

MASTER
EN CALCUL
HAUTE PERFORMANCE,
SIMULATION



Université
Perpignan
Via Domitia
CRÉATRICE D'AVENIRS DEPUIS 1350

UNIVERSITÉ DE PERPIGNAN

Stage de fin d'année Cloud HPC : Orchestration d'un cluster de Raspberry Pi

Auteur :
Anzid Zakaria

Sous l'encadrement de :
Mr. Gerald Becker

Table des matières

1	Introduction	1
1.1	Contexte et Motivations	1
1.1.1	Exemples en France	2
1.1.2	Exemples à l'échelle mondiale	2
1.1.3	Pourquoi et quand choisir un cluster de Raspberry Pi?	2
1.2	Objectifs	3
2	Technologies et outils utilisées dans ce Projet	4
2.1	Introduction aux Raspberry Pi	4
2.1.1	SSH Secure Shell	5
2.1.2	Docker et la Conteneurisation	5
2.1.2.1	Concept et définition	5
2.1.2.2	Avantages	5
2.1.2.3	Processus de création d'images Docker pour encapsuler des applications : définition des Dockerfiles	6
2.1.3	Docker Swarm pour l'orchestration des noeuds docker et la Gestion de Cluster	6
2.1.4	Kubernetes	7
2.1.4.1	Avantages de Kubernetes	7
2.1.5	Slurm, Gestionnaire de Tâches	7
3	Configuration	8
3.1	Configuration des raspberry Pis	8
3.1.1	Préparation des raspberry et Installation des systèmes d'exploitation	8
3.1.2	Association avec un réseau wifi	9
3.1.3	Activation du service ssh pour qu'on puisse nous connecter à distance à chacune de nos Raspberry Pis	9
3.1.4	Attribution d'une adresse ip statique pour nos raspberry pi	9
3.2	Configuration de Docker	10
3.2.1	Installation de Docker sur les Raspberry Pis	10
3.2.1.1	Mise à jour du Système	10
3.2.1.2	Installation de Docker	10
3.2.1.3	Ajout de l'utilisateur au Groupe Docker	10
3.2.1.4	Vérification de l'Installation	10
3.2.1.5	Démarrage Automatique de Docker	11
3.3	Configuration de Docker Swarm	11
3.3.1	Initialisation de swarm	11
3.3.1.1	Sélection d'un Nœud Manager	11
3.3.1.2	Rejoindre des Nœuds Workers au Swarm	11
3.3.1.3	Vérification du Cluster Swarm	12

3.4	Configuration de SLURM (Simple Linux Utility for Resource Management) . . .	13
3.4.1	Installation et Configuration de Munge	13
3.4.1.1	Installation de Munge	13
3.4.1.2	Génération et distribution de la clé Munge	13
3.4.1.3	Démarrage du service Munge	13
3.4.2	Installation de Slurm	14
3.4.2.1	Configuration de slurm.conf	14
3.4.2.2	Distribution de slurm.conf à l'aide de la commande scp	15
3.4.2.3	Création des répertoires Slurm	15
3.4.2.4	Démarrage des services Slurm	15
3.4.2.5	Vérification de la Configuration	16
3.4.2.6	Surveillance des journaux de SLURM	16
3.5	Configuration de Kubernetes	17
3.5.1	Prerequis	17
3.5.2	Set up du Noeud Maitre	18
3.5.2.1	Installation de k3s	18
3.5.2.2	Verification du status de k3s	18
3.5.2.3	recuperation du jeton pour faire joindre les noeuds travailleurs .	18
3.5.3	Set up des Noeuds esclaves	18
3.5.3.1	Verification du status de k3s sur les noeuds esclaves	19
3.5.4	Verification de letat du cluster	19
4	Déploiement	20
4.1	Déploiement d'une application avec docker et kubernetes comme outil d'orchestration	20
4.1.1	Compilation de l'application	20
4.1.2	Creation du Dockerfile	20
4.1.3	Creation de l'image docker	20
4.1.4	Upload au Docker HUB	21
4.1.4.1	Tag	21
4.1.4.2	Push	21
4.1.5	Creation du fichier .yaml	21
4.1.6	Deploiement	22
4.1.7	Vérification du deploiement	22
4.1.8	Journal/log	22
4.2	Déploiement d'une application avec Docker, Docker-swarm	22
4.2.1	Théoriquement	22
4.2.2	Pratiquement	23
4.2.2.1	Création du Dockerfile	23
4.2.3	Construction de l'image Docker	23
4.2.4	uploading de l'image au Docker Hub	23
4.2.5	Déploiement, creation d'un service	24
4.2.6	Creation du fichier docker-compose.yml	24
4.2.7	Déploiement avec docker compose .yaml	24
	Annexes	26
	Annexe 1	26
1	Code source	26

Chapitre 1

Introduction

1.1 Contexte et Motivations

Les clusters de calcul haute performance (HPC) sont des systèmes spécialisés conçus pour résoudre des problèmes complexes nécessitant une puissance de calcul substantielle. Ces clusters sont composés de nombreux ordinateurs interconnectés, travaillant en parallèle pour exécuter des calculs et des simulations, ce qui permet un traitement plus rapide et plus efficace de vastes quantités de données. Ce processus d'interconnexion et de travail en parallèle améliore non seulement la vitesse, mais aussi la capacité à gérer des tâches computationnelles intensives.

Les clusters HPC fonctionnent en divisant un gros problème de calcul en parties plus petites et plus faciles à gérer, réparties entre les nœuds du cluster. Chaque nœud exécute la tâche qui lui est attribuée et combine les résultats pour produire la sortie finale. Ce processus, appelé calcul parallèle, est essentiel au bon fonctionnement des clusters HPC. Les clusters HPC utilisent un planificateur de tâches, pour s'assurer que les charges de travail de calcul sont réparties de manière uniforme sur l'ensemble du cluster. Le planificateur de tâches gère l'allocation des ressources de calcul, ce qui garantit que chaque nœud fonctionne à sa capacité maximale et évite les goulots d'étranglement liés au traitement.

Les clusters HPC ont de nombreuses applications, parmi lesquelles :

1. **Recherche scientifique** : Ils sont fréquemment utilisés pour simuler des systèmes complexes, comme le comportement des matériaux, les modèles météorologiques et la dynamique des fluides.
2. **Ingénierie** : Ils permettent de simuler le comportement des structures et des systèmes, notamment les composants des avions et des automobiles.
3. **Analyse financière** : Dans le domaine de la finance, les clusters HPC analysent de vastes quantités de données, telles que les tendances boursières, pour identifier des modèles et réaliser des prévisions.
4. **Apprentissage automatique** : Les clusters HPC sont de plus en plus utilisés pour entraîner des réseaux de neurones profonds (deep learning), qui nécessitent une puissance de calcul significative.

Les clusters HPC (High Performance Computing) sont des systèmes informatiques de haute performance qui combinent plusieurs ordinateurs puissants pour offrir une capacité de calcul exceptionnelle. Ils sont devenus des outils indispensables pour les avancées dans de nombreux domaines scientifiques et industriels, permettant de traiter et d'analyser de vastes quantités de données à une vitesse et une efficacité sans équivalent.

