



MASTER OF SCIENCE
IN ENGINEERING

Hes·SO

Haute Ecole Spécialisée
de Suisse occidentale

Fachhochschule Westschweiz

University of Applied Sciences and Arts
Western Switzerland

Master of Science HES-SO in Engineering
Av. de Provence 6
CH-1007 Lausanne

Master of Science HES-SO in Engineering

Orientation : Information and Communication Technologies
(ICT)

Dockerisation d'environnement pour Les projets de bioinformatique

Author :

Déruaz Vincent

Under the direction of :
Prof. Carlos Andrés Pena
CI4CB at HEIG-VD

External expert :
[Title] [FirstName] [LastName]
Company/Lab

Lausanne, HES-SO//Master, 6 février 2017

« Nous sommes comme des nains assis sur des épaules de géants. Si nous voyons plus de choses et plus lointaines qu'eux, ce n'est pas à cause de la perspicacité de notre vue, ni de notre grandeur, c'est parce que nous sommes élevés par eux. »
— Metalogicon de Jean de Salisbury

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Key words :

Résumé

Aujourd'hui les médicaments perdent du terrain sur les bactéries. En effet, les bactéries deviennent résistantes. Une des alternatives possibles à l'utilisation des antibiotiques est la phagothérapie. Il s'agit d'utiliser des virus *mangeurs* de bactéries pour signer les infections bactériennes. Un des avantages majeurs à la phagothérapie est que les virus sont très spécifiques et ne s'attaquent qu'aux bactéries ciblées.

Ce projet fait suite à une autre thèse de master, [*Modélisation prédictive des interactions entre bactéries et virus bactériophages - Leite Diogo*]. Dans cette première thèse, il est question de tenter d'établir une méthode, afin de rapidement identifier les bactériophages capables de s'attaquer à une bactérie cible. Pour l'instant cette procédure est effectuée manuellement en laboratoire en testant les combinaisons possibles d'interaction virus/bactéries.

Dans la thèse [*Modélisation prédictive des interactions entre bactéries et virus bactériophages - Leite Diogo*], une solution été proposée. Cette thèse a pour objectifs d'améliorer cette solution. Notamment au niveau des performances et de son utilisation.

Pour se faire cette thèse explore une solution qui met en place un environnement Docker. Docker est un logiciel de conteneurisation. Docker permet s'automatiser le déploiement d'applications complexes de manière automatisée.

L'application produite durant ce travail permet de lancer une architecture Docker et d'exécuter l'application de manière autonome sans nécessité d'interaction utilisateur. De plus, il est possible de fournir autant de fichiers de configurations que l'on souhaite, et de produire ainsi autant de *dataset* que de configurations.

Table des matières

Abstract (English/Français)	i
Liste des figures	vii
1 Introduction	1
2 Etats de l'art	3
2.1 Introduction	3
2.2 Automatisation	3
2.3 Configuration	3
2.4 Hmmer	4
2.5 Parallélisation	4
2.5.1 Simple	4
2.5.2 Avancée	5
2.6 Optimisations	5
2.7 Conclusion	6
3 Bases de Docker	7
3.1 Introduction	7
3.1.1 Utilisations	7
3.1.2 Compatibilité inter-OS	7
3.1.3 Miracle or illusion	8
3.2 Pré-requis	8
3.2.1 Connaissance	8
3.2.2 Installations	8
3.2.3 Téléchargements	10
3.3 Fonctionnement	10
3.3.1 Docker	10
3.3.2 Docker-compose	10
3.4 Exemples	11
3.4.1 simple pull, build et run	11
3.4.2 Serveur Web : Docker compose	14
3.4.3 Parallélisation	16
4 Parallelisation python3	19
4.1 Code de base	19
5 Environnement et application	21
5.1 Images Docker	21
5.1.1 Hmmer	21
5.1.2 Database	21
5.1.3 Core	24
5.2 Docker Compose	24
5.3 «Inphinity Environment»	28
5.3.1 Diagramme de classes	29
5.3.2 Configuration	32
5.3.3 Logs	35

6	Déploiement et execution	37
6.1	Obtention des sources	37
6.2	Edition de la configuration	37
6.3	Execution de la composition Docker	37
7	Simplification d'usage	39
7.1	Commandes et alias	39
7.2	Scripts	39
8	Améliorations	41
8.1	Parallélisation	41
8.2	Machines Amazone	41
9	Conclusion	43
A	Annexes	45

Table des figures

2.1	Tableau performances Cython	6
3.1	vm vs Docker	7
3.2	Code - Installation Docker	8
3.3	docker services	9
3.4	Code - Configurer utilisateur Docker	9
3.5	Code - Installation Docker-compose	9
3.6	Code - Docker-compose version	9
3.7	Code - Docker pull	11
3.8	Code - Docker images	11
3.9	Code - Docker rmi	11
3.10	Docker run linux	12
3.11	Docker ps	12
3.12	Docker exec	12
3.13	Docker run	13
3.14	Docker build	13
3.15	mysql Docker	14
3.16	php-apache Docker	14
3.17	Code - docker-compose configuration	15
3.18	Docker-compose up	15
3.19	phpinfo	16
3.20	Docker Hmmer	17
5.1	Création bases de données	23
5.2	Core installations	24
5.3	architecture	25
5.4	Service base de données	26
5.5	Service contrôleur	27
5.6	classdiagram	29
5.7	Processus pool	30
5.8	Division en Chunks	30
5.9	Popen Docker Hmmer	31
5.10	Popen remove	32
5.11	Main	32
5.12	Fichier de configuration	33
5.13	hmmseach	35
5.14	Logger	36
6.1	Code - Obtention de sources	37
6.2	Code - Edition de la configuration	37
6.3	Code - Execution composition	38
6.4	Code - Execution composition vérification	38
6.5	Code - Execution Application	38
7.1	Code - Commande docker exec	39
7.2	Code - Alias	39

1 | Introduction

Ce travail a été réalisé dans le cadre du projet INPHINITY pour le groupe CI4CB de la HEIG-VD. Avec l'émergence de bactéries résistantes aux antibiotiques devenant une problématique mondiale qui menace les progrès de la médecine moderne, une alternative prometteuse pour lutter contre des bactéries multirésistantes consiste à utiliser leurs prédateurs naturels, des bactériophages, virus mangeurs de bactéries. Ces bactériophages, inoffensifs pour l'homme, sont extrêmement spécifiques, ne reconnaissant qu'un type bien précis de bactéries. Ceci présente l'avantage de ne pas détériorer la flore bactérienne humaine, mais pose, par contre, une limitation pour leur développement rapide. En effet, pour chaque type de bactérie il faut trouver le bactériophage correspondant. Face à la nécessité d'examiner systématiquement une multitude d'interactions possibles, le développement rapide des bactériophages comme alternative aux antibiotiques ne pourra se faire qu'avec l'aide d'un modèle permettant de prédire les interactions entre bactériophages et bactéries. Ceci permettra notamment de réduire le nombre de validations expérimentales nécessaires à l'identification du bactériophage approprié et contribuera à l'essor de cette voie thérapeutique.

Ce travail se place également dans la continuité d'une précédente thèse de master dont l'objectif était de prouver la pertinence d'une méthode d'analyse par *machine learning*. En effet, il s'agit d'une méthode permettant d'identifier rapidement les bactériophages capables de s'attaquer à une bactérie cible. Pour l'instant cette procédure est effectuée manuellement en laboratoire en testant les combinaisons possibles d'interaction virus/bactéries.

Dans la présente thèse, il est question de mettre en place plusieurs aspects permettant l'enrichissement du processus d'analyse de la thèse [*Modélisation prédictive des interactions entre bactéries et virus bactériophages - Leite Diogo*].

Afin de réaliser ces objectifs, une première phase du travail a consisté à réaliser des états de l'art pour les différents domaines utilisés (cf.chapitre 2-Etats de l'art).

