instances which all introduce various discounts and peculiarities. One of the plans, *Spot Instances*, is based on utilizing unused EC2 capacity in the cloud with significant discount. However, EC2 can reclaim the capacity anytime so when selecting this plan, user sets preferred way of handling evicted workload (hibernate, stop, terminate). In case capacity is needed by AWS services, user

[16] https://aws.amazon.com

receives notification two minutes before reclamation.[17] This approach shows resources are finely managed in enterprise infrastructures as well. However, users are still motivated to think about their requirements because they would have to pay more while infrastructure providers are maximizing earnings.

In the field of container orchestrators, solutions proposed to solve scheduling obstacles are usually crafted only for needs of subjects or as proof of concept without further integration into whole system. The reason is that modern technologies are widely used, open-source hence shaped by outsiders, individuals, enterprises and anyone interested which eliminates the utmost need for academic literature and deep research. Currently, real solutions or novel ideas introductions happen at community discussion forums or at source code management platforms such as *GitHub*[18]. As Randal points out[16]: *"... recent projects such as Docker and Kubernetes are largely written by outsiders providing external commentary rather than by the primary developers of the technologies. As a result, recent academic publications on containers tend to lack the depth of perspective and insight that was common to earlier publications on virtual machines, capabilities, and security in the Linux Kernel. The dialog driving innovation and improvements to the technology has not disappeared, but it has moved away from the academic literature and into other communication channels."*

From published works we choose four that discuss scheduling strategies, fairness resolution and analyze low cluster utilization. We observe that resource allocation issues have become recognized and researchers are trying to invent alternatives to basic Kubernetes scheduler. Still, problems discussed in the works remain unsolved as incorporating any change into the official system is a longer process.

In the work "Availability-driven scheduling in Kubernetes" Farias et al.[3] introduce different Kubernetes scheduling approach based on resource allocation according to promised quality of service (QoS). Their implementation and performed experiments show that QoS-driven scheduling yields better and more reliable service with fairer and more efficient resource division.

Medel et al. propose in their paper "Client-Side Scheduling Based on Application Characterization on Kubernetes"[14] idea that clients should provide a characterization of their applications which would allow scheduler to evaluate the best configuration to deal with the workload at a given moment. The enhanced scheduler design puts emphasis on balancing number of applications in each node and minimizing degradation caused by resource contention. Clients or developers are responsible for providing information about resources used intensively by their applications utilized by scheduler in advance. The solution achieved 20 percent improvement in a test case compared to basic Kubernetes scheduler but one can argue if user is capable of correctly assessing application's needs, especially in relation to HPC jobs where as mentioned, overestimation is ubiquitous.

[17] https://aws.amazon.com/ec2/spot/
[18] https://github.com

Apart from suggesting new scheduling strategies, researchers have noticed non-existence of fairness in Kuberentes[5]. Hamzeh et al. propose a model to calculate and assign resource limits fairly among the Pods in the Kubernetes environment. Authors state that due to early development stage no real case example scenarios could be presented but the work brings interesting view on cloud fair allocation algorithms (DRF[4], MLF-DRS[7] and FFMRA[6]).

Le and Liu in [12] open the work from different perspective where they discuss resource inefficiency of data centers with connection to global emissions and electric energy consumption. Overall, the paper focuses on improving cluster utilization without degrading quality of service. For that purpose, they developed an online resource manager that combines both load balancing and feedback control. Evaluations show that the tool achieved truly higher resource utilization compared to user submitting resource requests.

Ma and Wang [13] propose a new scheduler on top of standard K8s installaction called *Volcano*. It presents some interesting features such as support for efficient batch processing, however it is solely oriented on well-defined set of batch workloads, i.e., it is not a universal solution to our problems.

7 Conclusion

This paper discussed our experience with the Kubernetes container orchestrator and various problems related to the process of scheduling and resource allocation in a containerized environment. Based on existing Kubernetes concepts, we suggest several solutions to existing scheduling challenges such as infinite and interactive computing and/or overestimation of resource requests. Efficient scheduling and resource allocation can not be achieved with current Kubernetes tools easily. While Kubernetes is very good and robust in keeping the infrastructure in some "desired state", it does not provide automated mechanisms to *dynamically adjust* the "desired state" with respect to the changing situation in the system.

The transformation from HPC to containers will not happen in a few months, it is an ongoing, long-lasting process that should be performed by a stable provider to ensure a complete and reliable shift. Since such efforts are not strong enough now, we suggest that a hybrid cloud approach could serve as an interim solution of integrating and merging Kubernetes and HPC environments. Moreover, we offer several resource allocation concepts that might perform well in future versions of Kubernetes. Last but not least, we provide real-life workload trace from our installation to the scientific community.

