Introduction to using the EIDF GPU cluster for members of EIDF029 and EIDF107

Ulrich Germann

September 2024

What is the Edinburgh International Data Facility (EIDF)?

- “EIDF is a collection of computational, data management and safe haven services supported by the Data Driven Innovation Programme of the Edinburgh and South-East Scotland City Region Deal.”

  https://edinburgh-international-data-facility.ed.ac.uk/about/in-a-nutshell
- run by the Edinburgh Parallel Computing Centre, which is part of the University of Edinburgh
- provides
  - computing resources
  - data management resources
- access is through “projects”
  - EIDF029: Informatics
  - EIDF107: GAIL (Genrative Artificial Intelligence Laboratory)
- the GPU cluster runs under Kubernetes

Getting access

EIDF029: research staff and PGR students in Informatics only
EIDF107: members of the Generative AI Laboratory only

- Create a SAFE account with your UEDIN email address here: `https://safe.epcc.ed.ac.uk/` if you don't have one already.
- Through the SAFE portal, request access to project EIDF029 or EIDF107. You will be notified by email once approved.
- Once approved, set up ssh keys and MFA on the SAFE Portal (Go to 'Login Accounts' in the navigation bar on the top.)

Logging in

Access to the cluster is through ssh via a gateway *Jump Host*

```
ssh -i $SSH_KEY \
    -J $USER@eidf-gateway.epcc.ed.ac.uk \
    $USER@$PROJECT_HOST_IP
```

You can make your life easier by adding this to your ˜ /.ssh/config

```
Host $EIDF_PROJECT
   User $USER
   IdentityFile $SSH_KEY
   HostName $PROJECT_HOST
   ProxyJump $USER@eidf-gateway.epcc.ed.ac.uk
   IgnoreUnknown Usekeychain
   UseKeychain yes
```

What is Kubernetes?

- developed at Google to manage Software-as-a-Service at scale
- focus is on keeping services live and available
- dynamic scaling
- unlike *Slurm*, **NOT** per se an batch queueing system (conceptually, resources are assumed to be always available)
- resources are separated by *Name Spaces*
- no notion of individual ownership of Kubernetes artefacts by user
- has Docker containerisation technology at its heart

Docker

- Docker *Images* typically contain all but only the software to run a particular programme or application (the Docker command or entry point)
- Docker Images are hosted in a Docker *Repository*, or on the local host
- When you run an Image, the *Docker Daemon* lets you specify certain parameters, set environment variables, forward ports, and mount directories into the Docker *Container* that instantiates the image.
- Philosopy: one Image per service

Docker Containers are not Virtual Machines!!!

Example of a Docker run

```
docker run --name web -p 8080:80 --rm \
    -v /path/to/document/root:/usr/share/nginx/html nginx
```

The Docker Daemon

- pulls the `nginx` Image from the remote Repository if necessary
- names the Container 'web'
- maps the port 80 inside the Container to port 8080 on the Docker *Host*
- mounts `/path/to/document/root` from the host into the Container as `/usr/share/nginx/html`
- removes the Container when it is finished / stopped (`--rm`)

All changes to the Container are lost when the container is removed!!!

Kubernetes Pods: container orchestration

- Remember the Docker philosophy: one Image/Container per service
- Containers are *orchestrated* to form more complex web services, e.g.,
  - Container1: mysql database
  - Container2: web frontend
  - Container3: authentication service
- Kubernetes Pods orchestrate containers

Our first Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: $USER-hello-world1
  labels:
    eidf/user: $USER
spec:
  containers:
    - name: ubuntu
      image: ubuntu:20.04
      command: ["/bin/bash", "-c", "echo ’Hello World!’"]
      resources:
        limits:
          nvidia.com/gpu: 0
          cpu: 1
          memory: 4Gi
  restartPolicy: Never
```

Persistent Volume Claims (PVCs)

- Claim storage provided by Kubernetes
- Read-write
- Currently each PVC can be mounted into only one Pod at a time
- PVCs can only be accessed when mounted into a Pod

Creating PVCs

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
   name: $PVCNAME
   labels:
      eidf/user: $USER
spec:
   accessModes:
   - ReadWriteOnce
   resources:
      requests:
         storage: $STORAGE
   storageClassName: csi-rbd-sc
```

Mounting PVCs

- add the following to .spec in your Pod specification

```
volumes:
  - name: workspace
    persistentVolumeClaim:
      claimName: $PVCNAME
```

- add the following to the respective container specification in .spec.containers:

```
volumeMounts:
  - name: workspace
    mountPath: /workspace
```

Mounting NFS directories (1)

- any accessible NFS server can be mounted
- currently EIDF029 has its own NFS server that
  - is accessible on the login node (RW)
  - can be mounted into Pods (RO)
  - currently has a quota of 50GB per user
  - provides a mirror of commonly used datasets and models
- EIDF107 currently has none, but the EIDF029 NFS server is mountable from EIDF107
- Apologies to EIDF107-only users! An NFS server is in the works ...