1.1.1 Exemples en France

En France, on peut citer notamment :

- **GENCI (Grand Équipement National de Calcul Intensif)** : GENCI gère les ressources de calcul intensif en France, englobant plusieurs supercalculateurs répartis dans diverses institutions comme le CEA (Commissariat à l'énergie atomique et aux énergies alternatives) et le CNRS (Centre National de la Recherche Scientifique).
- **Jean Zay** : Ce supercalculateur, situé à l'IDRIS (Institut du Développement et des Ressources en Informatique Scientifique), est parmi les plus puissants en France. Il est utilisé pour la recherche en intelligence artificielle, en physique, en climatologie, et dans de nombreux autres domaines.

1.1.2 Exemples à l'échelle mondiale

À l'échelle mondiale, on trouve des clusters HPC encore plus puissants, comme :

- **Fugaku** : Ce supercalculateur, basé au Japon, est actuellement l'un des plus puissants au monde. Il est utilisé dans des domaines variés tels que la médecine et la modélisation climatique.
- **Summit** : Situé au Oak Ridge National Laboratory aux États-Unis, Summit est un supercalculateur de pointe utilisé pour des recherches en biologie, en astrophysique, et en intelligence artificielle.

Ces clusters HPC illustrent les dernières avancées technologiques en calcul intensif, permettant des progrès significatifs dans divers secteurs grâce à leur capacité à traiter et analyser de grandes quantités de données de manière extrêmement rapide et efficace.

1.1.3 Pourquoi et quand choisir un cluster de Raspberry Pi ?

Les Raspberry Pi se présentent comme une alternative attrayante et économique aux clusters HPC traditionnels. Ces petits ordinateurs à faible coût, initialement destinés à l'éducation et aux projets de bricolage, se sont révélés étonnamment puissants et polyvalents. Leur faible consommation d'énergie, leur coût ****abordable**** et leur facilité d'interconnexion en font des candidats idéaux pour la création de clusters HPC accessibles.

Il est important de souligner que, bien que les Raspberry Pi offrent une solution économique, ils ne sont pas aussi puissants que les clusters HPC traditionnels composés de serveurs et de stations de travail haut de gamme. Les processeurs ARM des Raspberry Pi, bien qu'efficaces pour leur taille et leur coût, ne peuvent rivaliser en termes de puissance brute de calcul avec les processeurs utilisés dans les environnements HPC professionnels. Cependant, leur capacité à être regroupés en grand nombre permet d'effectuer des tâches de calcul distribué à une échelle plus modeste.

L'objectif principal de l'utilisation des Raspberry Pi pour construire un cluster HPC n'est donc pas de remplacer les infrastructures HPC traditionnelles, mais plutôt de fournir une plateforme abordable et éducative. Cette plateforme permet aux étudiants, aux chercheurs et aux passionnés de technologie d'apprendre les principes fondamentaux de la mise en place, de la gestion et de l'optimisation des clusters de calcul.

L'adoption croissante des technologies de conteneurisation comme Docker et des systèmes d'orchestration comme Kubernetes a aussi ouvert de nouvelles voies pour la gestion efficace des ressources informatiques. Les conteneurs Docker encapsulent les applications, facilitant leur déploiement et leur gestion tout en assurant leur portabilité et leur isolation. Chaque conteneur peut fonctionner de manière autonome avec ses propres dépendances et configurations, simplifiant grandement la gestion des applications complexes sur le cluster.

Kubernetes est un système qui automatise la gestion des conteneurs, facilitant leur déploiement, leur mise à l'échelle et leurs opérations. Il inclut des fonctions avancées comme l'équilibrage de charge, la gestion des ressources et la résilience aux pannes, ce qui renforce la fiabilité et l'efficacité du cluster. Avec Kubernetes, les charges de travail sont mieux réparties pour une utilisation optimale des ressources disponibles.

En combinant ces technologies avec des gestionnaires de tâches comme Slurm, il est possible de maximiser l'efficacité et les performances d'un cluster de Raspberry Pi. Slurm, largement utilisé dans les environnements HPC traditionnels, planifie et gère les tâches de calcul en répartissant d'une manière intelligente les travaux entre les différents nœuds du cluster. Il assure une gestion efficace des files d'attente et des priorités, ce qui optimise l'utilisation du cluster.

1.2 Objectifs

L'objectif principal de ce projet est de démontrer la faisabilité et l'efficacité de la création d'un cluster HPC en utilisant des Raspberry Pis. Plus spécifiquement, ce projet vise à :

- Configurer et interconnecter plusieurs Raspberry Pis afin de former un cluster de calcul haute performance.
- Utiliser Docker pour encapsuler les applications, garantissant ainsi leur portabilité et leur isolation.
- Utiliser docker-swarm pour définir un nœud maître, et les nœuds travailleurs.
- Mettre en œuvre Kubernetes pour orchestrer les conteneurs, permettant une gestion optimisée des ressources et des charges de travail.
- Intégrer Slurm en tant que gestionnaire de tâches pour planifier et distribuer les calculs au sein du cluster.

Chapitre 2

Technologies et outils utilisées dans ce Projet

2.1 Introduction aux Raspberry Pi

Nous utilisons quatre Raspberry Pi 4 Model B Rev 1.4 dans notre environnement. Chaque unité est équipée d'un processeur ARM Cortex-A72 quad-core, chaque cœur fonctionnant à une vitesse de 108.00 BogoMIPS. Les fonctionnalités du processeur comprennent le support pour fp (unité de calcul en virgule flottante), asimd (instructions SIMD avancées)

```
pi@rpil:~$ cat /proc/cpuinfo
processor       : 0
BogoMIPS      : 108.00
Features      : fp asimd evtstrm crc32 cpuid
CPU implementer : 0x41
CPU architecture: 8
CPU variant   : 0x0
CPU part      : 0xd08
CPU revision   : 3

processor       : 1
BogoMIPS      : 108.00
Features      : fp asimd evtstrm crc32 cpuid
CPU implementer : 0x41
CPU architecture: 8
CPU variant   : 0x0
CPU part      : 0xd08
CPU revision   : 3

processor       : 2
BogoMIPS      : 108.00
Features      : fp asimd evtstrm crc32 cpuid
CPU implementer : 0x41
CPU architecture: 8
CPU variant   : 0x0
CPU part      : 0xd08
CPU revision   : 3

processor       : 3
BogoMIPS      : 108.00
Features      : fp asimd evtstrm crc32 cpuid
CPU implementer : 0x41
CPU architecture: 8
CPU variant   : 0x0
CPU part      : 0xd08
CPU revision   : 3

Revision      : d03114
Serial        : 10000000703f30e9
Model         : Raspberry Pi 4 Model B Rev 1.4
pi@rpil:~$ |
```

2.1.1 SSH Secure Shell

Pour la gestion à distance et l'administration de nos Raspberry Pi, nous utilisons SSH (Secure Shell). SSH est un protocole sécurisé qui nous permet de nous connecter de manière sécurisée aux systèmes Linux embarqués comme nos Raspberry Pi, afin d'exécuter des commandes, transférer des fichiers et configurer les appareils sans avoir besoin d'une interface utilisateur directe.

