Formation LRI,
Corentin Tallec & Diviyan Kalainathan

Plan

1. Introduction
2. Présentation générale de slurm
3. Comment envoyer un job sur le cluster
4. Monitorer ses jobs
5. Gestion des ressources GPU
6. Gestion de ses données/environnement de travail
7. Bonnes/Mauvaises pratiques
8. Management d'expérience à grande échelle
9. Docker/Rkt: Gestion avancée d'environnements

Introduction

Contexte:

- Ressources de calcul **partagées** e.g. **GPUs**
- Ressources de calcul **importantes** e.g. **beaucoup de GPUs**
- Besoin de **gestion de tâches**
  - Éviter les **erreurs d’utilisation des ressources**: “Oups, OutOfMemoryException sur ton job d’une semaine, j’espère que tu fais des checkpoints…”
  - Faciliter le lancement d’**expés à grande échelle**

Introduction (2)

Intérêts d’un ordonnanceur (job scheduler):

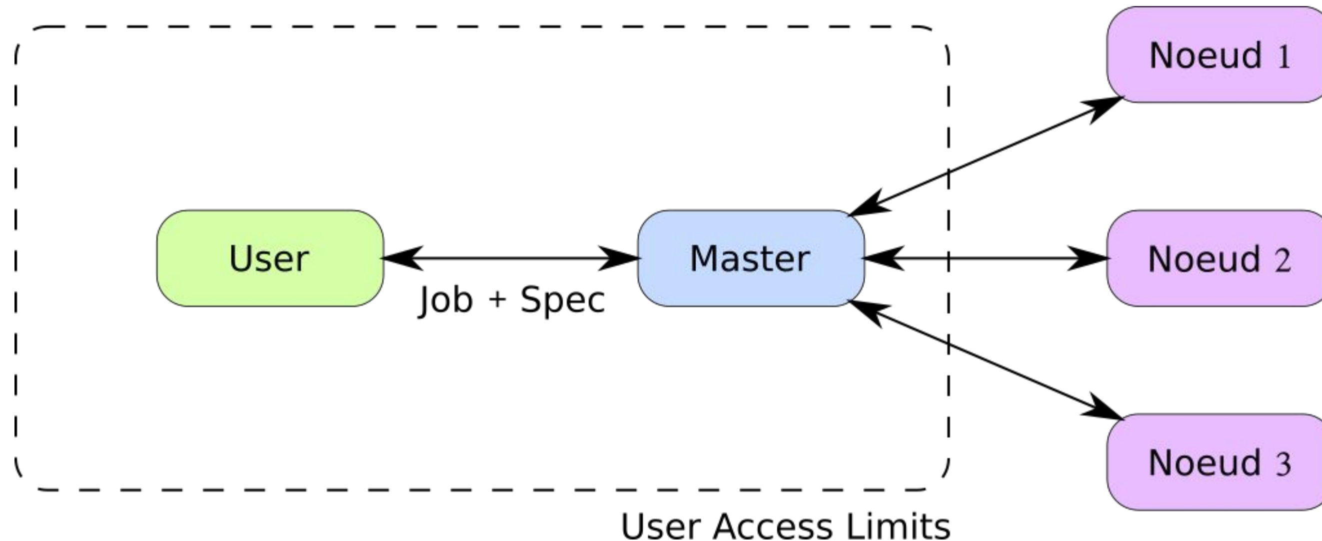
- Gestion **automatique** et optimale des ressources:
  - si SLURM m’alloue un GPU, **il est à moi**, personne d’autre (sauf comportement malveillant) ne pourra s’en servir
- **Système de queue** pour des tâches:
  - 10 jobs ou 100 jobs, même combat, quitte à patienter pour les ressources.
- Système de **partitions** pour gérer les **priorités**:
  - Typiquement **n GPUs réservés par utilisateurs**, ininterruptibles. Au delà, allocation en fonction de l’utilisation de l’utilisateur + possibilité d'interruption.

Pourquoi SLURM ?

1) Léger
2) Modulaire
3) Populaire
4) Scalable
5) Et surtout, relativement facile à installer