Plusieurs phases distinctes de travail ont été nécessaires durant ce travail.

Premièrement, il a fallu reprendre la thèse [*Modélisation prédictive des interactions entre bactéries et virus bactériophages - Leite Diogo*] et comprendre ce qu'il y a été fait. Les informations concernant la thèse de Mr.Leite Diogo nécessaires à la compréhension de ce travail ont été abordées dans l'introduction, pour davantage d'informations veuillez consulter la thèse en question.

Deuxièmement, une fois les objectifs de thèse fixés, il a été important de réaliser un état de l'art des différentes technologies et aspects techniques susceptibles d'être utilisés dans la présente thèse, voir chapitre 2-Etats de l'art.

Troisièmement, c'est uniquement après ces deux premières phases que le développement a pu commencer, voir chapitre 4-Parallelisation python3 et chapitre 5-Environnement et application. Durant cette phase, un certain nombre d'aspects ont été développés : notamment, l'utilisation de python 3 afin de remplacer l'utilisation de python2, moins efficace.

De plus, on souhaite être capable d'automatiser le lancement de "l'application" et par la même occasion rendre le déploiement facile et unifié, quelle que soit la machine hôte, pour autant qu'elle utilise le système d'exploitation Linux. Ensuite, on souhaite pouvoir lancer l'analyse pour différentes configurations, créées à l'avance. Un autre objectif important était de remplacer l'utilisation d'une API

en ligne par une utilisation de sa version locale cf.chapitre 6-Déploiement et execution. On souhaite également qu'il soit possible de fournir autant de fichiers de configurations que l'on souhaite, et de produire ainsi autant de *dataset* que de configurations.

Finalement, le temps de travail étant limité, il faut penser aux utilisations futures de ce qui a été développé. Ceci passe notamment par l'utilisation de l'application réalisée dans ce travail de manière simple voir chapitre 7-Simplification d'usage, mais aussi par les améliorations possibles à cette thèse, voir chapitre 8-Améliorations. C'est pour cela qu'un environnement de développement et d'exécution Docker a été produit dans ce travail, qui pourra être utile à d'autres membres du projet.

2 | Etats de l'art

2.1 Introduction

Dans ce chapitre, nous aborderons les différentes pistes envisagées afin de remplir les objectifs fixés dans cette thèse, comme listés dans l'introduction (chapitre 1-Introduction).

Avant toutes choses, il a fallu se mettre au niveau et comprendre la thèse [*Modélisation prédictive des interactions entre bactéries et virus bactériophages* - Leite Diogo] .

2.2 Automatisation

En terme d'automatisation, une pratique bien courante chez les développeurs s'agit d'utiliser des scripts bash afin de pouvoir exécuter un certain nombre de commandes et de codes successivement. Bien que cette méthode présente l'avantage d'être simple, il suffit d'une console UNIX et d'un éditeur de texte, elle présente un défaut majeur. En effet, le développeur du script contrôle quelle commande et code sont exécutés et peut également définir des paramètres pour ceux-ci, mais il ne peut pas contrôler l'environnement d'exécution.

Une façon de faire, en plein essor depuis quelque temps, est l'utilisation de la plateforme Docker. Il s'agit d'un logiciel de containerisation. C'est-à-dire la création de briques d'applications, qui mises ensemble permettent de réaliser une application globale. De plus, le développement d'une telle solution permet un partage facilité grâce à un déploiement facilité et autonome. Pour davantage d'explications sur le sujet je vous renvoie au chapitre 3-Bases de Docker.

Vous l'aurez bien compris, le choix qui a été fait est celui de l'utilisation de Docker.

2.3 Configuration

En ce qui concerne la recherche d'une méthode afin de réaliser facilement des fichiers de configurations précréées, beaucoup de solutions existent. Ces différentes méthodes sont plus ou moins flexibles aux modifications.

Les fichiers de configurations dont il est question ici sont spécifiques à la partie python du code qui sera exécuté par notre application chapitre 5-Environnement et application. En effet, l'on souhaite entre autres être capable de donner des fichiers de configuration en entrée et d'obtenir pour chacun un résultat en sortie.

Nous citerons ici uniquement la solution retenue, car les autres solutions trouvées sont soit trop incompatibles soit presque identiques à la solution retenue.

Nous utilisons le module python *Configparser*, qui permet de lire et parser des fichiers à l'extension .ini de manière simple. De plus, la structure d'un fichier .ini est très simple et ne laisse donc que très peu de place aux erreurs de format.

2.4 Hmmer

Dans la thèse [*Modélisation prédictive des interactions entre bactéries et virus bactériophages - Leite Diogo*] les séquences protéiniques sont recherchées dans la base de données de profile-HMM à l'aide d'une interface de programmation applicative (API) en ligne. Cette API est disponible depuis le site <https://www.ebi.ac.uk/Tools/hmmer/>.

Comme dit précédemment, un des objectifs de ce travail est de se passer de l'utilisation de cette API car son accès n'est pas toujours disponible ou stable.

Une recherche rapide a permis de se rendre compte que l'application utilisée derrière cette API est disponible au téléchargement et peut donc être utilisée de manière locale. Pour davantage d'information chapitre 5-Environnement et application, sous-chapitre Hmmer.

2.5 Parallélisation

La version existante du code se trouvant dans la thèse [*Modélisation prédictive des interactions entre bactéries et virus bactériophages - Leite Diogo*] est une version sous forme de script, proof-of-concept, en python2 et non *multiprocessed*. Afin de garantir une utilisation optimale des ressources de la machine hôte, sur laquelle le code est exécuté, nous souhaitons rendre le code parallèle là où il est possible de le faire.

Plusieurs solutions sont possibles, encore une fois les solutions les plus compliquées ne sont pas toujours les plus efficaces. De plus une méthode trop complexe pourrait réduire la bonne transmission du code à d'autres développeurs.

La partie principale que l'on souhaite paralléliser est l'utilisation de la fonction de scanne de HMMER, étant donné qu'un très grand nombre de séquences protéiniques doivent être analysées.

2.5.1 Simple

Docker

Docker, mis à part de rendre le déploiement et l'exécution d'application automatisée, permet également de lancer plusieurs conteneurs simultanément, chapitre 3-Bases de Docker. Un conteneur englobe un système de fichier complet possédant tout ce qui est nécessaire à remplir sa fonction.

Python

En python on retrouve deux principales méthodes permettant de réaliser de code parallèle. En effet, on peut utiliser le *multiprocessing* ou le *multithreading*.

Notre but est de réaliser et d'optimiser un code Central processing unit (CPU) dépendant, c'est-à-dire coeurs dépendants. Lors de l'utilisation du langage python il faut savoir qu'avec des codes CPU dépendants, python limite les possibilités de parallélisme à cause de la Global Interpreter Lock (GLI). La GLI est nécessaire en python, car python n'est pas *tread safe*. En effet, il y a, en python, un verrou global lorsque l'on essaye d'accéder à un objet depuis un thread.

À cause de se verrou les codes CPU dépendants ne gagneront pas en performance lorsqu'ils sont parallélisés à l'aide de *multithreading*, mais uniquement avec le *multiprocessing*.

2.5.2 Avancée

Docker Swarm

Une autre méthode utilisant une librairie avancée de Docker, consiste à utiliser Docker Swarm. Docker Swarm apporte à Docker une gestion native du *clustering*, afin de transformer un groupe de *Docker engines* en un unique et virtuel *Docker engine*. Grâce à cela, il est possible d'exécuter une application sur une architecture partagée sur plusieurs systèmes physiquement indépendants.

Spark

Spark est un framework *open source* de calcul distribué. Il permet d'effectuer des analyses complexes sur un grand nombre de données.

Il est également un ensemble d'outils pour le traitement de grandes sources de données, notamment grâce à des fonctions *MapReduce*.