Cela facilite la maintenance et le contrôle de nos dispositifs à partir d'une seule interface, améliorant ainsi l'efficacité opérationnelle de notre projet.

2.1.2 Docker et la Conteneurisation

Conteneurisation et ses avantages par rapport à la virtualisation traditionnelle(virtual machine)

2.1.2.1 Concept et définition

La conteneurisation est une forme de virtualisation du système d'exploitation où les applications sont exécutées dans des espaces utilisateurs isolés appelés conteneurs, utilisant le même système d'exploitation partagé. Un conteneur d'application est un environnement informatique totalement empaqueté et portable.

Un conteneur contient tout ce dont une application a besoin pour fonctionner, y compris ses fichiers binaires, ses bibliothèques, ses dépendances et ses fichiers de configuration, le tout encapsulé et isolé dans un conteneur.

Isolation : La conteneurisation permet d'isoler l'application du système d'exploitation hôte, avec un accès limité aux ressources(on n'utilise pas l'OS entièrement), similaire à une machine virtuelle légère.

Portabilité : L'application conteneurisée peut être exécutée sur différents types d'infrastructure, tels qu'un serveur bare metal, dans le cloud ou sur des machines virtuelles, sans nécessiter de modifications pour chaque environnement.

2.1.2.2 Avantages

Efficacité des Ressources : Les conteneurs utilisent moins de ressources que les machines virtuelles car ils partagent le noyau du système d'exploitation de l'hôte.

Démarrage Rapide : Les conteneurs démarrent rapidement, en quelques secondes, comparé aux minutes nécessaires pour démarrer une machine virtuelle.

Portabilité : Les conteneurs encapsulent toutes les dépendances nécessaires à l'application, assurant une exécution cohérente dans divers environnements sans besoin de configurations spécifiques.

Gestion Simplifiée : Des outils comme Docker facilitent le déploiement, la mise à l'échelle et la gestion des conteneurs par rapport aux machines virtuelles.

Mise à jour et déploiement : Les conteneurs permettent des mises à jour rapides en remplaçant simplement l'image du conteneur, sans nécessiter de redémarrage complet comme pour les machines virtuelles.

Sécurité : Malgré le partage du noyau, les conteneurs sont isolés les uns des autres, réduisant les risques de propagation des vulnérabilités.

2.1.2.3 Processus de création d'images Docker pour encapsuler des applications : définition des Dockerfiles

Un Dockerfile est un fichier texte qui contient les instructions pour construire une image Docker. Il définit les étapes à suivre pour créer l'environnement d'exécution de l'application, y compris l'installation des dépendances, la configuration de l'environnement et la copie des fichiers de l'application.

Exemple d'un dockerfile :

```
FROM arm32v7/alpine:latest

RUN apk update && apk add build-base

WORKDIR /app

COPY main.c .

RUN gcc -o myapp main.c

CMD [ "./myapp" ]
```

Explication des instructions clés d'un Dockerfile :

- **FROM** : Spécifie l'image de base sur laquelle s'appuie l'image que vous construisez.
- **RUN** : Exécute des commandes shell dans l'environnement du conteneur. **COPY** : Copie des fichiers du système hôte vers le système de fichiers du conteneur.
- **ENV** : Définit des variables d'environnement pour l'application.
- **CMD** : Définit la commande à exécuter lorsque le conteneur démarre.

2.1.3 Docker Swarm pour l'orchestration des noeuds docker et la Gestion de Cluster

On a opté pour Docker Swarm afin de gérer efficacement notre cluster. Après avoir installé Docker individuellement sur chaque nœud, on a choisi Docker Swarm en raison de sa capacité à orchestrer et à administrer plusieurs nœuds de manière centralisée. Contrairement à Docker seul, qui ne permet pas la gestion à travers plusieurs nœuds, Docker Swarm offre une solution intégrée pour le déploiement et la gestion des conteneurs sur un cluster, facilitant ainsi la mise à l'échelle et la gestion des applications distribuées.

Docker Swarm est plus simple à configurer que Kubernetes par exemple. Il offre des capacités de gestion de cluster moins étendues que Kubernetes, mais il convient particulièrement aux déploiements directs et rapides. Cela en fait un choix idéal pour les projets nécessitant une solution de gestion de conteneurs efficace et facile à mettre en œuvre.

2.1.4 Kubernetes

Kubernetes, souvent abrégé en K8s, est une plateforme open source d'orchestration de conteneurs. Elle automatise le déploiement, la gestion et la mise à l'échelle des applications conteneurisées sur un cluster d'ordinateurs.

2.1.4.1 Avantages de Kubernetes

Portabilité et flexibilité : Déploiement cohérent sur divers environnements (local, cloud, hybride).

Mise à l'échelle automatique : Ajuste dynamiquement les ressources en fonction de la charge de travail.

Gestion des Échecs et Résilience : Redémarrage automatique des conteneurs défectueux et réplication des pods.

Déploiements et Rollbacks Faciles : Permet des mises à jour progressives et des retours rapides en cas de problème.

Isolation et Sécurité : Isolations des applications via namespaces et gestion sécurisée des secrets.

Efficacité des Ressources : Optimisation de l'utilisation des ressources (CPU, mémoire) pour réduire les coûts.

2.1.5 Slurm, Gestionnaire de Tâches

SLURM est un logiciel de gestion de ressources utilisé principalement dans les environnements informatiques à grande échelle, tels que les clusters de supercalculateurs et les centres de données. Il aide à coordonner et à attribuer les ressources informatiques, comme les processeurs et la mémoire, entre différents utilisateurs et applications.

On utilise SLURM pour optimiser l'utilisation des ressources disponibles et pour planifier les tâches de manière efficace. Cela permet aux utilisateurs de soumettre leurs travaux, de spécifier les ressources nécessaires, et SLURM se charge de les allouer de manière équitable tout en respectant les priorités et les politiques définies par l'administrateur du système. En résumé, SLURM simplifie la gestion des calculs complexes en organisant les ressources de manière ordonnée et efficace.