8 Acknowledgments

Access to the CERIT-SC computing and storage facilities provided by the CERIT-SC Center, under the program "Projects of Large Research, Development, and Innovations Infrastructures" (CERIT Scientific Cloud LM2015085), is greatly

appreciated. We also acknowledge the support supplied by the project "e-Infrastruktura CZ" (e-INFRA LM2018140) provided within the program Projects of Large Research, Development and Innovations Infrastructures.

References

1. CERIT Scientific Cloud (February 2022), http://www.cerit-sc.cz
2. Chen, J., Cao, C., Zhang, Y., Ma, X., Zhou, H., Yang, C.: Improving cluster resource efficiency with oversubscription. In: 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC). vol. 01, pp. 144–153 (2018). https://doi.org/10.1109/COMPSAC.2018.00027
3. Farias, G., da Silva, V.B., Brasileiro, F., Lopes, R., Turull, D.: Availability-driven scheduling in kubernetes
4. Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., Stoica, I.: Dominant resource fairness: Fair allocation of multiple resource types. In: 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11) (2011)
5. Hamzeh, H., Meacham, S., Khan, K.: A new approach to calculate resource limits with fairness in kubernetes. In: 2019 First International Conference on Digital Data Processing (DDP). pp. 51–58 (2019). https://doi.org/10.1109/DDP.2019.00020
6. Hamzeh, H., Meacham, S., Khan, K., Phalp, K., Stefanidis, A.: Ffmra: A fully fair multi-resource allocation algorithm in cloud environments. In: 2019 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Internet of People and Smart City Innovation. pp. 279–286 (2019). https://doi.org/10.1109/SmartWorld-UIC-ATC-SCALCOM-IOP-SCI.2019.00091
7. Hamzeh, H., Meacham, S., Virginas, B., Khan, K., Phalp, K.: Mlf-drs: A multi-level fair resource allocation algorithm in heterogeneous cloud computing systems. In: 2019 IEEE 4th International Conference on Computer and Communication Systems (ICCCS). pp. 316–321 (2019). https://doi.org/10.1109/CCOMS.2019.8821774
8. JSSPP workloads archive (February 2022), https://jsspp.org/workload/
9. Kane, K., Dillaway, B.: Cyclotron: a secure, isolated, virtual cycle-scavenging grid in the enterprise. In: Proceedings of the 6th International Workshop on Middleware for Grid Computing. Association for Computing Machinery, Inc. (December 2008)
10. Klusáček, D., Parák, B.: Analysis of mixed workloads from shared cloud infrastructure. In: Klusáček, D., Cirne, W., Desai, N. (eds.) Job Scheduling Strategies for Parallel Processing. pp. 25–42. Springer International Publishing, Cham (2018)
11. Klusáček, D., Chlumský, V.: Planning and metaheuristic optimization in production job scheduler. In: Job Scheduling Strategies for Parallel Processing. LNCS, vol. 10353. Springer (2017)
12. Le, T.N., Liu, Z.: Flex: Closing the gaps between usage and allocation. In: Proceedings of the Eleventh ACM International Conference on Future Energy Systems. p. 404–405. e-Energy '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3396851.3403514, https://doi.org/10.1145/3396851.3403514
13. Ma, K., Wang, K.: Introducing Volcano : A Kubernetes native batch system for high performance workload. In: KubeCon Europe. CNCF (2019)
14. Medel, V., Tolón, C., Arronategui, U., Tolosana-Calasanz, R., Bañares, J., Rana, O.: Client-side scheduling based on application characterization on kubernetes. pp. 162–176 (10 2017). https://doi.org/10.1007/978-3-319-68066-8_13

15. Morris, A.: Choosing the right scheduler for hpc and ai workloads, https://www.hpcwire.com/solution_content/ibm/cross-industry/choosing-the-right-scheduler-for-hpc-and-ai-workloads/
16. Randal, A.: The ideal versus the real: Revisiting the history of virtual machines and containers. ACM Comput. Surv. **53**(1) (feb 2020). https://doi.org/10.1145/3365199, https://doi.org/10.1145/3365199
17. Tsafrir, D.: Using inaccurate estimates accurately. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) Job Scheduling Strategies for Parallel Processing, LNCS, vol. 6253, pp. 208–221. Springer Verlag (2010)