2.6 Optimisations

Le code repris de la thèse [*Modélisation prédictive des interactions entre bactéries et virus bactériophages - Leite Diogo*] est un code séquentiel, sous forme de script nécessitant des inputs utilisateurs à chaque étape. De plus, ce code est écrit en python dans sa version 2.

Grâce au travail du Dr. Brett Cannon, voir ici, on se rend compte que python 3.3 pourrait optimiser les performances de notre application. On peut lire ici que même l'appel des fonctions est en moyenne 1.20 fois plus rapide. De plus, les *threaded count* sont également plus rapides.

Une autre possibilité est d'utiliser *Cython*. Cython est un compilateur/langage de python permettant d'utiliser des appels au langage C et de compiler un code python en exécutable C. Il faut savoir qu'un exécutable C est généralement plus rapide que l'exécution de l'interpréteur Python.

On trouve le tableau suivant dans la documentation de Cython, qui permet de nous rendre compte des différences.

Method	Time (ms)	Compared to Python	Compared to Numpy
Pure Python	183	x1	x0.03
Numpy	5.97	x31	x1
Naive Cython	7.76	x24	x0.8
Optimised Cython	2.18	x84	x2.7
Cython calling C	2.22	x82	x2.7

FIGURE 2.1 – Tableau de comparaison de Cython - http://notes-on-cython.readthedocs.io/en/latest/std_dev.html/).

2.7 Conclusion

Après des tests sur ces différentes technologies et méthodes et quelques discussions ici et là, l'idée ayant été arrêtée est d'utiliser *Docker* et *Docker Compose* comme contexte applicatif et de transformer les scripts en une application orientée objet en python 3.3 gérant les fichiers de configurations avec la librairie *Configparser*. Pour ce qui est du parallélisme, il sera réalisé en utilisant la librairie *Multiprocess*.

3 | Bases de Docker

3.1 Introduction

Dans ce chapitre nous allons parler du fonctionnement de *Docker* et *Docker Compose*. Ce chapitre est réalisé sous le ton d'un cours d'introduction à Docker, afin de pouvoir transmettre les connaissances de base à l'utilisation et à la modification du travail réalisé lors de cette thèse. Cela passera entre autres par certains exemples et codes qui seront fournis en annexe notamment.

Docker permet l'exécution de code dans un conteneur indépendant de votre système hôte.

3.1.1 Utilisations

3.1.2 Compatibilité inter-OS

Docker permet d'éviter les problèmes liés aux différences entre les environnements d'exécution. En effet, lorsque l'on exécute un code avec Docker on contrôle exactement l'état et le type d'environnement d'exécution. Cela rend donc possible l'exécution d'un code sous différents Systèmes d'exploitation (OS) hôte (OSX, Linux, Windows).

Mais pourquoi ne pas utiliser une simple machine virtuelle ? Une première différence entre une machine virtuelle et Docker est le fait que Docker n'encapsule pas tout un OS, ceci permet une exécution beaucoup plus rapide, et c'est bien ce que l'on cherche dans ce travail.

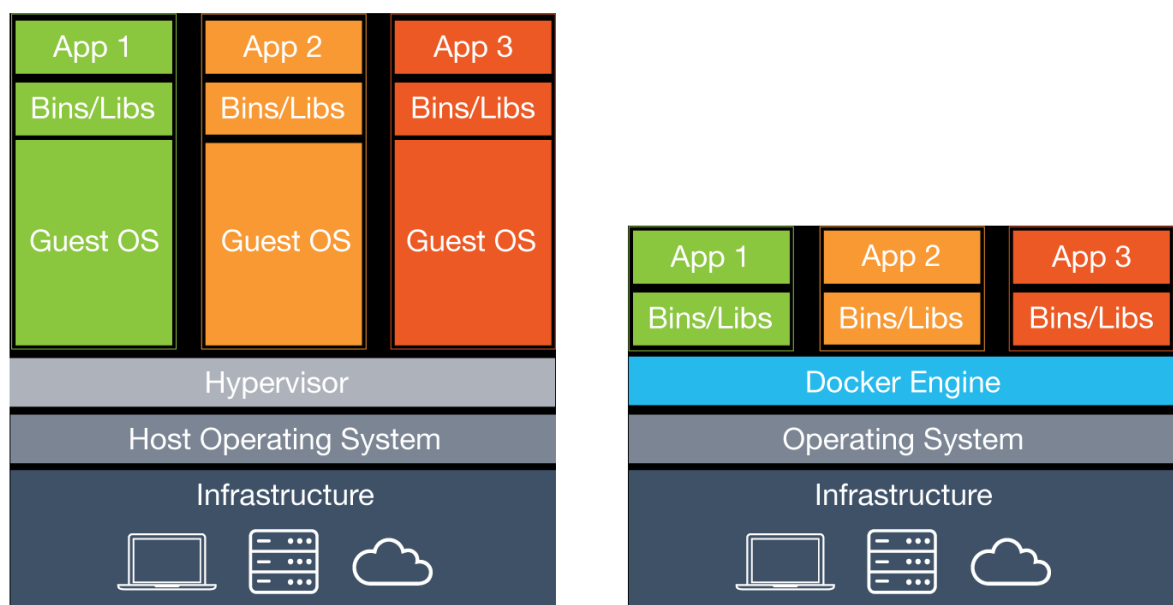


FIGURE 3.1 – Virtual machine vs Docker

3.1.3 Miracle or illusion

Je tiens ici à faire une mise en garde vis-à-vis de l'utilisation de Docker. Dans son utilisation Docker est, à mon sens, une solution assez miraculeuse, notamment par le fait que l'on peut partager une application sans se poser de questions sur l'hôte cible. Mais attention Docker n'est pas aussi miraculeux que cela dans le développement d'une solution applicative. En effet, il peut parfois être compliqué d'arriver du premier coup à réaliser ce que l'on souhaite.

Docker n'est donc pas une solution miracle, mais présente beaucoup d'avantages en termes d'exécution standardisée, de partage de code et de déploiement.

3.2 Pré-requis

3.2.1 Connaissance

Il est nécessaire d'être à l'aise avec l'OS Linux et l'utilisation de commande UNIX. En effet, la plupart du temps les conteneurs utiliseront un système Linux. Pour plus d'informations cf. ci-après.

3.2.2 Installations

Installation Docker

```
$ sudo apt-get update
$ sudo apt-get install apt-transport-https ca-certificates
$ sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 --
  ↪ recv-keys 58118E89F3A912897C070ADB76221572C52609D

$ sudo apt-add-repository 'deb https://apt.dockerproject.org/repo ubuntu
  ↪ -xenial main'
$ sudo apt-get update
$ sudo apt-cache policy docker-engine
$ sudo apt-get install -y docker-engine
$ sudo systemctl status docker
```

FIGURE 3.2 – Installation Docker

Vous devriez maintenant voir une sortie console semblable à celle de la figure 3.3 :

```

• docker.service - Docker Application Container Engine
  Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)
  Active: active (running) since Wed 2016-11-30 01:28:41 CET; 8h ago
    Docs: https://docs.docker.com
  Main PID: 1171 (dockerd)
    Tasks: 41
   Memory: 3.7G
      CPU: 1min 18.316s
   CGroup: /system.slice/docker.service
           └─ 1171 /usr/bin/dockerd -H fd://
           └─ 1607 docker-containerd -l unix:///var/run/docker/libcontainerd/docker-containerd.sock --shim docker-con
           └─ 21678 docker-containerd-shim 1ed0c2cbc7dbd228de8ffd82a822df63abc6f75f3978b96a8b9e82832489a343 /var/run/d

```

FIGURE 3.3 – Services Docker opérationnel

Il faut à présent configurer Docker pour votre utilisateur hôte.

```
$ sudo usermod -aG docker $(whoami)
```

FIGURE 3.4 – Configurer utilisateur Docker

À ce point-ci, il vous faut redémarrer votre machine.