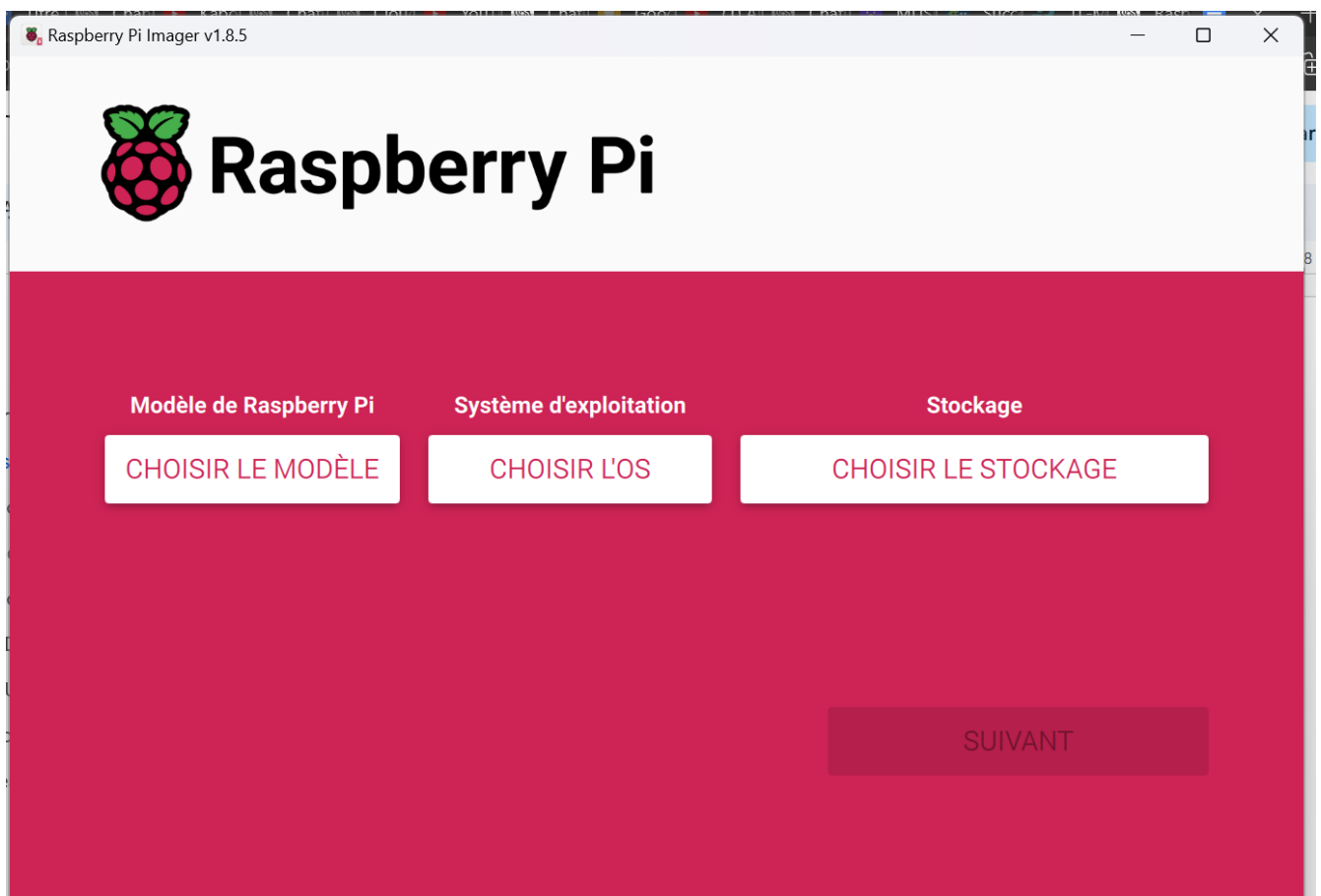
Chapitre 3

Configuration

3.1 Configuration des raspberry Pis

3.1.1 Préparation des raspberry et Installation des systèmes d'exploitation

Pour l'installation des systemes d'exploitaion, on a utilisé le logiciel RaspberryPi Imager.



On a choisi comme OS une version 64 bits, minimale de Raspberry Pi OS basée sur Debian 12 (Bookworm), destinée à des configurations sans interface graphique, et sortie le 15 mars 2024.

En utilisant ce programme on a écrit l'image de Raspbian sur la carte SD. Et on a attribué un nom à chacune de nos 4 Raspberry Pi (rpi1, rpi2, rpi3, rpi4)

3.1.2 Association avec un réseau wifi

Pour associer nos machines avec un reseau wifi, on a ajouter sur le fichier `/etc/wpa_supplicant/wpa_supplicant.conf`

```
country=FR
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
update_config=1

network={
    ssid="zakpi"
    psk="123456789"
    key_mgmt=WPA-PSK
}
```

3.1.3 Activation du service ssh pour qu'on puisse nous connecter à distance à chacune de nos Raspberry Pis

Pour activer le service ssh avant le boot, on a du creer un fichier ssh sans extension sur notre espace de travail.

Connection avec SSH :

```
$ ssh pi@ip_address or pi@name
```

Password : Le mot de passe par défaut est raspberry, on le change après en utilisant la commande `passwd`

3.1.4 Attribution d'une adresse ip statique pour nos raspberry pi

Pour cette étape, on doit installer le service `dhcpcd` :

```
$ sudo apt install dhcpcd
```

Puis, on dois changer le contenu du fichier `/etc/dhcpcd.conf` en ajoutant la section suivante :

```
interface wlan0
static ip_address=<IP_ADDRESS>/24
static routers=<ROUTER_IP_ADDRESS>
static domain_name_servers=<DNS_SERVER_IP_ADDRESS>
```

En faisant ça, nos raspberries sont prêtes à accueillir Docker.

3.2 Configuration de Docker

Docker est une plateforme open-source qui automatise le déploiement, la mise à l'échelle et la gestion des applications en utilisant la technologie des conteneurs. Les conteneurs emballent une application et ses dépendances dans une unité portable qui s'exécute de manière cohérente dans différents environnements.

3.2.1 Installation de Docker sur les Raspberry Pis

3.2.1.1 Mise à jour du Système

Mise à jour du système d'exploitation en exécutant

```
$sudo apt update  
$sudo apt upgrade
```

3.2.1.2 Installation de Docker

En utilisant le script officiel fourni par Docker :

```
$ curl -sSL https://get.docker.com | sh
```

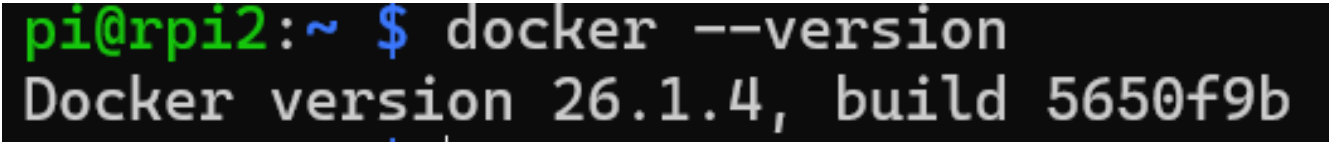
3.2.1.3 Ajout de l'utilisateur au Groupe Docker

Si on souhaite utiliser Docker sans "sudo" :

```
$ sudo command --addtogroup groupname $username  
  
$ sudo usermod -aG docker $USER
```

3.2.1.4 Vérification de l'Installation

```
$ sudo docker --version
```



```
pi@rpi2:~ $ docker --version  
Docker version 26.1.4, build 5650f9b
```