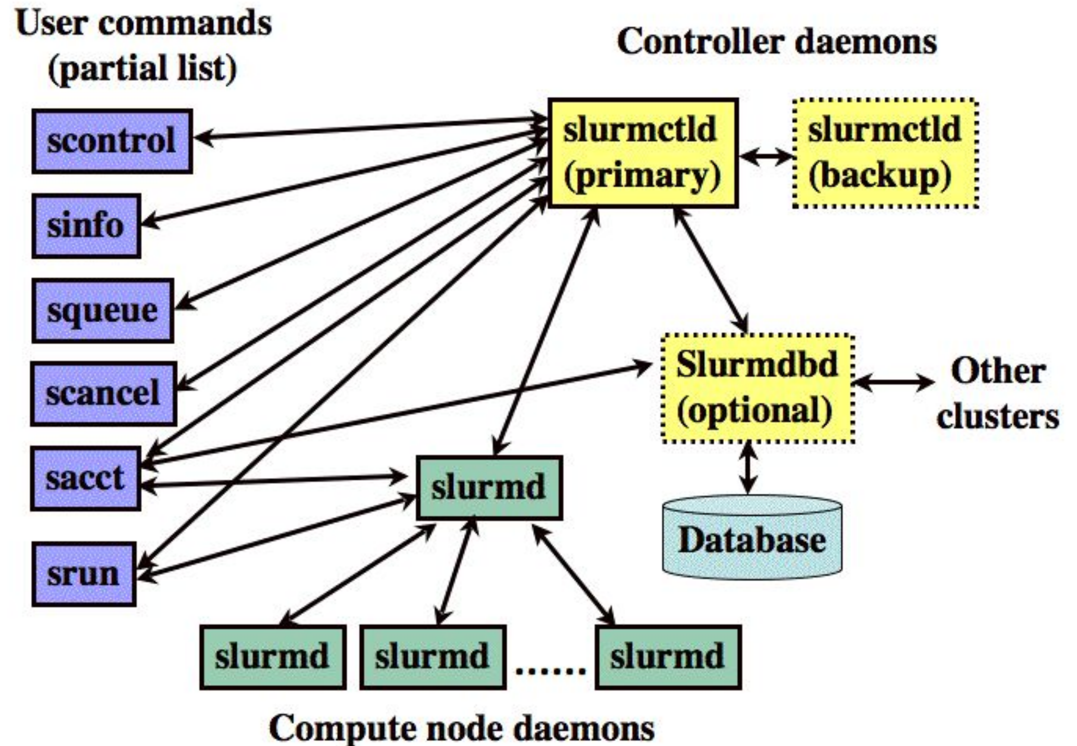
Sunway TaihuLight, 3e calculateur mondial en Juin 2018
(273 millions $ US, 93 PFlops)

Fonctionnement de SLURM: User side



Fonctionnement de SLURM: Admin side

- **slurmctld**: daemon de contrôle de slurm sur le **noeud master**
- **slurmd**: daemon slurm sur les **noeuds de calcul**



En pratique ?

- Les utilisateurs **lancent des jobs** sur le cluster. Un **job** est typiquement un **calcul** plus ou moins long et demandant plus ou moins de **ressources de calcul**. Exemple simple de job: **lancer l'entraînement d'un modèle de ML sur un set d'hyperparamètres.**
- A chaque **job**, l'utilisateur associe un ensemble de **ressources requises**, typiquement nombre de **GPUs**, nombre de **CPUs**, nombre de **tâches** (différent du nombre de CPUs).
- Le **batch scheduler** (ici SLURM), gère **l'allocation des ressources**, et donc le **scheduling** des jobs: un job n'est lancé que si **les ressources qu'il requiert sont disponibles**.

Envoyer un job sur le cluster

- **1ère étape:** Se connecter sur la frontale du cluster (e.g. **titanic-master** pour le cluster **titanic**). Souvent, cette connexion se fait en plusieurs sauts: penser à **paramétrer des ProxyCommand** dans votre **configuration ssh** pour vous faciliter la vie !

Exemple typique:

```
Host saclay
    User ctallec
    Hostname ssh.saclay.inria.fr

Host titanic
    User ctallec
    Hostname titanic
    ForwardAgent yes
    ProxyCommand ssh saclay nc %h %p
```

Envoyer un job sur le cluster

Deux types de jobs:

a) Jobs interactifs (**srun**)

Pour le **debug** & **tâches légères**

b) Jobs non-interactifs (**sbatch**)

Pour les expériences plus larges (majorité des jobs)

Job Interactif (srun)

Lancer un job interactif (i.e. une session de terminal) sur le cluster:

```
srun --time=1:00:00 --gres=gpu:1 --pty bash
```

Lance un job interactif, pour une durée de 1 jour, en réservant 1 GPU, et va utiliser bash comme shell.

A l'execution, un **JOB_ID** unique est associé au job lancé. Il va servir à **monitorer l'état du job**, et si échec il y a, il permet de récupérer les raisons de cet échec.

