Hochschule **Rosenheim**
University of Applied Sciences

ML and Big Data scheduling in Docker cluster management systems

MaSe - Peter Kurfer

Hochschule **Rosenheim**
University of Applied Sciences

Agenda

- Motivation
- Classic job scheduling systems - Slurm
- Comparison Slurm vs. Kubernetes

Motivation

- Docker containers are omnipresent
- Are Docker containers also useful for ML and Big Data processing?
- Is it possible to use current Docker cluster management systems for job scheduling in multi-user environments?
- Is it possible to use one or multiple GPUs to increase the performance?
- Is it possible to run multiple jobs in parallel without them interfering with each other?

Classic job scheduling systems - Slurm

- Slurm is an abbreviation for "Simple Linux Utility for Resource Management"
- Slurm has three key features:
  1. Allocation of exclusive/non-exclusive access to ressources
  2. Providing a framework for starting, executing and monitoring tasks on a set of allocated nodes
  3. Managing pending jobs in queues until they can be executed
- Slurm is used on 60% of the TOP500 supercomputers

Features

- Modular design (supports plugins)
- Highly scalable
- Fair-share scheduling (with hierarchical bank accounts)
- Preemptive and gang scheduling
- Accounting
- Different operating systems can be booted for each job
- Scheduling for generic resources (e.g. GPUs)
- Resource limits for users/bank account

Hochschule Rosenheim
University of Applied Sciences

Comparison Slurm vs. Kubernetes

Feature	Slurm	Kubernetes
Highly scalable	👍	👍
Fair scheduling	👍	👍
Gang scheduling	👍	👍 with kube-arbitrator
Accounting	👍	👎
Different OS for each job	👍	👍
Scheduling of GPUs	👍	👍 still in alpha state
Scheduling of generic resources	👍	👎
Resource limits for users/bank account	👍	👍 Resource limits can be set for namespaces

Kubernetes jobs

- Jobs are natively supported by Kubernetes
- Jobs can be executed in a single *pod* or in multiple parallel *pods*
- A job **should always** define resource requests and limits
- A job is recognized as completed when the container exits with a exit code `0`

Simple sample job

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2
      restartPolicy: Never
  backoffLimit: 4
```

Job with specified resource allocation policy

```
apiVersion: batch/v1
kind: Job
metadata:
  name: john-job
spec:
  template:
    spec:
      containers:
        - name: john
          image: knsit/johntheripper:latest
          command: ["/bin/bash", "-c", "..."]
          resources:
            requests:
              cpu: 1000m
            limits:
              nvidia.com/gpu: 1
```

Parallel jobs in Kubernetes

Kubernetes offers two different kinds of parallel jobs:

1. jobs with fixed count of parallel workers
2. jobs based on a work queue

Jobs with fixed count of completions

Declare the job like this:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: parallel-job-1
spec:
  template:
    ...
  completions: 10
```

Jobs with fixed count of completions - considerations

- A job that declares a fixed count of completions has a default `spec.parallelism` value of `1` but that value can be increased
- If a pod fails it might be restarted depending on the values for the `backoffLimit` and the `restartPolicy`
- In a future release Kubernetes will pass the partition index (a value between `1` and `spec.completions`) to each pod to enable the pod to work only on his partition without the need for any external coordinator
- A higher number for `spec.parallelism` than `spec.completions` is ignored and will fallback to `spec.completions`

Jobs with a work queue

Declare the job like this:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: parallel-job-1
spec:
  template:
    ...
  parallelism: 10
```