3.2.1.5 Démarrage Automatique de Docker

```
$ sudo systemctl enable docker
```

```
$ sudo systemctl start docker
```

3.3 Configuration de Docker Swarm

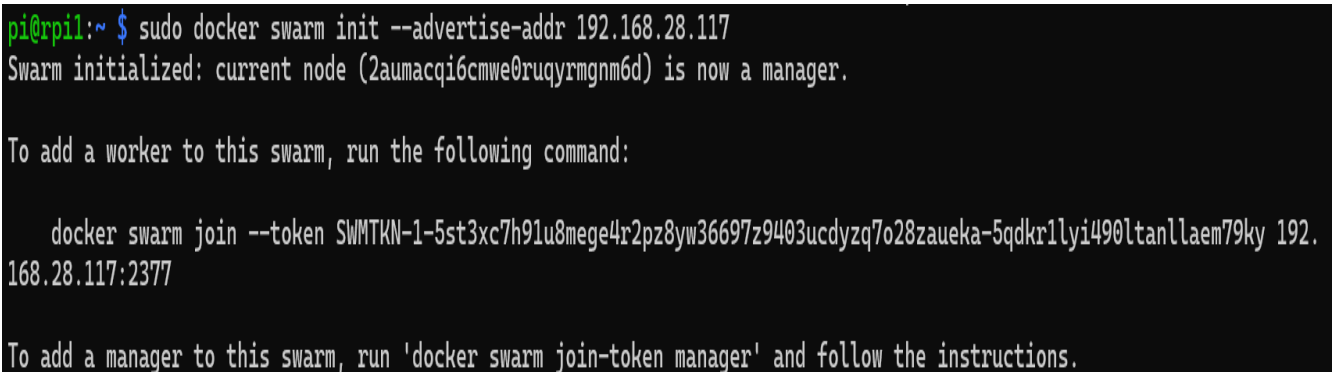
Docker Swarm est une fonctionnalité de Docker qui permet de gérer un cluster de machines comme une seule instance de Docker.

3.3.1 Initialisation de swarm

3.3.1.1 Sélection d'un Nœud Manager

On a choisi `rpil` pour agir en tant que nœud manager du cluster Docker Swarm. Sur ce nœud, On a initialisé Docker Swarm :

```
$ docker swarm init --advertise-addr <IP_du_manager>
```

A terminal window on a Raspberry Pi showing the command `sudo docker swarm init --advertise-addr 192.168.28.117` being executed. The output indicates that the swarm is initialized and the current node is now a manager. It also provides a token for adding workers and instructions for adding more managers.

```
pi@rpil:~$ sudo docker swarm init --advertise-addr 192.168.28.117
Swarm initialized: current node (2aumacqi6cmwe0ruqyrmgnm6d) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-5st3xc7h91u8mege4r2pz8yw36697z9403ucdyzq7o28zaueka-5qdkr1lyi490ltanllaem79ky 192.168.28.117:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.
```

Cette commande génère un jeton de commande `docker swarm join` que les autres nœuds (workers) utilisent pour rejoindre le cluster.

3.3.1.2 Rejoindre des Nœuds Workers au Swarm

Sur chaque Raspberry Pi destiné à agir en tant que nœud worker, j'exécute la commande `docker swarm join` fournie par le nœud manager.


```
$ docker swarm join --token <token> <IP_du_manager>:2377
```

```
pi@rpi2:~ $ docker swarm join --token SWMTKN-1-5st3xc7h91u8mege4r2pz8yw36697z9403ucdyzq7o28zaueka-5qdkr1lyi490ltanl1aem79ky 192.168.28.117:2377
This node joined a swarm as a worker.
```

```
pi@rpi3:~ $ sudo docker swarm join --token SWMTKN-1-5st3xc7h91u8mege4r2pz8yw36697z9403ucdyzq7o28zaueka-5qdkr1lyi490ltanl1aem79ky 192.168.28.117:2377
This node joined a swarm as a worker.
```

```
pi@rpi4:~ $ sudo docker swarm join --token SWMTKN-1-5st3xc7h91u8mege4r2pz8yw36697z9403ucdyzq7o28zaueka-5qdkr1lyi490ltanl1aem79ky 192.168.28.117:2377
This node joined a swarm as a worker.
```

3.3.1.3 Vérification du Cluster Swarm

On vérifie que tous les nœuds ont rejoint le cluster en exécutant

```
$ sudo docker node ls
```

```
pi@rpil:~ $ sudo docker node ls
```

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS	ENGINE VERSION
2aumacqi6cmwe0ruqyrmgnm6d *	rpi1	Ready	Active	Leader	26.1.4
lmjo4l8wwitch7n94alvl39td	rpi2	Ready	Active		26.1.4
n5cjs2vidlhnid1lnconkkyv	rpi3	Ready	Active		26.1.4
j3rez3d273svt8j3kbrngy5ai	rpi4	Ready	Active		26.1.4

```
pi@rpil:~ $ |
```

3.4 Configuration de SLURM (Simple Linux Utility for Resource Management)

3.4.1 Installation et Configuration de Munge

3.4.1.1 Installation de Munge

Nous avons installé Munge, un outil d'authentification requis par Slurm, sur tous les nœuds du cluster

```
$ sudo apt-get install munge libmunge-dev libmunge2
```

3.4.1.2 Génération et distribution de la clé Munge

Nous avons généré une clé Munge sur le nœud de contrôle, puis nous l'avons copiée sur chaque nœud de calcul.

— **Sur le nœud Master :**

```
$ sudo dd if=/dev/urandom bs=1 count=1024 of=/etc/munge/munge.key
```

```
$ sudo chown munge:munge /etc/munge/munge.key
```

```
$ sudo chmod 400 /etc/munge/munge.key
```

Distribution de la clé via la commande scp :

```
$ sudo scp /etc/munge/munge.key pi@rpi1:/tmp/munge.key
```

```
$ sudo scp /etc/munge/munge.key pi@rpi2:/tmp/munge.key
```

```
$ sudo scp /etc/munge/munge.key pi@rpi3:/tmp/munge.key
```

— **Sur les nœuds de calcul**

```
$ sudo mv /tmp/munge.key /etc/munge/munge.key
```

```
$ sudo chown munge:munge /etc/munge/munge.key
```

```
$ sudo chmod 400 /etc/munge/munge.key
```

3.4.1.3 Démarrage du service Munge

Nous avons ensuite démarré le service Munge sur tous les nœuds.

```
$ sudo systemctl enable munge
```

```
$ sudo systemctl start munge
```

```
$ sudo systemctl status munge
```

```
pi@rpil:~ $ sudo systemctl status munge
● munge.service - MUNGE authentication service
   Loaded: loaded (/lib/systemd/system/munge.service; enabled; preset: enabled)
   Active: active (running) since Mon 2024-07-01 04:45:52 CEST; 35min ago
     Docs: man:munged(8)
   Process: 893 ExecStart=/usr/sbin/munged $OPTIONS (code=exited, status=0/SUCCESS)
  Main PID: 903 (munged)
    Tasks: 4 (limit: 8739)
      CPU: 182ms
   CGroup: /system.slice/munge.service
           └─903 /usr/sbin/munged
```

3.4.2 Installation de Slurm

Nous avons installé Slurm sur tous les nœuds du cluster.

```
$ sudo apt-get install slurm-wlm
```

3.4.2.1 Configuration de slurm.conf

Sur le nœud de contrôle, nous avons créé et configuré le fichier `/etc/slurm/slurm.conf` pour définir les paramètres du cluster.

```
$ sudo nano /etc/slurm/slurm.conf
```

```
# SLURM Configuration File

ClusterName=pi_cluster
SlurmctldHost=rpil
SlurmdSpoolDir=/var/spool/slurmd
SlurmUser=slurm
SlurmdUser=root
StateSaveLocation=/var/spool/slurmctld

# Nodes definition
NodeName=rpil[1-4] CPUs=4 State=UNKNOWN

# Partition definition
PartitionName=debug Nodes=rpil[1-4] Default=YES MaxTime=INFINITE State=UP
```