Installation Docker-compose (1.9)

```

$ curl -L "https://github.com/docker/compose/releases/download/1.9.0/
  ↪ docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-
  ↪ compose
$ sudo chmod +x /usr/local/bin/docker-compose
$ docker-compose --version

```

FIGURE 3.5 – Installation Docker-compose

Vous devriez à présent obtenir la sortie console suivante :

```
docker-compose version: 1.9.0
```

FIGURE 3.6 – Docker-compose version

3.2.3 Téléchargements

3.3 Fonctionnement

3.3.1 Docker

Il faut commencer par clarifier de quoi on parle lorsque l'on utilise le mot conteneur. Il s'agit d'une *enveloppe* virtuelle permettant de packager une application ou un code avec toutes les dépendances nécessaires au fonctionnement de l'application. On package donc les fichiers source, librairies, runtime, outils, fichiers, base de données, etc.

Un conteneur n'embarque pas de OS, il s'appuie sur celui de l'hôte sur lequel il est déployé. Ce qui rend un conteneur beaucoup moins lourd qu'une machine virtuelle 3.1.

Il faut également spécifier que Docker opère une isolation, des conteneurs, au niveau du système d'exploitation.

Un conteneur Docker est décrit à l'aide d'un simple fichier *.Dockerfile*, il décrit la création du conteneur, en détail. On peut personnaliser cette description de manière très détaillée.

Il faut voir une application réalisée avec Docker comme une somme de microservices. Nous verrons dans la section suivante des exemples basiques d'application Docker. Le but étant de :

- rendre l'application davantage élastique ;
- améliorer les performances ;
- le déploiement continu est facilité. On peut relancer les services indépendamment les uns des autres.

3.3.2 Docker-compose

Docker-compose permet de définir et d'exécuter des applications multi-conteneurs. En effet, sans compose il fallait lancer les différents services de votre application soit manuellement soit en utilisant des scripts.

Compose utilise un fichier de composition, *docker-compose.yml*, afin de configurer une application Docker. Ce qui permet de lancer une application à l'aide d'une seule commande.

Pour résumer, une application Docker, utilisant Compose, est la combinaison de trois étapes :

1. Définir les différents Dockerfile des vos micro services composant l'application ;
2. Définir les services qui seront utilisés dans le composé, leurs relations et leurs configurations ;
3. Lancer l'application avec la simple commande, *docker-compose up*.

3.4 Exemples

3.4.1 simple pull, build et run

Les trois commandes les plus importantes de Docker sont *pull*, *build* et *run*. En effet, ces commandes sont indispensables et doivent impérativement être comprises.

Premièrement, intéressons-nous à la commande *pull* :

```
$ docker pull debian:jessie

jessie: Pulling from library/debian
5040bd298390: Pull complete
Digest: sha256:
  ↳ abbe80c8c87b7e1f652fe5e99ff1799cdf9e0878c7009035afe1bccac129cad8
Status: Downloaded newer image for debian:jessie
```

FIGURE 3.7 – Docker pull

Avec cette commande l'engin Docker a téléchargé l'image de debian Jessie depuis le *Docker Hub*. Vous pouvez trouver un grand nombre d'images sur *Docker Hub* - <https://hub.docker.com/>.

Il est possible de gérer les images contenues sur une machine hôte. La commande suivante permet d'afficher la liste des images :

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED
↳	SIZE		
debian	jessie	e5599115b6a6	2 weeks ago
↳	123 MB		

FIGURE 3.8 – Docker images

Il est également possible de supprimer une image, afin de libérer de l'espace disque :

```
$ docker rmi e5599115b6a6

Untagged: debian:jessie
Untagged: debian@sha256:
  ↳ abbe80c8c87b7e1f652fe5e99ff1799cdf9e0878c7009035afe1bccac129cad8
Deleted: sha256:
  ↳ e5599115b6a67e08278d176b05a3defb30e5564f5be6d73264ec560b484514a2
Deleted: sha256:
  ↳ a2ae92ffcd29f7ededa0320f4a4fd709a723beae9a4e681696874932db7aee2c
```

FIGURE 3.9 – Docker rmi

Chapitre 3. Bases de Docker

Je tiens à signaler que l'on peut déjà se rendre compte d'un avantage de Docker par rapport à une machine virtuelle, l'image de Debian que l'on vient de télécharger ne fait que 123MB.

À présent lançons notre premier conteneur Docker :

```
$ docker run -it debian:jessie /bin/bash  
root@47e35436f723: /#
```

Il est maintenant possible d'exécuter n'importe quelle commande à l'intérieur du conteneur :

```
$ date  
Sun Feb 5 11:01:11 UTC 2017
```

Ctrl+d permet de sortir du conteneur.

```
$ docker run -itd debian:jessie  
5c09ebfa8bc99235ad482256dfb9fa1fa470a42314ed61654b6a653aff6fee6b
```

FIGURE 3.10 – Docker run linux

En ajoutant l'option "d" le conteneur est exécuté de manière *détachée*.

En utilisant la commande suivante, il est possible de visualiser les conteneurs en cours d'exécution :

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED
↪	STATUS	PORTS	NAMES
5c09ebfa8bc9	debian:jessie	"/bin/bash"	About a
↪ minute ago	Up About a minute		
↪ admiring_brattain			

FIGURE 3.11 – Docker ps

Il est possible d'exécuter n'importe quelle commande dans un conteneur en cours d'exécution :

```
$ docker exec -it admiring_brattain date  
Sun Feb 5 11:06:21 UTC 2017
```

FIGURE 3.12 – Docker exec

Ici le conteneur est identifié par son nom, si à l'exécution aucun nom n'est défini, Docker en attribue un de manière aléatoire. Il est également possible d'identifier un conteneur en utilisant son *CONTAINER ID*.

En effet, il est possible et bien utile de définir un nom à vos conteneurs :

```
$ docker run -itd --name inphinity debian:jessie
c4be3f837adcbd5ab077e2b4a7903d2c1d4b5c434c7f877079c3ab5e22a2c555

$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED
c4be3f837adc	debian:jessie	"/bin/bash"	3 seconds
ago	Up 3 seconds		inphinity

FIGURE 3.13 – Docker run

Finalement, abordons la commande *build*. En effet, pour le moment nous n'avons fait qu'utiliser des images déjà construites. Un aspect très intéressant de Docker est de pouvoir décrire en détail la construction d'une image qui sera ensuite utilisée afin de lancer notre conteneur.

Prenons comme exemple le cas où vous souhaitez lancer un conteneur qui possède Python 3 préinstallé dans la version que l'on souhaite.

Pour se faire, il faut créer un fichier *Dockerfile* dans un dossier vide. Vous pouvez retrouver le fichier de cet exemple dans le dossier "sources" et en annexe.

1. Il faut spécifier l'image de base :

```
FROM debian:jessie
```

2. On souhaite premièrement mettre à jour les paquets :

```
RUN apt-get update && \
    apt-get upgrade -y
```

3. Finalement, on installe le paquet que l'on souhaite, python dans sa version 3 :

```
RUN apt-get install -y python3
```

À présent, on peut *build* notre image Docker, l'exécuter et voir que python 3 est bien présent :

```
docker build -t exemple_1 .

[...]
Successfully built dead33da7d24

$ docker run -it exemple_1

root@41ac57ce9c0f:/# python3
Python 3.4.2 (default, Oct 8 2014, 10:45:20)
[GCC 4.9.1] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

FIGURE 3.14 – Docker build

3.4.2 Serveur Web : Docker compose

Afin de montrer l'utilisation de *Docker Compose* nous allons réaliser un petit serveur web. Cela nous permettra également de voir comment plusieurs microservices encapsulés chacun dans un conteneur Docker, peuvent former une application.

Toutes les sources de cet exemple sont disponibles dans le dossier, *sources/exemple₂*.

Cet exemple consiste en deux conteneurs Docker, construits à partir de deux *Dockerfile*.