Envoyer un job sur le cluster

Énormément d’options slurm disponible pour **spécifier des ressources**:

1. **--time**: spécifie la **limite de durée du job**
2. **--gres**: spécifie un **type de ressources** auxiliaires (typiquement GPUs) à allouer
3. **--ntasks**: nombre de tâches à lancer (voire plus loin)
4. **-C**: spécifie des **features** en particulier (voire plus loin)
5. **--pty**: spécifie le **shell** à utiliser (zsh, sh) pour les jobs interactifs

La doc est exhaustive, mais les infos sont parfois dur à trouver: https://slurm.schedmd.com/sbatch.html

Envoyer un job via un script SBATCH

- Script SLURM: script **bash classique** + **directives SBATCH.**
- Les directives SBATCH servent à préciser les ressources à utiliser.
- Exemple typique:

```
#SBATCH --mem=64g
#SBATCH --nodes=1
#SBATCH --partition=besteffort
#SBATCH --cpus-per-task=8
#SBATCH --gres=gpu:1
#SBATCH --output=local_logs/pendulum_inv_test/2019_01_30_17_07_14/%j.stdout
#SBATCH --error=local_logs/pendulum_inv_test/2019_01_30_17_07_14/%j.stderr
#SBATCH --job-name=pendulum_inv_test
#SBATCH --exclude=baltic-1
#SBATCH --open-mode=append
#SBATCH --signal=B:USR1@120

mkdir local_logs/pendulum_inv_test/2019_01_30_17_07_14/$SLURM_JOB_ID

LOG_STDOUT="local_logs/pendulum_inv_test/2019_01_30_17_07_14/$SLURM_JOB_ID.stdout"
LOG_STDERR="local_logs/pendulum_inv_test/2019_01_30_17_07_14/$SLURM_JOB_ID.stderr"

function restart
{
        echo "Calling restart" >> $LOG_STDOUT
        scontrol requeue $SLURM_JOB_ID
        echo "Scheduled job for restart" >> $LOG_STDOUT
}

function ignore
{
        echo "Ignored SIGTERM" >> $LOG_STDOUT
}
trap restart USR1
trap ignore TERM

# Start (or restart) experiment
date >> $LOG_STDOUT
which python >> $LOG_STDOUT
echo "---Beginning program---" >> $LOG_STDOUT
echo "Exp name     : pendulum_inv_test" >> $LOG_STDOUT
echo "Slurm Job ID : $SLURM_JOB_ID" >> $LOG_STDOUT
echo "SBATCH script: /tmp/00b4aadb-6cea-41f6-8b80-a9c5a4e1390b.sh" >> $LOG_STDOUT

python $HOME/dockers/dock.py -c "xvfb-run -s '-screen 0 1400x900x24' python main.py --steps_btw_train 10 --env_id pendulum --noise_type action --batch_size 256 --hidden_size 256 --nb_tru
e_epochs 0.5 --gamma 0.7 --nb_steps 10 --sigma_eval 0 --sigma 1.5 --theta 7.5 --nb_train_env 128 --nb_eval_env 64 --memory_size 1000000 --learn_per_step 50  --lr 0.001 --time_limit 10.0
--weight_decay 0 --optimizer rmsprop --algo discrete_advantage --steps_btw_catchup 10 --tau 0 --eval_gap 0.01 --dt 0.001 --nb_layers 1 --logdir local_logs/pendulum_inv_test/2019_01_30_17
_07_14/$SLURM_JOB_ID/"

wait $!
```

Pas à pas dans un script slurm

```
#SBATCH --mem=64g
#SBATCH --nodes=1
#SBATCH --partition=besteffort
#SBATCH --cpus-per-task=8
#SBATCH --gres=gpu:1
```

Allocations de ressources:

- 64g de RAM
- 1 noeud de calcul
- 8 CPUs (nous sommes sur une seul tâche)
- 1 GPU
- Sur partition besteffort (sur titanic, la partition besteffort **autorise la préemption**)

Un aparté sur la préemption

- La plupart des clusters SLURM possède plusieurs **partitions** (option -p de SLURM).
- Une partition est le plus souvent associée à une **qualité de service**.
- Le facteur le plus important de la **qualité de service** est la possibilité de **préemption**: la préemption désigne la possibilité pour le job d’être interrompu par un autre **job** ayant une **priorité** plus élevée.

Préemption: le cas titanic

- Sur **titanic**, deux partitions sont disponibles pour la plupart de utilisateurs:
  - La partition **default**, elle **n'autorise pas la préemption**. Si je lance un job sur la partition par défaut, il ne peut pas être interrompu. Cela signifie notamment que je n'ai pas à me soucier de relancer mon job s'il est suspendu. En contrepartie, chaque utilisateur ne peut utiliser au maximum que **4 GPUs** sur cette partition.
  - La partition **besteffort**. Sur cette partition, **les jobs peuvent être préempté.** Si un job est lancé sur la partition **default** et qu'il ne peut être lancé qu'en suspendant mon job, mon job sera stoppé, et le nouveau job sera lancé.
- **Le point important: si un job peut être préempté, c'est à l'utilisateur de faire en sorte que le job soit relancé en cas de préemption, et qu'il recharge les précédents résultats obtenus !**

- Lancer des jobs **Préemptibles** suppose la mise en place **par l'utilisateur** d'un système de **checkpointing**.
- Une méthode simple pour **checkpointer**: associer à chaque **job** un dossier de log uniquement associé au job. Placer dans ce fichier de log les **résultats du job** ainsi que **toutes les informations nécessaires à la relance du job** (par exemple, pour des modèles de machine learning, les paramètres courant/les meilleurs paramètres du modèle).