3.4.2.2 Distribution de slurm.conf à l'aide de la commande scp

```
$ sudo scp /etc/slurm/slurm.conf pi@rpi2:/etc/slurm/slurm.conf
```

```
$ sudo scp /etc/slurm/slurm.conf pi@rpi3:/etc/slurm/slurm.conf
```

```
$ sudo scp /etc/slurm/slurm.conf pi@rpi4:/etc/slurm/slurm.conf
```

3.4.2.3 Création des répertoires Slurm

Nous avons créé les répertoires nécessaires pour Slurm et configuré les permissions appropriées

```
$ sudo mkdir -p /var/spool/slurmd
```

```
$ sudo mkdir -p /var/spool/slurmd
```

```
$ sudo chown slurm:slurm /var/spool/slurmd
```

```
$ sudo chown slurm:slurm /var/spool/slurmd
```

3.4.2.4 Démarrage des services Slurm

— Sur le Noeud Master

```
$ sudo systemctl enable slurmd
```

```
$ sudo systemctl start slurmd
```

```
$ sudo systemctl status slurmd
```

```
pi@rpi1:~$ sudo systemctl status slurmd
● slurmd.service - Slurm controller daemon
   Loaded: loaded (/lib/systemd/system/slurmd.service; enabled; preset: enabled)
   Active: active (running) since Mon 2024-07-01 05:17:40 CEST; 5min ago
     Docs: man:slurmd(8)
    Main PID: 2557 (slurmd)
      Tasks: 14
        CPU: 250ms
    CGroup: /system.slice/slurmd.service
            └─2557 /usr/sbin/slurmd -D -s
              └─2558 "slurmd: slurmdscriptd"
```

— Sur les Nœuds travailleurs

```
$ sudo systemctl enable slurmd
```

```
$ sudo systemctl start slurmd
```

```
$ sudo systemctl status slurmd
```

```
pi@rpi2:~$ sudo systemctl status slurmd
● slurmd.service - Slurm node daemon
   Loaded: loaded (/lib/systemd/system/slurmd.service; enabled; preset: enabled)
   Active: active (running) since Mon 2024-07-01 05:09:00 CEST; 15min ago
     Docs: man:slurmd(8)
    Main PID: 2509 (slurmd)
      Tasks: 1
         CPU: 80ms
    CGroup: /system.slice/slurmd.service
            └─2509 /usr/sbin/slurmd -D -s
```

3.4.2.5 Vérification de la Configuration

Sur le nœud de contrôle, nous avons exécuté :

```
$ sinfo
```

```
pi@rpi1:~$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
debug*      up    infinite     4   idle rpi[1-4]
```

Le démon de contrôle SLURM (slurmctld) gère les ressources, planifie les tâches, surveille leur exécution, et contrôle l'accès des utilisateurs dans un cluster SLURM. Sans lui, la gestion efficace des ressources et des tâches serait compromise.

Le contrôleur SLURM slurmd est responsable de la gestion globale du cluster SLURM.

Le démon SLURM slurmctld fonctionne sur chaque nœud de calcul individuel dans le cluster SLURM.

3.4.2.6 Surveillance des journaux de SLURM

On vérifie les journaux de SLURM pour tout problème ou mise à jour de statut :

```
$ tail -f /var/log/slurm/slurmd.log
```

```
$ tail -f /var/log/slurm/slurmctld.log
```

After sending a job with *sbatch* We can take a look at its state using *squeue*

Journaux de slurmctld : Ces journaux contiennent des informations relatives au contrôleur SLURM (slurmctld) Ils incluent des détails sur les soumissions de tâches, la planification des tâches, les mises à jour de l'état des tâches et les événements spécifiques au contrôleur. Ces journaux sont généralement stockés dans le répertoire spécifié par la directive LogFile dans le fichier de configuration slurm.conf.

Journaux de slurmd : Ces journaux concernent le démon SLURM (slurmctld) s'exécutant sur chaque nœud de calcul. Ils contiennent des informations sur la disponibilité des ressources, l'exécution des tâches, l'état du nœud et la communication avec le contrôleur. Les journaux de slurmd sont généralement stockés dans le répertoire spécifié par la directive SlurmdLogFile dans le fichier de configuration slurm.conf.

Journaux de tâches : Lorsqu'une tâche est soumise au cluster SLURM, elle génère ses propres fichiers journaux contenant des informations spécifiques à cette tâche. Ces journaux comprennent généralement des détails tels que l'heure de début de la tâche, l'heure de fin, l'utilisation des ressources, la sortie standard, les erreurs rencontrées pendant l'exécution, etc. Les journaux de tâches sont stockés dans le répertoire spécifié par la directive JobCompLoc dans le fichier de configuration slurm.conf.

3.5 Configuration de Kubernetes

On a choisi k3s, pourquoi ?

Le choix de k3s comme distribution de Kubernetes est optimal pour un cluster de Raspberry Pi en raison de sa légèreté et de sa simplicité. k3s, une version allégée de Kubernetes, est conçue pour les appareils à ressources limitées, comme les Raspberry Pi. Sa faible utilisation des ressources et son installation facile en font une solution idéale pour des environnements de petite échelle. De plus, k3s maintient la compatibilité avec les outils Kubernetes standard, garantissant des performances optimales et une gestion efficace des conteneurs, tout en simplifiant la maintenance du cluster.

3.5.1 Prerequis

Une étape nécessaire pour configurer Kubernetes est bien la modification du fichier `/boot/cmdline` en ajoutant la ligne suivante :

```
cgroup_enable=cpuset cgroup_enable=memory cgroup_memory=1
```

3.5.2 Set up du Noeud Maitre

3.5.2.1 Installation de k3s

On lance le script bash fourni par google :

```
$ curl -sfL https://get.k3s.io | sh -
```

3.5.2.2 Verification du status de k3s

```
$ sudo systemctl status k3s
```

```
pi@rpi1:~$ sudo systemctl status k3s
● k3s.service - Lightweight Kubernetes
   Loaded: loaded (/etc/systemd/system/k3s.service; enabled; preset: enabled)
   Active: active (running) since Thu 2024-07-04 03:57:43 GMT; 26s ago
     Docs: https://k3s.io
  Process: 1030 ExecStartPre=/bin/sh -xc ! /usr/bin/systemctl is-enabled --quiet nm-cloud-setup.service 2>/dev/null (
  Process: 1041 ExecStartPre=/sbin/modprobe br_netfilter (code=exited, status=0/SUCCESS)
  Process: 1054 ExecStartPre=/sbin/modprobe overlay (code=exited, status=0/SUCCESS)
 Main PID: 1063 (k3s-server)
    Tasks: 34
   Memory: 499.0M
      CPU: 39.289s
   CGroup: /system.slice/k3s.service
           └─1063 "/usr/local/bin/k3s agent"
             └─2229 "containerd "
```

3.5.2.3 recuperation du jeton pour faire joindre les noeuds travailleurs

```
$ sudo cat /var/lib/rancher/k3s/server/node-token
```