Premièrement, un conteneur avec mysql, qui gère la base de données :

```
FROM mysql:5.7

COPY ./my.cnf /etc/mysql/conf.d/
```

FIGURE 3.15 – Mysql Docker

Deuxièmement, un conteneur avec PHP et apache, afin d'exécuter les pages web :

```
FROM php:7-apache

COPY php.ini /usr/local/etc/php/

RUN apt-get update \
    && apt-get install -y libfontconfig-dev libjpeg62-turbo-dev libpng12-
    ↪ dev libmcrypt-dev \
    && docker-php-ext-install pdo_mysql mysqli mbstring gd iconv mcrypt
```

FIGURE 3.16 – php-apache Docker

De plus, les sources se trouvent dans ce que l'on appelle un volume, une source de données pour nos conteneurs. Dans cet exemple nous y mettons uniquement un fichier *index.php* avec la commande *phpinfo()*, afin de vérifier que notre application fonctionne bien.

Nous allons utiliser *Docker Compose*, afin de lancer notre application complète :

```

$ nano docker-compose.yml

version: '2'
services:
  mysql:
    build: ./mysql
    environment:
      MYSQL_ROOT_PASSWORD: pass
    volumes:
      - db:/var/lib/mysql
  php:
    build: ./php
    ports:
      - '8080:80'
    volumes:
      - ./html:/var/www/html
    depends_on:
      - mysql
volumes:
  db:

```

FIGURE 3.17 – docker-compose configuration

Exécutons, à présent, notre application web :

```

$ cd exemple_2/
$ docker-compose up -d

[...]

Creating exemple2_mysql_1
Creating exemple2_php_1

$ docker ps

```

CONTAINER ID	IMAGE	COMMAND	CREATED
e921dc5930ac	exemple2_php	"docker-php-entrypoint"	4
↪ minutes ago	Up 4 minutes	0.0.0.0:8080->80/tcp	
↪ exemple2_php_1			
cdb19eaa3cb7	exemple2_mysql	"docker-entrypoint.sh"	4
↪ minutes ago	Up 4 minutes	3306/tcp	
↪ exemple2_mysql_1			

FIGURE 3.18 – Docker-compose up

Pour vérifier que tout fonctionne, utilisez votre navigateur favori et accédez à l'adresse *localhost :8080* 3.19.


PHP Version 7.1.1 	
System	Linux e921dc5930ac 4.4.0-21-generic #37-Ubuntu SMP Mon Apr 18 18:33:37 UTC 2016 x86_64
Build Date	Jan 24 2017 18:33:18
Configure Command	./configure '--with-config-file-path=/usr/local/etc/php' '--with-config-file-scan-dir=/usr/local/etc/php/conf.d' '--disable-cgi' '--enable-ftp' '--enable-mbstring' '--enable-mysqlnd' '--with-curl' '--with-libedit' '--with-openssl' '--with-zlib' '--with-apxs2'
Server API	Apache 2.0 Handler
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/usr/local/etc/php
Loaded Configuration File	/usr/local/etc/php/php.ini
Scan this dir for additional .ini files	/usr/local/etc/php/conf.d
Additional .ini files parsed	/usr/local/etc/php/conf.d/docker-php-ext-gd.ini, /usr/local/etc/php/conf.d/docker-php-ext-mcrypt.ini, /usr/local/etc/php/conf.d/docker-php-ext-mysqli.ini, /usr/local/etc/php/conf.d/docker-php-ext-pdo_mysql.ini
PHP API	20160303
PHP Extension	20160303
Zend Extension	320160303
Zend Extension Build	API320160303,NTS
PHP Extension Build	API20160303,NTS
Debug Build	no
Thread Safety	disabled
Zend Signal Handling	enabled
Zend Memory Manager	enabled
Zend Multibyte Support	provided by mbstring
IPv6 Support	enabled
DTrace Support	disabled
Registered PHP Streams	https, ftps, compress.zlib, php, file, glob, data, http, ftp, phar
Registered Stream Socket Transports	tcp, udp, unix, udg, ssl, tls, tlsv1.0, tlsv1.1, tlsv1.2
Registered Stream Filters	zlib.*, convert.iconv.*, string.rot13, string.toupper, string.tolower, string.strip_tags, convert.*, consumed, dechunk, mcrypt.*, mdecrypt.*

FIGURE 3.19 – phpinfo()

3.4.3 Parallélisation

Il est possible de lancer plusieurs conteneurs avec la même image. Nous allons utiliser ceci dans la suite du développement afin de lancer plusieurs conteneurs pour l'analyse des séquences.

```
$ docker run -itd debian:jessie
```

kamyh@kamyh-linux-tower ~/projects/master/documents/sources/exemple_2 \$

```
↪ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED
↪	STATUS	PORTS	NAMES
8901929d2bf0	debian:jessie	"/bin/bash"	2 seconds
↪ ago	Up 1 seconds		
↪ determined_sinoussi			
877d2cddce6d	debian:jessie	"/bin/bash"	3 seconds
↪ ago	Up 2 seconds		boring_nobel
4ff340b9c797	debian:jessie	"/bin/bash"	4 seconds
↪ ago	Up 3 seconds		loving_mccarthy
e9a8bda7688f	debian:jessie	"/bin/bash"	4 seconds
↪ ago	Up 3 seconds		focused_dijkstra
2060301458d6	debian:jessie	"/bin/bash"	5 seconds
↪ ago	Up 5 seconds		
↪ awesome_blackwell			

FIGURE 3.20 – Docker Hmmer

Ici nous avons cinq conteneurs debian lancés simultanément. Ceci peut être très utile, par exemple, dans le cas d'une application web, afin de partager la charge des connexions utilisateurs entre plusieurs conteneurs.

4 | Parallelisation python3

Nous allons principalement nous servir de deux fonctionnalités afin de rendre une partie, critique, du code parallèle. En effet, la recherche de domaine protéinique, à l'aide de *HMMER Scan* est une opération très longue.

Nous allons rendre parallèle les appels à un conteneur Docker encapsulant une version locale de *HMMER* et la lecture des résultats.

Pour plus d'informations sur la transformation du code, que ce soit l'ajout de la philosophie objet ou au niveau des optimisations, veuillez consulter le chapitre 5-Environnement et application

4.1 Code de base

Dans ce sous-chapitre il est question de comment paralléliser un code en python 3, lorsque l'on a un traitement à appliquer sur un ensemble de données de même type.

Le code suivant montre comment utiliser la librairie *multiprocessing* et les *pool*.

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    p = Pool(processes=5)
    print(p.map(f, [1, 2, 3]))
```

Ce qui se passe dans ce code c'est que l'on *map* une fonction sur un tableau de données. Ce code va donc appliquer la fonction carrée sur chaque élément du tableau et renvoyer les résultats dans un tableau.

Il est également possible de passer à la fonction *Pool.map()* des paramètres plus complexes :

```
from multiprocessing import Pool

def f(x, y):
    return x+y

if __name__ == '__main__':
    p = Pool(processes=5)
    print(p.starmap(f, [[1,2],[3,4],[5,6]]))
```

La ligne suivante, permet de définir le nombre de coeurs CPU que l'on souhaite utiliser :

```
p = Pool(processes=5)
```

Attention la fonction *starmap* n'est disponible que depuis python 3.3.

Retrouver ces codes dans le dossier *sources/exemple₃*. Dans ce chapitre nous n'aborderons que les éléments nécessaires à la compréhension et à l'utilisation du code produit dans cette thèse. Pour plus d'informations sur la librairie multiprocessing cf. <https://docs.python.org/2/library/multiprocessing.html>.

Dans un des scripts du code de la thèse [*Modélisation prédictive des interactions entre bactéries et virus bactériophages - Leite Diogo*], des séquences protéiniques sont traitées en faisant appel à l'API de HMMER. Dans le chapitre 5-Environnement et application nous verrons comment cet appel a été remplacé par un conteneur Docker.