Redirection des sorties standards

```
#SBATCH --output=local_logs/pendulum_inv_test/2019_01_30_17_07_14/%j.stdout
#SBATCH --error=local_logs/pendulum_inv_test/2019_01_30_17_07_14/%j.stderr
#SBATCH --job-name=pendulum_inv_test
#SBATCH --exclude=baltic-1
#SBATCH --open-mode=append
```

- **--output** et **--error** gère la redirection des stream de sortie et d'erreur du job. Le **%j** présent dans le chemin de redirection permet de récupérer le **JOB_ID** du job qui va s'exécuter. Si le **JOB_ID** du job courant est 342, le chemin correspondant au stdout sera

  local_logs/pendulum_inv_test/2019_01_30_17_07_14/342.stdout

- **--job-name** correspond au nom que l'on assigne au job. Il permet de monitorer le job plus facilement.
- **--open-mode=append** signifie que la redirection de l'output et de l'error se font en mode append, si les fichiers existent déjà, ils ne seront pas réécrits, mais les nouveaux logs seront ajoutés à la suite des anciens (surtout utile en cas de préemption et de relance).
- **--exclude=baltic-1** exclut l'un des noeuds de calcul.

Lancement du script

```
mkdir local_logs/pendulum_inv_test/2019_01_30_17_07_14/$SLURM_JOB_ID

LOG_STDOUT="local_logs/pendulum_inv_test/2019_01_30_17_07_14/$SLURM_JOB_ID.stdout"
LOG_STDERR="local_logs/pendulum_inv_test/2019_01_30_17_07_14/$SLURM_JOB_ID.stderr"
```

```
# Start (or restart) experiment
date >> $LOG_STDOUT
which python >> $LOG_STDOUT
echo "---Beginning program---" >> $LOG_STDOUT
echo "Exp name     : pendulum_inv_test" >> $LOG_STDOUT
echo "Slurm Job ID : $SLURM_JOB_ID" >> $LOG_STDOUT
echo "SBATCH script: /tmp/00b4aadb-6cea-41f6-8b80-a9c5a4e1390b.sh" >> $LOG_STDOUT

python $HOME/dockers/dock.py -c "xvfb-run -s '-screen 0 1400x900x24' python main.py --steps_btw_train 10 --env_id pendulum
e_epochs 0.5 --gamma 0.7 --nb_steps 10 --sigma_eval 0 --sigma 1.5 --theta 7.5 --nb_train_env 128 --nb_eval_env 64 --memory_
--weight_decay 0 --optimizer rmsprop --algo discrete_advantage --steps_btw_catchup 10 --tau 0 --eval_gap 0.01 --dt 0.001 --
_07_14/$SLURM_JOB_ID/"

wait $!
```

Gérer la relance

```
#SBATCH --signal=B:USR1@120
```

```
function restart
{
        echo "Calling restart" >> $LOG_STDOUT
        scontrol requeue $SLURM_JOB_ID
        echo "Scheduled job for restart" >> $LOG_STDOUT
}

function ignore
{
        echo "Ignored SIGTERM" >> $LOG_STDOUT
}
trap restart USR1
trap ignore TERM
```

- On signale la terminaison avec **USR1** plutôt que **SIGTERM**, pour pouvoir disposer de plus de temps pour terminer le job avant un **SIGKILL**.
- On gère la réception des signaux: en cas de **USR1** (préemption ou durée expirée), on relance avec **scontrol requeue {jobid}**
- On ignore sigterm, pour disposer de plus de temps.

Monitoring du cluster

- Tous les utilisateurs ont accès à des outils pour monitorer **l'état général du cluster**, à savoir le **nombre de noeuds**, **les noms des noeuds**, **l'état des noeuds, les jobs lancés par chaque utilisateurs** …
- Les commandes à retenir:
  - **squeue:** liste les jobs lancés sur le cluster. Notamment utile pour vérifier que ces propres jobs tournent toujours, ou sont en attente: **squeue | grep {usr_id}**
  - **sinfo:** permet de monitorer l'état des noeuds et des partitions. Les noeuds peuvent être dans différents états, soit fonctionnels, **idle** ou **mixed**, soit dysfonctionnel, **drain, draining, down**. Permet de comprendre pourquoi un job ne se lance pas forcément.

Monitoring des jobs

1. **Premier réflexe**: via les logs du job: (cf les options --output et --error de sbatch)

   Les logs peuvent être **non informatifs si l'erreur vient du script slurm lui même**. Dans ce cas:

2. Via Slurm: **scontrol** show job {jobid}. Donne notamment des informations sur les raisons de l'échec du lancement d'un job.