Token :

```
K104010c1d3fb9eb17fddbdea98e337fe4f1979
63d0bc35fdbec39b266c4b4d6c21::server:ad1
f84701fe696660dbdecc8c6d201a5
```

3.5.3 Set up des Noeuds esclaves

Pour cette partie , la commande suivante nous facillite la tache :

```
pi@rpi2:~$ curl -sfL https://get.k3s.io | K3S_URL=https://192.168.165.117:6443 K3S_TOKEN=K104010c1d3fb9eb17fddbdea98e
7fe4f197963d0bc35fdbec39b266c4b4d6c21::server:ad1f84701fe696660dbdecc8c6d201a5 sh -
[INFO] Finding release for channel stable
[INFO] Using v1.29.6+k3s1 as release
[INFO] Downloading hash https://github.com/k3s-io/k3s/releases/download/v1.29.6+k3s1/sha256sum-arm64.txt
[INFO] Downloading binary https://github.com/k3s-io/k3s/releases/download/v1.29.6+k3s1/k3s-arm64
[INFO] Verifying binary download
[INFO] Installing k3s to /usr/local/bin/k3s
[INFO] Skipping installation of SELinux RPM
[INFO] Creating /usr/local/bin/kubectl symlink to k3s
[INFO] Creating /usr/local/bin/crictl symlink to k3s
[INFO] Skipping /usr/local/bin/ctr symlink to k3s, command exists in PATH at /usr/bin/ctr
[INFO] Creating killall script /usr/local/bin/k3s-killall.sh
[INFO] Creating uninstall script /usr/local/bin/k3s-agent-uninstall.sh
[INFO] env: Creating environment file /etc/systemd/system/k3s-agent.service.env
[INFO] systemd: Creating service file /etc/systemd/system/k3s-agent.service
[INFO] systemd: Enabling k3s-agent unit
Created symlink /etc/systemd/system/multi-user.target.wants/k3s-agent.service → /etc/systemd/system/k3s-agent.service.
[INFO] systemd: Starting k3s-agent
pi@rpi2:~$
```

3.5.3.1 Verification du status de k3s sur les noeuds esclaves

```
$ sudo systemctl status k3s
```

```
pi@rpi4:~$ sudo systemctl status k3s-agent
● k3s-agent.service - Lightweight Kubernetes
   Loaded: loaded (/etc/systemd/system/k3s-agent.service; enabled; preset: enabled)
   Active: active (running) since Thu 2024-07-04 04:27:11 GMT; 2min 17s ago
     Docs: https://k3s.io
  Process: 151678 ExecStartPre=/bin/sh -xc ! /usr/bin/systemctl is-enabled --quiet nm-cloud-setup.service 2
  Process: 151680 ExecStartPre=/sbin/modprobe br_netfilter (code=exited, status=0/SUCCESS)
  Process: 151681 ExecStartPre=/sbin/modprobe overlay (code=exited, status=0/SUCCESS)
 Main PID: 151682 (k3s-agent)
    Tasks: 38
   Memory: 318.5M
      CPU: 18.154s
   CGroup: /system.slice/k3s-agent.service
           └─151682 "/usr/local/bin/k3s agent"
             └─152529 "containerd "
               └─153982 /var/lib/rancher/k3s/data/1a8e734aa3b590325597cdd42d205ac3d6ecc27e506e36563c83b9d77602a
```

3.5.4 Verification de letat du cluster

```
$ sudo k3s kubectl get nodes
```

```
pi@rpi1:~$ sudo k3s kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
rpi1	Ready	control-plane,master	37m	v1.29.6+k3s1
rpi2	Ready	<none>	12m	v1.29.6+k3s1
rpi3	Ready	<none>	8m42s	v1.29.6+k3s1
rpi4	Ready	<none>	7m52s	v1.29.6+k3s1

Chapitre 4

Déploiement

4.1 Déploiement d'une application avec docker et kubernetes comme outil d'orchestration

4.1.1 Compilation de l'application

Avant tout, on a compilé notre application :

```
$ gcc -fopenmp -o fibo mainfibo.c
```

4.1.2 Creation du Dockerfile

On a créé un fichier Dockerfile pour définir l'environnement nécessaire à l'exécution de notre application.

```
FROM ubuntu:22.04
RUN apt-get update && apt-get install -y gcc libomp-dev
COPY fibo /usr/local/bin/
ENV OMP_NUM_THREADS=4
#ENTRYPOINT ["/usr/local/bin/fibo"]
CMD ["/usr/local/bin/fibo", "50"]
```

4.1.3 Creation de l'image docker

```
$ sudo docker build -t /repertoire/ou/on a/Dockerfile
```

```

pi@rpil:~/fibonacci $ sudo docker build -t fibo3 .
[+] Building 1.9s (9/9) FINISHED
=> [internal] load build definition from Dockerfile                                docker:default 0.0s
=> => transferring dockerfile: 236B                                              0.0s
=> [internal] load metadata for docker.io/library/ubuntu:22.04                  1.3s
=> [auth] library/ubuntu:pull token for registry-1.docker.io                    0.0s
=> [internal] load .dockerignore                                                 0.0s
=> => transferring context: 2B                                                  0.0s
=> [1/3] FROM docker.io/library/ubuntu:22.04@sha256:340d9b015b194dc6e2a13938944e0d016e57b9679963fdeb9ce021daac43 0.0s
=> [internal] load build context                                                0.0s
=> => transferring context: 27B                                                0.0s
=> CACHED [2/3] RUN apt-get update && apt-get install -y gcc libomp-dev         0.0s
=> CACHED [3/3] COPY fibo /usr/local/bin/                                       0.0s
=> exporting to image                                                           0.2s
=> => exporting layers                                                         0.0s
=> => writing image sha256:2280da3880e97d536604bfaae9b2f9bc05d1aa94e7a72c658a0cd235939c55eb 0.0s
=> => naming to docker.io/library/fibo3                                         0.0s

```

4.1.4 Upload au Docker HUB

4.1.4.1 Tag

```
$ sudo docker tag fibo zakpi/fibo
```

4.1.4.2 Push

```
$ sudo docker push zakpi/fibo
```

4.1.5 Creation du fichier .yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: fibo3
spec:
  replicas: 3
  selector:
    matchLabels:
      app: fibo3
  template:
    metadata:
      labels:
        app: fibo3
    spec:
      containers:
      - name: fibo3
        image: zakpi/fibo3
        resources:
          limits:
            cpu: "4"
            memory: "512Mi"
        ports:
        - containerPort: 80

```

4.1.6 Deploiment

Finalement, on deploye l'image qu'on a créé dans notre cluster :

```
$ sudo k3s kubectl apply -f dep.yaml
```

4.1.7 Vérification du deploiment

```
$ sudo k3s kubectl get pods
```

```
pi@rpil:~/fibonacci $ sudo k3s kubectl get pods
NAME                                READY   STATUS              RESTARTS   AGE
fibo3-5697c87d7b-dxf4q             0/1     CrashLoopBackOff    10 (3m13s ago)  59m
fibo3-5697c87d7b-sq422             0/1     CrashLoopBackOff    10 (2m58s ago)  59m
fibo3-5697c87d7b-wjdw             0/1     CrashLoopBackOff    16 (3m38s ago)  75m
```

4.1.8 Journal/log

Regarder dans les journaux nous aide a determiner le statut de chaque pod, et les nous aide a fixer les erreurs.

```
$ sudo k3s kubectl logs POD_ID
```

```
pi@rpil:~/fibonacci $ sudo k3s kubectl logs fibo3-5697c87d7b-dxf4q
Entrez n : fib(-361334760) = -361334760
temps pris : 0.000012 sec
```

Notre application s'exécute, mais la variable d'entrée ne coincide pas avec celle qu'on choisi dans le dockerfile.