5 | Environnement et application

Nous allons aborder, dans ce chapitre, le nouvel environnement créé à l'aide de Docker et le nouveau code produit à partir de celui de la thèse [*Modélisation prédictive des interactions entre bactéries et virus bactériophages* - Leite Diogo] .

L'environnement est réalisé à l'aide de *Docker Compose*, il est composé de trois images différentes. Le code, orienté objet sera exécuté dans ce nouvel environnement.

5.1 Images Docker

Tous les fichiers nécessaires à la construction des images de notre environnement se trouvent dans le dossier *developpement/dockers*.

5.1.1 Hmmer

La première image est celle visant à remplacer l'utilisation de l'API en ligne de HMMER. Il s'agit d'encapsuler l'application HMMER afin de pouvoir l'utiliser pour traiter les séquences protéiniques.

Les fichiers nécessaires se trouvent dans le dossier *developpement/dockers/hmmer*. Il s'agit d'une image basée sur *centos*, qui est une image de base Docker très légère.

Tout d'abord, le *Dockerfile* de cette image installe le compilateur C++ *gcc*. Puis, il récupère les sources de l'application HMMER et les compile.

Pour davantage de détails, veuillez consulter directement le fichier *Dockerfile* de l'image.

5.1.2 Database

L'application nécessite plusieurs bases de données Mysql. L'image *database*, dont les sources se trouvent dans le dossier *developpement/dockers/database*, remplit cette fonctionnalité.

Pour des raisons de simplicité, cette image est construite à partir de Debian Jessie, la différence entre Centos et Debian est négligeable du fait que nous ne lancerons qu'un seul conteneur de cette image.

Dû à la fois à la phase de debug et pour de futurs debugs, l'image est construite avec un certain nombre de paquets afin de faciliter la vie du développeur.

La mise en place d'une image *Docker* avec un serveur mysql n'est, par expérience, jamais une chose facile à réaliser. C'est pour cela que nous allons entrer un peu plus dans les détails des différentes commandes qui composent le fichier *Dockerfile* de cette image. Il n'est pas forcément nécessaire de lire ce sous-chapitre pour comprendre les aboutissants de cette thèse, mais il est nécessaire que les informations qui suivent y figurent.

Premièrement, afin de pouvoir accéder au conteneur lancé à partir de cette image, il faut faire en sorte que mysql écoute les connexions en entrée.


```
RUN sed -i -e "s/^bind-address\s*=\s*127.0.0.1/bind-address=_0.0.0.0/" /  
    ↪ etc/mysql/my.cnf
```

Maintenant que notre conteneur peut recevoir des connexions, on installe *mysql-server* *mysql-client* *libmysqlclient-dev* qui installent Mysql sur notre image.

On peut démarrer le service Mysql :

```
RUN mysqld &  
RUN service mysql start
```

On n'oublie pas d'exposer le port de connexion que l'on souhaite. Ici le 3306 qui est le port par défaut de Mysql.

```
EXPOSE 3306
```

On modifie également quelques configurations Mysql, afin de pouvoir utiliser des fichiers *.sql* de tailles plus grandes.

```
RUN sed -ire 's/max_allowed_packet.*=.* / max_allowed_packet = 200M/g' /  
    ↪ etc/mysql/my.cnf  
RUN sed -ire 's/key_buffer_size.*=.* / key_buffer_size = 128M/g' /etc/  
    ↪ mysql/my.cnf
```

On ajoute le fichier *startup.sh* et on fixe qu'au lancement du conteneur il soit exécuté.

```
ADD ./startup.sh /opt/startup.sh  
CMD [ "/bin/bash", "/opt/startup.sh" ]
```

Regardons ce que l'on trouve dans le fichier *startup.sh* :

```

if [ ! -f /var/lib/mysql/ibdata1 ]; then

    mysql_install_db

    /usr/bin/mysqld_safe &
    sleep 10s

    echo "GRANT_ALL_ON_*.*_TO_admin@'%'_IDENTIFIED_BY_'root'_WITH_
        ↪ GRANT_OPTION;_FLUSH_PRIVILEGES" | mysql

    # For 1_F1 DetectDomains.py
    echo "CREATE_DATABASE_phage_bact" | mysql
    mysql phage_bact < /tmp/db/phagesVD.sql
    mysql phage_bact < /tmp/db/bacteriaVD.sql
    mysql phage_bact < /tmp/db/interactionsVD.sql
    mysql phage_bact < /tmp/db/neg_interactionsVD.sql
    mysql phage_bact < /tmp/db/protdom_create.sql
    mysql phage_bact < /tmp/db/progress_create.sql
    mysql phage_bact < /tmp/db/progress_interaction_create.sql

    # For 3_F1 countScoreInteraction.py
    mysql phage_bact < /tmp/db/score_interactions_create.sql

    echo "CREATE_DATABASE_domine" | mysql
    mysql domine < /tmp/db/domTGo.sql
    mysql domine < /tmp/db/domPfam.sql
    mysql domine < /tmp/db/domPgmap.sql
    mysql domine < /tmp/db/domTInteract.sql

    # For 4_F1 FreqQtdScores.py
    mysql phage_bact < /tmp/db/qtd_scores_create.sql

    killall mysqld
    sleep 10s
fi

/usr/bin/mysqld_safe

```

FIGURE 5.1 – Création bases de données

Dans ce script, on commence par donner les privilèges Mysql à l'utilisateur admin. Ensuite on met en place les différentes bases de données, *phage_bact*, *domine*, dont l'application a besoin. On crée ces deux bases de données et leurs tables.

Dans l'éventualité où l'on souhaiterait ajouter une nouvelle base de données, c'est ici qu'il faudra ajouter la commande Mysql de création.

De plus si l'on souhaite ajouter des tables ou données à une de nos bases de données, c'est également dans ce fichier qu'il faudra le faire.

Comme vous pouvez le voir, tous les fichiers *.sql* nécessaires sont placés dans le dossier *developpement/dockers/databas*

5.1.3 Core

Dû à la fois à la phase de debug et pour de futurs debugs, l'image est construite avec un certain nombre de paquets afin de faciliter la vie du développeur.

C'est véritablement cette image qui contient tout ce qui est nécessaire à l'exécution de l'aspect bio-informatique de l'application. C'est également elle qui contrôle l'application, étant donné que c'est dans ce conteneur que le code de l'application est exécuté.

Comme pour l'image *database* on installe les paquets nécessaires à utiliser Mysql, *mysql-server*, *mysql-client*, *libmysqlclient-dev* et *python*.

On installe pip3, le gestionnaire de paquets pip pour python3 :

```
RUN apt-get install -y python3-pip r-base
```

On installe aussi un des paquets les plus importants de cette image.

```
RUN pip3 install biopython
```

Afin d'accéder à la base de données depuis le code python il nous faut installer le paquet :

```
RUN pip install mysql-connector
```

Finalement, on installe Docker, car il nous faudra pouvoir communiquer avec l'engin Docker de l'hôte, afin d'exécuter des conteneurs de l'image HMMER.

```
RUN apt-get install -y curl  
RUN curl -fsSL https://get.docker.com/ | sh
```

FIGURE 5.2 – Core installations

5.2 Docker Compose

Maintenant que nous avons toutes les images nécessaires à notre environnement applicatif orienté bioinformatique, nous allons voir comment les combiner à l'aide de *Docker Compose*.

La figure 5.3 montre l'architecture qui est mise en place à l'aide de *Docker*.