Gestion des GPUs

La demande de GPU s’effectue avec l’option

`--gres=gpu:1` pour 1 gpu

Sur la plupart des clusters, on peut demander une architecture en particulier avec l’option:

`-C kepler`

Attention : Cette commande ne change que la valeur de la variable $CUDA_VISIBLE_DEVICES, gérée automatiquement par les frameworks de DeepLearning. Variable à ne pas modifier.

Gestion des données

- Les différents clusters ont des politiques différentes concernant le stockage, à la fois des **résultats d’expérience** et des **datasets**.
- Idéalement, **ne pas retélécharger de datasets déjà présent sur le cluster**, et se renseigner sur comment y accéder.
- Faire attention à ne pas faire exploser la quantité de logs, et à les placer aux endroits appropriés.

- En l’absence de **Docker**, utiliser **miniconda/venv** pour les **environnements python**:

  https://conda.io/en/latest/miniconda.html

- Chaque utilisateur doit installer miniconda lui même. Miniconda permet à chaque utilisateur d’avoir plusieurs environnements python qui lui sont propre, à l’intérieur desquels il peut installer **toutes les bibliothèques** dont il a besoin.
- Pour tout autre installation, contacter les administrateurs du cluster.

Mauvaises pratiques

- Accéder aux noeuds de calculs **sans passer par SLURM**: normalement impossible.
- **Ignorer/Modifier** la variable d'environnement **CUDA_VISIBLE_DEVICES**: la gestion des GPUs de SLURM n'est **qu'informative**. Pour assurer le bon fonctionnement du **cluster**, chaque utilisateur doit se limiter aux ressources GPUs qui lui sont alloués.
- **Logger** de manière non parcimonieuse. En ML, les données et les modèles peuvent peser lourd. Chaque utilisateur doit avoir une utilisation **responsable des ressources de stockage du cluster**.
- Effectuer des opérations **intensives sur la frontale**. Tous les utilisateurs du cluster ont accès à la frontale, c'est un élément crucial du bon fonctionnement du cluster. Il faut éviter de la ralentir en s'en servant comme noeud de calcul.
- Éviter d'encombrer **des noeuds GPUs** avec de grandes quantité de **jobs CPUs...**

Quelques notes annexes

- Vérifier que les scripts **utilisent bien les GPU et pas les CPU** des noeuds !
- **Expérimenter incrémentalement**:
    - Débugger sur les partitions par défaut, tester ses scripts sur des petits échantillons.
    - Rendre ses scripts résilients à la préemption, automatiser leur lancement et leur relance.
- Checkpoint **régulièrement** et **intelligemment** ses résultats.

Demain:

- Job arrays.
- Gestion de Docker, écriture de Dockerfiles.
- Introduction au Multi-GPUs/Multi-tasks avec MPI.

Job arrays

- Lancer plus d’une centaine de jobs avec sbatch peut représenter une charge importante pour master
- Une bonne pratique est d’utiliser les jobs arrays

https://slurm.schedmd.com/job_array.html

- Le job array exécutera le même script, avec des variables d’environnement $SLURM_ARRAY.. différentes

```bash
#!/bin/bash
#SBATCH --gres=gpu:1
#SBATCH --requeue
#SBATCH -p besteffort
#SBATCH -x baltic-1
#SBATCH --time=1-00:00:00
#SBATCH --no-kill
#SBATCH --array=0-196

args=()
for metric in "mse" "mmd"
do
  for gnh in 5 10 20 50 100 500 2000
  do
    for anh in 5 10 20 50 100 500 2000
    do
     for l1 in 0.05 0.01
     do
         args+=("--metric ${metric} --anh ${anh} --gnh ${gnh} --l1 ${l1}")
     done
    done
  done
done

python cc.py ${args[${SLURM_ARRAY_TASK_ID}]}
```

Exercice: Lancer des expés à grande échelle

Pour pratiquer **Slurm**:

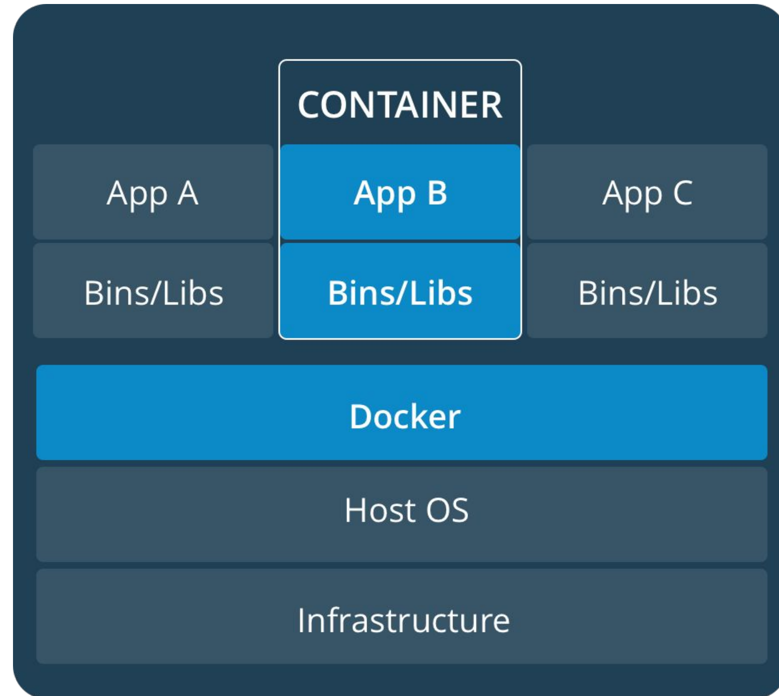
1. Lancer une expé simple en **sbatch.**
2. Encapsuler le lancement de l’expé dans un **script python.**
3. Gérer la **relance automatique** en **cas de préemption**.
4. **Paramétrer les contraintes slurm** à partir de python.
5. Mettre en place les **bonnes pratiques** en matière de gestion de log.

Docker

- A virtual environment manager
- Much more independent from the system than Anaconda
- A universal platform that uses containers (images) to store their environment
- "If it works on 1 device using the docker, it should run without issue on any device using docker"
- A space efficient git-like management of images.
- Multi-platform (Linux, Windows, Mac)

Docker (2)

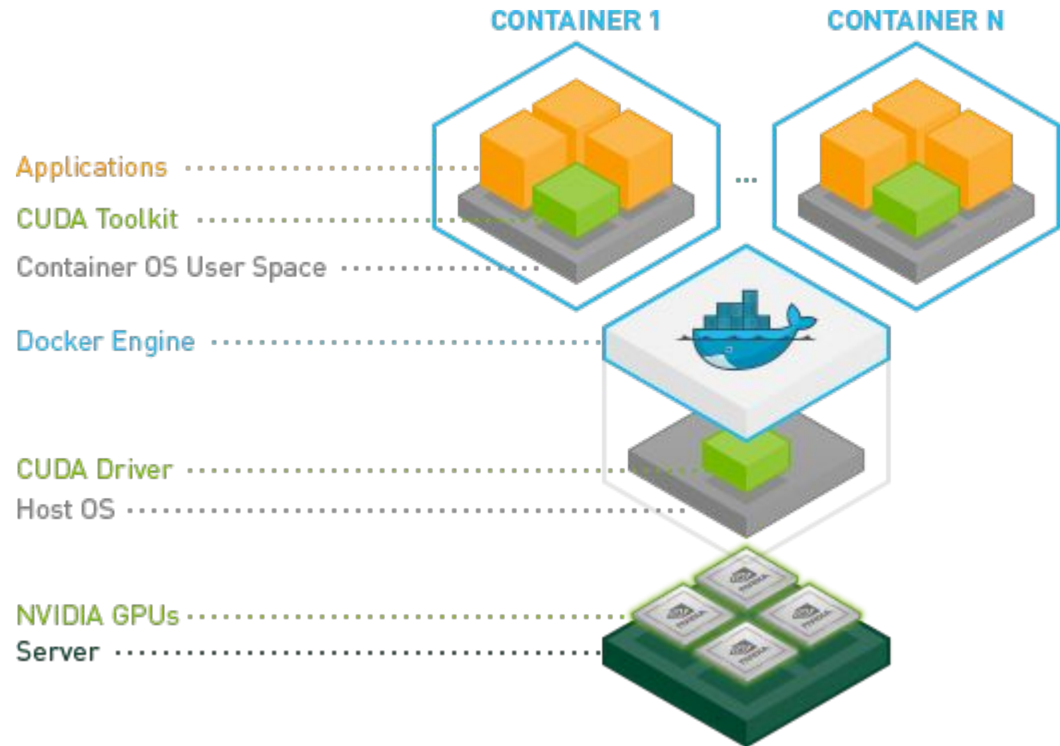
Gestionnaire d'environnement efficace



Pourquoi Docker?

- We got more and more users with their own needs in libs/apps.
- All those libs are becoming harder and harder to maintain
- Users can make their own tailored container
- We can use NVIDIA's NGC dockers, for a more well-maintained ecosystem between CUDA, CUDNN, and the python libs.

Nvidia-docker



Options importantes dans l’exécution

--init : Slurm pourra kill les docker et les process

-u=$(id -u):$(id -g) : Les fichiers crée par les utilisateurs pourront être lus et modifiés par ces derniers.