Mounting NFS directories (2)

- Declare the volume. In `.spec.volumes`, add, e.g.,

```
- name: publicdata
  nfs:
    server: $EIDF029_NFS_SERVER_IP
    path: /public
- name: userdata
  nfs:
    server: $EIDF029_NFS_SERVER_IP
    path: /user/$USER
```

- Mounting the declared volume works just like mounting PVC volumes. In the respective `volumeMounts` section, add, for example,

```
  - name: publicdata
    mountPath: /publicdata
  - name: userdata
    mountPath: /mydata
```

Interactive Pods/Jobs: Basic Rules

Rule #1 Don't! Unless you absolutely have to.

Rule #2 Request only what you really need (memory, CPUs, GPUS).

Rule #3 GPUs may only be requested through Jobs, not Pods.

Rule #4 Don't let interactive Pods/Jobs sit idly. Check frequently if they are up and running, and delete them when you are done (only your own, of course).

Rule #5 Directly submitted Pods currently bypass the queuing and quota allocation system. If they ever get in the way of overall operations, they will be terminated immediately without warning.

Rule #6 As long as you request laptop-scale resources (1-4 CPUs, 16GB RAM or less, and no GPUs, your Pod has a good chance of being tolerated, but without guarantees.

Interactive Pods/Jobs: How-to

- At least one Container within the Pod must be running forever.
- The most straightforward way to implement this is to specify something like this in `.spec.containers[0].command`:

  ```
  ["bash", "-c", "trap TERM; sleep infinity& wait; exit 0"]
  ```

  in the Pod specification.
- “Log in” to an interactive pod via

  ```
  kubectl exec -it <podname> -- bash
  ```

Kubernetes Secrets

- Kubernetes secrets aren't at all secrets within the Name Space. Everyone can see them and use them.
- They are still a good way of separating confidential information from code that you may want to distribute freely.

Creating Kubernetes Secrets

In principle (but won't work on EIDF029/EIDF107 directly!!!)

- specify key-value pairs on the command line

  ```
  kubectl create secret generic $SECRETNAME
  --from-literal=$KEYWORD1=$VALUE1
  --from-literal=$KEYWORD2=$VALUE2
  ```

- get value from a file, e.g.

  ```
  kubectl create secret generic $SECRETNAME
  --from-file=id_rsa=$HOME/.ssh/id_rsa
  --from-file=id_rsa.pub=$HOME/.ssh/id_rsa.pub
  ```

- create from an 'env'-file

  ```
  kubectl create secrets generic $SECRETNAME
  --from-env-file=./secrets.env
  ```

Creating Kubernetes Secrets (2)

```
kubectl create secret generic $SECRETNAME ...
--dry-run=client -ojson | jq ’.metadata.labels |= {
"eidf/user" :  env.USER }’
```

Using Kubernetes Secrets

- Declare secret as volume in `.spec.volumes`:

```
- name: $SECRETNAME
  secret:
    secretName: $SECRETNAME
```

- Mount secret in `.spec.containers[*].volumeMounts`

```
- name: $SECRETNAME
  mountPath: /secrets/$SECRETNAME
```

- Then access as files from the Pod.

From Pods to Jobs

- currently, only Jobs can be queued
- Pods bypass the queue
- directly scheduled Pods will be deleted without warning

From Pods to Jobs

```
apiVersion: batch/v1
kind: Job
metadata:
  generateName: $USER-job-
  labels:
    eidf/user: $USER
    kueue.x-k8s.io/queue-name: $INFK8S_QUEUE_NAME
spec:
  backoffLimit: 0
  ttlSecondsAfterFinished: 300
  template:
    metadata:
      labels:
        eidf/user: $USER
    spec:
      restartPolicy: Never
      containers:
      - name: ubuntu
        image: ubuntu:20.04
        command: ["/bin/bash", "-./run.sh"]
        resources:
          limits:
```

Requesting GPUs

- MUST be done in a Job
- specify the number of GPUs requested in `.spec.template.spec.resources.limits.nvidia\.com/gpu`
- adjust the RAM requested in `.spec.template.spec.resources.limits.memory` (e.g., VRAM + 20Gi)
- specify the type of GPU in `.spec.template.spec.nodeSelector.nvidiac̣om/gpup̣roduct`
  - NVIDIA-A100-SXM4-40GB-MIG-3g.20gb
  - NVIDIA-A100-SXM4-40GB
  - NVIDIA-A100-SXM4-80GB
  - NVIDIA-H100-80GB-HBM3

Customizing your Container

- Quick and dirty: do it at the beginning of your run script
- "The proper way": create your own custom image
- In between: use a python virtual environment in a PVC

User Guide

https://git.ecdf.ed.ac.uk/infk8s/getting-started-on-the-eidf-gpu-cluster