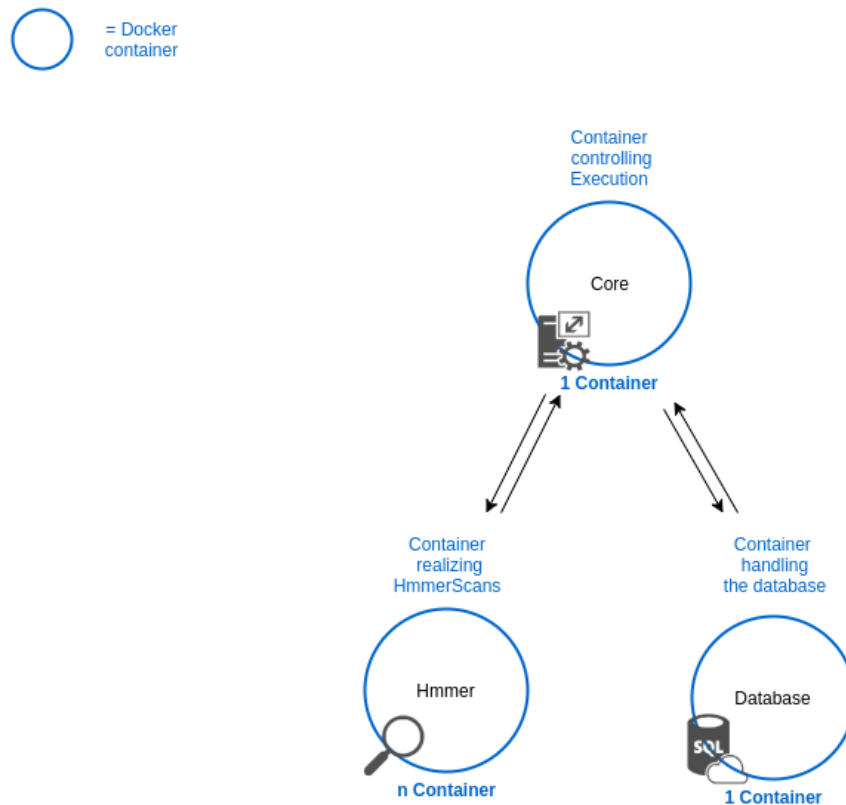


FIGURE 5.3 – Architecture de l'environnement

La figure 5.3 montre qu'un conteneur construit à partir de l'image *Core* contrôlera l'exécution du code applicatif et la création de conteneurs basés sur l'image *hmmer*. De plus, tout ce qui a trait aux données stockées en SQL sera géré par le conteneur *Database*.

De cette manière, le conteneur *Core* peut créer des conteneurs *hmmer* pour exécuter les recherches de domaines protéiniques.

Regardons comment cela est mis en place grâce à *Docker Compose*.

Premièrement, il nous faut un service qui lancera notre conteneur de base de données :

```
database:
  build: ../dockers/database
  tty: true
  environment:
    MYSQL_ROOT_PASSWORD: SecretPasswordInphinity
    MYSQL_USER: inphinity
    MYSQL_PASSWORD: SecretPasswordInphinity
    MYSQL_DATABASE: phage_bact
  ports:
    - 3309:3306
  networks:
    mynet:
      ipv4_address: 172.25.0.102
  container_name: inphinity-database
  volumes:
    - /inphinity-data/mysql:/var/lib/mysql
```

FIGURE 5.4 – Service base de données

On spécifie que l'on souhaite construire notre conteneur à partir de l'image se trouvant dans le dossier */dockers/database*. En effet, à l'exécution de la commande de lancement de notre *compose*, docker compose construira l'image *Database* et créera un conteneur à partir de cette image.

On fixe les paramètres de configuration de Mysql, tels que le mot de passe root, l'utilisateur, son mot de passe et le nom de la base de données par défaut.

On route les ports utiles au conteneur, ainsi que le réseau virtuel qui sera utilisé.

On fixe manuellement le nom du conteneur qui sera créé afin de facilement pouvoir y accéder en cas de besoin (debug, tests, etc.).

Finalement, on ajoute les volumes que l'on souhaite à notre conteneur. En effet, avec cette dernière commande, on attache le dossier hôte */inphinity-data/mysql* au dossier */var/lib/mysql* du conteneur. De cette manière lorsque l'on arrête et relance notre application notre base de données ne sera pas détruite. Si l'on souhaite supprimer la base de données et en recréer une vierge, il suffit de supprimer le contenu du dossier */inphinity-data/mysql*.

Deuxièmement, il nous faut un service qui lancera le conteneur de notre contrôleur :

```

core:
  build: ../dockers/core
  hostname: core
  networks:
    mynet:
      ipv4_address: 172.25.0.101
  tty: true
  volumes:
    - ../inphinity:/inphinity:Z
    - ../dockers/core/data-hmm:/data-hmm:Z
    - /var/run/docker.sock:/var/run/docker.sock
  privileged: true
  links:
    - database:database
  depends_on:
    - database
  container_name: inphinity-core

```

FIGURE 5.5 – Service contrôleur

On suit exactement la même logique que pour notre premier service. On spécifie que l'on souhaite construire notre conteneur à partir de l'image se trouvant dans le dossier `/dockers/core`. En effet, à l'exécution de la commande de lancement de notre *compose*, docker compose construira l'image *Core* et créera un conteneur à partir de cette image.

On attribue le réseau virtuel qui sera utilisé, le même que pour le service de la base de données.

On fixe manuellement le nom du conteneur qui sera créé, afin de facilement pouvoir y accéder en cas de besoin (debug, tests, etc.).

On ajoute les volumes nécessaires :

- `../inphinity:/inphinity`, contient le code applicatif ;
- `../dockers/core/data-hmm:/data-hmm`, servira à transmettre les fichiers de résultats des analyses réalisés par les conteneurs *hmm* au contrôleur ;
- `/var/run/docker.sock:/var/run/docker.sock`, permet au contrôleur de communiquer avec l'engin Docker de l'hôte et donc de créer les conteneurs *hmm* à la volée.

On lie, grâce à la commande *links*, le contrôleur à la base de données.

La commande *depends_on* permet de garantir que le conteneur "core" sera créé uniquement après la création du conteneur "database".

Le troisième service se trouvant dans le fichier *compose* n'est pas nécessaire, car on lancera les conteneurs *hmm* directement depuis le code python de notre application.

Finalement, on définit un réseau dans lequel on place les conteneurs de l'application.

5.3 «Inphinity Environment»

Comme déjà précisé précédemment, une première optimisation du code est passée par le fait de traduire les codes existants du python2 vers le python3.3 .

Au niveau du code, la logique suivie a été de diviser le code en *phases* successives. Chacune de ces phases est chargée d'un traitement spécifique de remplir un objectif.

Attention, ce chapitre n'aborde pas la logique du code réalisé dans la thèse [*Modélisation prédictive des interactions entre bactéries et virus bactériophages - Leite Diogo*] , il n'est question ici que de la plus-value ajoutée lors de ce travail de master.

Les fonctions utilisées dans les scripts de la thèse [*Modélisation prédictive des interactions entre bactéries et virus bactériophages - Leite Diogo*] restent très proches de celle de ce travail !