--rm: supprimer le container à la fin de l’exécution pour sauver un peu d’espace

NV_GPU=$CUDA_VISIBLE_DEVICES : (devant la commande docker) Pour n’allouer à l’utilisateur que les GPUs attribués par SLURM

-it : Mode interactif ; si le job est lancé en srun --pty bash.

Options simplifiées

La commande **sdocker** est mise à disposition et comporte toutes les options nécessaires au bon fonctionnement.

```
NV_GPU=$CUDA_VISIBLE_DEVICES exec nvidia-docker run --ipc=host --rm \
-u=$(id -u):$(id -g) --init
```

Exercice: Lancer un container docker !

Créer son image Docker avec un DockerFile

```
FROM divkal/nvidia-tensorflow:1.5
MAINTAINER diviyan <diviyan@diviyan-hp1040>

RUN pip install pandas scikit-learn numpy scipy \
       joblib jupyter pytest seaborn matplotlib cython scikit-image \
     Pillow numba cma hdbscan tensorflow-probability \
     pyvarinf argload tqdm h5py
# RUN pip install git+https://github.com/uber/pyro.git
RUN rm -rf /root/.cache/pip/*
CMD /bin/bash
```

Puis build et push son image:

```
#!/bin/bash

docker build  -t divkal/nvidia-tensorflow:1.6 .
docker push divkal/nvidia-tensorflow:1.6
```

Exercice: Scripts sbatch + Docker

Jobs Multi-noeuds

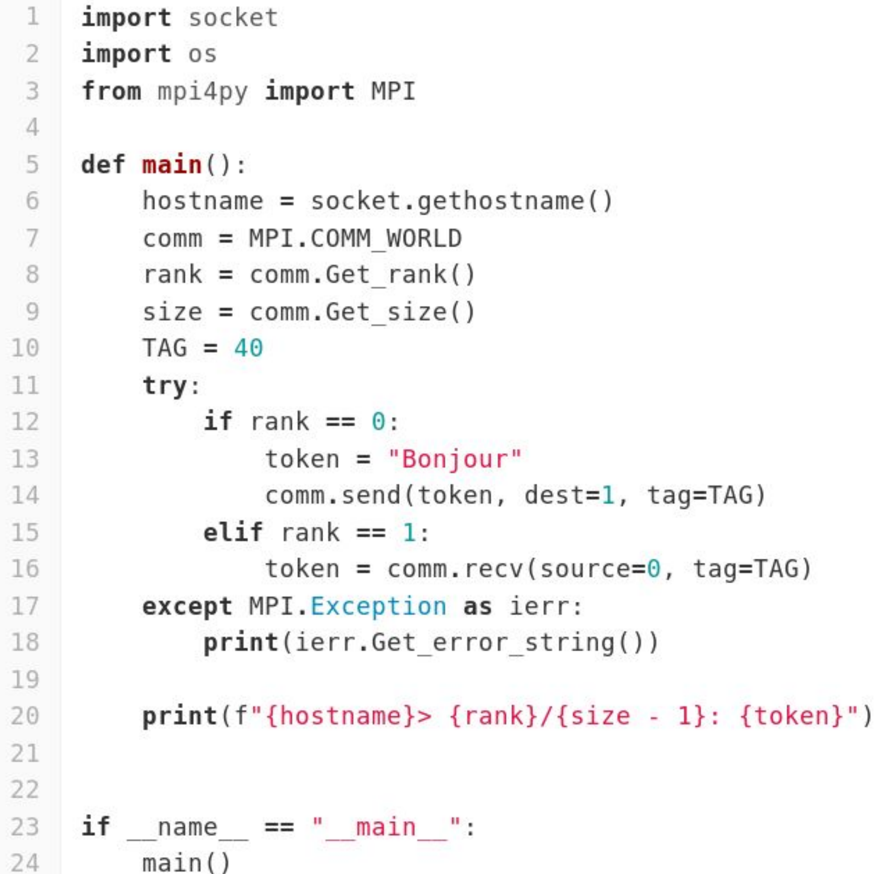
- Deux cas de figures:
    - **Pas de transfert de donnée:** facile, diviser le jobs **multi-noeuds** en plusieurs petits jobs.
    - **Transfert de donnée:** difficile.
- Pour les jobs **avec transfert de donnée**, utiliser de préférences les options de relativement **haut niveau**, souvent fournies de manière standard par les bibliothèques de deep.
- Si les options de haut niveau ne sont pas assez flexibles, **revenir sur MPI (Message Passing Interface)**, ou les équivalents dans le framework considéré.

MPI: Principe

- On lance **N tâches distinctes**, potentiellement sur des machines **séparées**.
- A chaque **tâche** est associé un **rang**. On peut **différencier le traitement** entre les différentes **tâches** en fonction du **rang.**
- Les différentes **tâches** peuvent communiquer via des mécanisme de communication, soit **point-à-point**, soit collectifs.