La commande describe Nous aide a voir un journal plus long

```
$ sudo k3s kubectl describe pod fibo-6799b646d6-qrgtd
```

4.2 Déploiement d'une application avec Docker, Docker-swarm

4.2.1 Théoriquement

Avant de déployer l'application sur notre cluster, on commence par créer une image Docker en compilant l'application et en définissant ses dépendances via un Dockerfile. Cette étape de conteneurisation assure la portabilité et la cohérence de l'application sur tous les nœuds du cluster, facilitant ainsi sa gestion et son déploiement.

Orchestration avec Docker Swarm

Une fois l'image Docker prête, l'étape suivante consiste à utiliser Docker Swarm pour orchestrer le déploiement des conteneurs à travers les nœuds du cluster. Docker Swarm optimise la distribution des conteneurs, garantissant une utilisation efficace des ressources disponibles.

En définissant des services via Docker Swarm, l'application est déployée de manière redondante et équilibrée, renforçant sa disponibilité et améliorant sa tolérance aux pannes dans un environnement de calcul intensif.

Gestion des tâches avec Slurm

Pour optimiser la gestion des tâches de calcul, on essaiera d'intégrer slurm pour planifier et allouer les ressources de manière optimale. Slurm permet de définir des files d'attente, de prioriser les tâches et de surveiller l'utilisation des ressources en temps réel. Cette intégration optimise l'efficacité opérationnelle du cluster.

4.2.2 Pratiquement

4.2.2.1 Création du Dockerfile

On a créé un fichier Dockerfile pour définir l'environnement nécessaire à l'exécution de notre application.

```
FROM ubuntu:22.04

RUN apt-get update && apt-get install -y gcc libomp-dev

COPY fibo /usr/local/bin/

ENV OMP_NUM_THREADS=4

#ENTRYPOINT ["/usr/local/bin/fibo"]

CMD ["/usr/local/bin/fibo", "50"]
```

4.2.3 Construction de l'image Docker

Dans le repertoire ou on a notre Dockerfile, on a lancé cette commande :

```
$ sudo docker build -t fibo-image .
```

Le point dans notre commande indique que docker utilisera les fichiers du repertoire ou on a lancé la commande pour construire notre image docker.

4.2.4 uploading de l'image au Docker Hub

— Login

```
$ sudo docker login
```

— Faire un tag a l'image

```
$ sudo docker tag fibo-imag1:latest zakpi/fibo-imag1:latest
```

— Push l'image au docker hub

```
$ sudo docker push zakpi/fibo-imag1:latest
```

4.2.5 Déploiement, creation d'un service

Creation d'un service avec 3 replicas :

```
$ sudo docker service create --name fibo-service --replicas 3 zakpi/fibo-imag1
```

4.2.6 Creation du fichier docker-compose.yml

Le fichier docker-compose.yml est un fichier de configuration utilisé par Docker Compose pour définir et exécuter des applications multi-conteneurs. Il permet de décrire les services, les réseaux et les volumes nécessaires à notre application dans un format lisible et facile à comprendre :

```
version: '3.8'

services:
  fibo3:
    image: zakpi/fibo3
    deploy:
      replicas: 3
      resources:
        limits:
          cpus: '4'
          memory: 512M
    ports:
      - "80:80"
```

4.2.7 Déploiement avec docker compose .yml

On lance la commande suivante :

```
$ sudo docker stack deploy -c docker-compose.yml fibo3_stack
```

Annexes

Annexe 1

Notre application s'agit d'un programme en langage C qui calcule les nombres de Fibonacci en exploitant la parallélisation offerte par OpenMP. Ce programme se compose de deux fonctions principales pour le calcul des nombres de Fibonacci : une fonction séquentielle pour les petites valeurs et une fonction parallèle pour les valeurs plus grandes.

- **fib_s(int n)** : Cette fonction calcule les nombres de Fibonacci d'une manière séquentielle lorsqu'elle est inférieure à un seuil.
- **fib(int n)** : Cette fonction est conçue pour tirer parti de la parallélisation via OpenMP. Pour les valeurs de n supérieures au seuil, elle divise les calculs en tâches parallèles. Plus précisément, elle utilise la directive `pragma omp task` pour créer des tâches distinctes pour les appels récursifs, ce qui permet d'exécuter ces calculs en parallèle et de réduire le temps de calcul total. Si n est inférieur ou égal au seuil, la fonction appelle **fib_s** pour un calcul séquentiel.

L'utilisation d'OpenMP dans ce programme permet d'accélérer le calcul des nombres de Fibonacci pour des valeurs élevées de n, en exploitant la puissance de traitement parallèle des processeurs modernes.

1 Code source

```
#include <stdio.h>
#include <omp.h>

#define SEUIL 30

int fib_s(int n)
{
    int x, y;
    if (n < 2){
        return n;
    }
    x = fib_s(n - 1);
    y = fib_s(n - 2);

    return x+y;
}

int fib(int n)
{
    if (n < 2)
        return n;
```

```

// eviter le nombre de tests de if seuil < 30
if (n <= SEUIL)
    return fib_s(n);

// if (n <= SEUIL)
//     return fib(n-1) + fib(n-2);

int x,y;
#pragma omp parallel shared(n)
#pragma omp single
{
    #pragma omp task
    x = fib(n-1);

    #pragma omp task
    y = fib(n-2);
}

return x + y;
}

int main()
{
    int n, fibonacci;
    double starttime;
    printf("Entrez n: ");
    scanf("%d", &n);
    starttime = omp_get_wtime();

    fibonacci = fib(n);

    printf("fib(%d) = %d\n", n, fibonacci);
    printf("temps pris: %lf sec\n", omp_get_wtime() - starttime);
    return 0;
}

```


Bibliographie

- [1] https://www.reddit.com/r/docker/comments/bebuo4/dockerecr_image_could_not_be_accessed_on_a/, 2019.
- [2] <https://github.com/docker/for-linux/issues/400>, 2019.
- [3] <https://stackoverflow.com/questions/45502822/docker-swarm-error-response-from-daemon>, 2020.
- [4] <https://github.com/NVIDIA/deepops/blob/master/workloads/examples/slurm/mpi-hello/README.md>, 2021.
- [5] <https://github.com/stackhpc/slurm-k8s-cluster>, 2021.
- [6] <https://github.com/stackhpc/slurm-k8s-cluster>, 2021.
- [7] https://www.reddit.com/r/homelab/comments/csotwr/pi_4_slurm_k8s_cluster/, 2021.