MPI et Slurm, un exemple basique

- Nous allons utiliser, dans une premier temps, **mpi4py**, un package python qui wrap une implémentation de MPI en python.
- Pour cela, installer d'abord **miniconda** dans votre homedir, créer un nouvel environnement virtuel, et installer **mpi4py** dans cet environnement.



```python
import socket
import os
from mpi4py import MPI

def main():
    hostname = socket.gethostname()
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()
    TAG = 40
    try:
        if rank == 0:
            token = "Bonjour"
            comm.send(token, dest=1, tag=TAG)
        elif rank == 1:
            token = comm.recv(source=0, tag=TAG)
    except MPI.Exception as ierr:
        print(ierr.Get_error_string())

    print(f"{hostname}> {rank}/{size - 1}: {token}")


if __name__ == "__main__":
    main()
```

MPI et SLURM: script SBATCH

```
#!/bin/bash
#SBATCH -N 2

source activate mpi
mpirun --mca btl_tcp_if_include eno1 python script.py
```

- **SBATCH -N 2**: on réserve **deux noeuds** ! Pour réserver deux **tâches: -n 2.**
- On active **notre environnement conda** où est installé mpi4py.
- On utilise **mpirun** pour run nos jobs. **Sans cela, un seul job sera lancé !**
- Parfois certaines interfaces réseaux seront invalide, on précise l’interface en utilisant **--mca btl_tcp_if_incude eno1**

Multi-GPUs + Multi-noeuds en PyTorch

- **Très proche de MPI**. Différence majeure: **Il faut setter à la main les paramètres de rang, de nombre de jobs et les moyens de communications sur le réseau, à partir des paramètres SLURM.**
- Nous allons le faire en détail ici. Pour plus d'info, ne pas hésiter à aller voir:
  - https://pytorch.org/tutorials/intermediate/dist_tuto.html
  - https://github.com/facebookresearch/XLM

Jobs distribués en PyTorch

Initialiser torch.distributed avec les variables SLURM:

- **Rank et size:**
  - rank = os.environ[“SLURM_PROCID”]
  - size = os.environ[“SLURM_NTASKS”]
- **Master port:**
  - Pick one.
- **Master address:**
  - os.environ[“SLURM_JOB_NODELIST”] donne la **liste des noeuds** => facile, on prend le premier !
  - Mais donne la liste des noeuds au format SLURM … i.e, si les noeuds alloués sont republic-1 et 2, on a republic-[1-2] …
  - 2 solutions: traitement de string pénible ou utiliser **scontrol show hostnames {nodelist}** qui renvoie la liste des noeuds séparés par des \n cette fois !

Initialiser torch.distributed

- On va utiliser **la backend distribuée nccl** et initialiser via des **variables d'environnement.**
- On a récupéré **l'hostname du noeud maitre, sélectionné son port, déterminé le rang du processus courant et le nombre de processus..**
- On change les **valeurs des variables d'environnement nécessaires à l'initialisation**:

```
os.environ['MASTER_ADDR'] = master_addr
os.environ['MASTER_PORT'] = str(master_port)
os.environ['WORLD_SIZE'] = str(size)
os.environ['RANK'] = str(rank)
```

- On **initialise**

```
print("Initializing PyTorch distributed ...")
torch.distributed.init_process_group(
    init_method='env://',
    backend='nccl',
)
```

Utiliser torch.distributed

- Une fois initialisé, on peut utiliser l'**API de communication** de torch.distributed.
- Cette **API** est un sous ensemble de celle de **MPI**. **Attention, à la compatibilité des opérations et des backends !!** Notamment, **send et recv ne sont pas toujours disponible** … Avant utilisation, regarder attentivement

https://pytorch.org/docs/stable/distributed.html

- **Remarque importante:** pour pouvoir recevoir un tenseur, il faut que ce tenseur ait été alloué dans le processus, **avec la bonne taille !**

Lancer le job distribué avec SLURM

- Même principe que pour MPI.
- **Quelques différences majeures**:
  - on n’utilise **pas MPI,** mais **slurm directement**. Il faut donc préciser **srun** plutôt que mpirun dans le **fichier SBATCH.**
- Comme pour MPI on peut avoir des problèmes d’**interfaces réseau invalides**, pour les résoudre, spécifier l’**interface réseau à la main via NCCL_SOCKET_IFNAME={interface_reseau}.**

```
#!/bin/bash
#SBATCH -N 2
#SBATCH --ntasks-per-node 4
#SBATCH --gres=gpu:4
#SBATCH -p besteffort
#SBATCH --output=logs/%j.stdout
#SBATCH --error=logs/%j.stderr

source activate torch
NCCL_SOCKET_IFNAME=eno1 srun python main.py --logdir logs/exp
```

Fin de la formation: Aux exercices !