5.3.1 Diagramme de classes

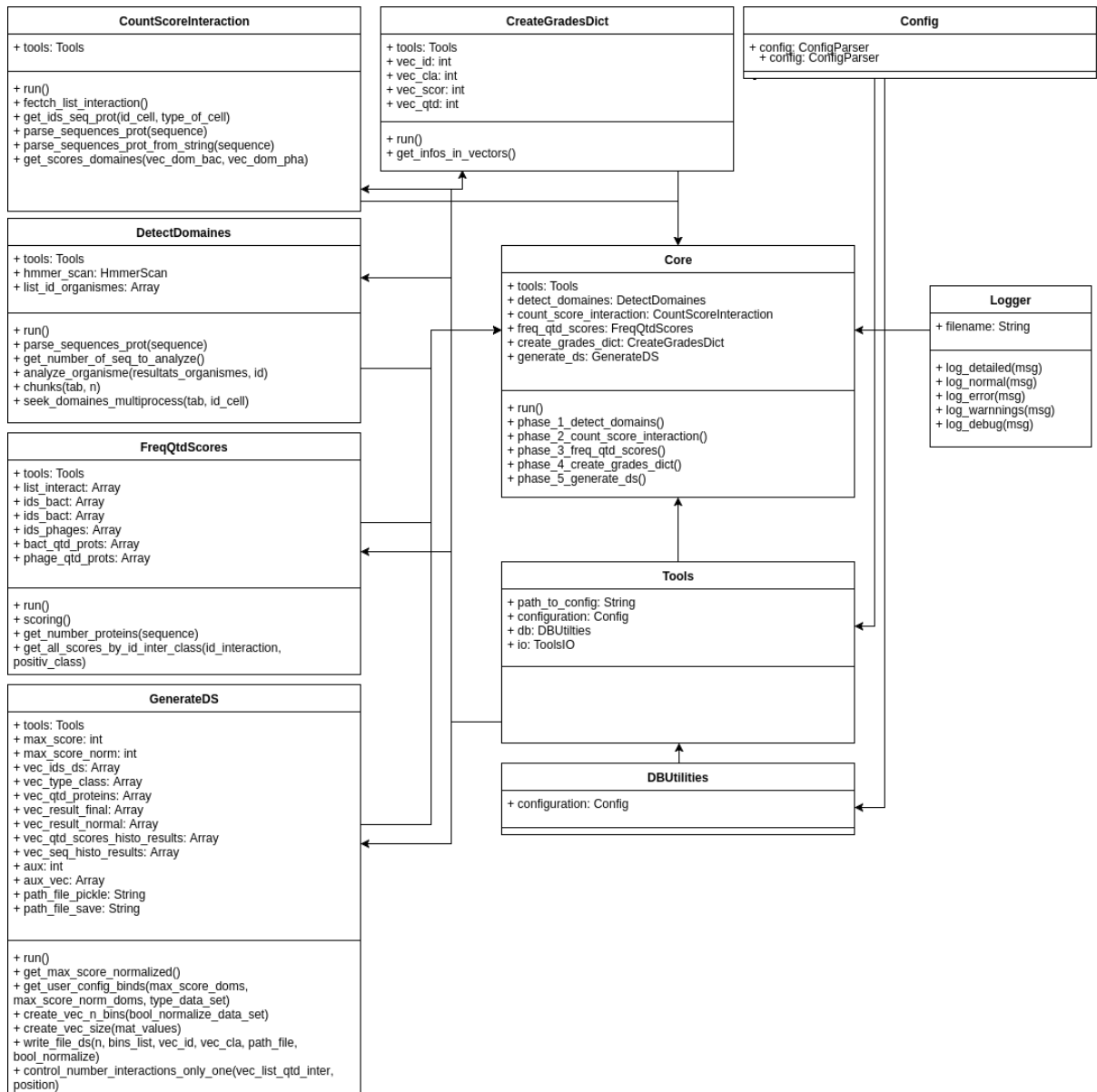


FIGURE 5.6 – Diagramme de classes

Core

La classe *Core* est la classe principale de l'application. C'est elle qui est chargée d'instancier les classes nécessaires au bon déroulement du code et de lancer les phases du processus.

Comme on le voit sur la figure 5.6, la classe *Core* possède une instance des cinq classes, représentant chacune une phase du processus. Elle possède également une instance de la classe *Tools*, qui donne accès à l'objet de la classe *Config*, gérant le *parsing* de la configuration en cours. De plus, la classe *Tools* possède un accès à l'objet *DBUtilities*, qui gère l'ensemble des accès aux bases de données.

Phase 1 - DetectDomaines

C'est durant cette phase que l'utilisation à HMMER est faite, c'est dans cette phase que la parallélisation de cette utilisation est réalisée. Plus précisément regardons la fonction *seek_domains_multiprocess()*, on y initialise une *Multiprocessing.Pool* avec un certain nombre de coeurs à disposition. Ce nombre de coeurs est défini dans les fichiers de configuration, cf. 5.3.2.

Ensuite, on divise le tableau des séquences à traiter en *chunk*, cela permet d'obtenir les résultats également par séries de 'n' réponse et de les incérer dans la base de données au fur et à mesure. Cela afin d'éviter de perdre du temps d'analyse en cas d'arrêt de l'application.

```
chunk_size = pool_size * self.tools.configuration.  
    ↪ get_chunk_size_multiplier()  
    chunks = self.chunks(list(tab), chunk_size)
```

FIGURE 5.7 – Processus pool

Lorsque l'on insère les résultats dans la base de données on insère également dans la table *progress* quelles séquences ont été analysées. Cela nous permet d'arrêter et de relancer l'application quand on veut sans pour autant devoir recommencer la phase d'analyse à zéro.

On exécute le code suivant pour chacun des *chunks* :

```
for chunk in chunks:  
    print('%d sequences processed!' % total_processed)  
    LOGGER.log_normal('%d sequences processed!' %  
        ↪ total_processed)  
    total_processed += chunk_size  
    results = p.map(self.hmmer_scan.analyze_domains, chunk)  
    print(results)  
  
    for prot in chunk:  
        self.tools.db.analyze_done(prot[0], 1)  
  
    for result in results:  
        try:  
            id_prot = result[0]  
            domaines_returned = result[1]  
  
            self.tools.db.execute_insert_domains(id_prot,  
                ↪ domaines_returned, id_cell, bool_bacteria, "—  
                ↪ ")
```

FIGURE 5.8 – Division en Chunks

On fait appel de manière parallèle à la fonction *hmmerScan.analyze_domains()*. Grâce à la *pool* de processus la fonction sera exécutée simultanément un certain nombre de fois.

Allons voir plus en détails ce qui se passe dans cette fonction. Pour voir le code de la fonction en détails, veuillez vous rendre directement dans les sources du code (*developpement/inphinity/v0.5/hmmerScan.py*).

La séquence envoyée à la fonction est *parsée* dans un fichier temporaire :

```
fasta = '>' + values_tab[0] + '\n' + values_tab[1] + '\n'
fasta_filename = '/data-hmm/tmp/' + str(uuid.uuid4()) + '.fasta'
self.io.write(fasta_filename, fasta)
```

On crée également un fichier temporaire où sera stocké le résultat de l'analyse par *HMMER scan* :

```
results_filename = "/data-hmm/results/hits_test_" + str(uuid.uuid4()) +
    ↪ ".txt"
```

Par la suite le code fait appel à la fonction *Subprocess.Popen()* permettant de réaliser de commande UNIX directement depuis un code Python. Grâce à cela on peut lancer une commande qui créera un nouveau conteneur Docker qui effectuera une analyse de séquence, une fois l'analyse effectuée le conteneur s'arrête automatiquement.

```
p = subprocess.Popen([
    "docker_" +
    "run_" +
    "--rm_" +
    "--privileged_" +
    "-v_" +
    path_to_core + "/data-hmm:/data-hmm_" +
    "inphinity-hmmer_" +
    "hmmsearch_" +
    "--tblout_" +
    results_filename + "_" +
    "/data-hmm/Pfam-A.hmm_" +
    fasta_filename
], stdout=subprocess.PIPE, shell=True)

(output, err) = p.communicate()

# This makes the wait possible
p_status = p.wait()
```

FIGURE 5.9 – Popen Docker Hmmer

Cette commande lance un conteneur docker avec l'image que l'on a préalablement créée pour encapsuler HMMER. On spécifie le fichier dans lequel se trouve la séquence au format *FASTA*, ainsi que le fichier dans lequel on souhaite inscrire les résultats de l'analyse.

Ensuite, on lit, récupère et filtre les résultats :

```
results = self.io.read_results(results_filename, self.configuration.
    ↪ get_detailed_logs())

[...]
```

Nous aborderons la sélection des résultats dans la section 5.3.2.

Finalement on supprime les deux fichiers temporaires et retourne les résultats.

```
p = subprocess.Popen([
    "rm",
    results_filename
])
p.communicate()
p_status = p.wait()

p = subprocess.Popen([
    "rm",
    fasta_filename
])
p.communicate()
p_status = p.wait()

return [values_tab[0], returned_domains, [start_time, end_time]]
```

FIGURE 5.10 – Popen

5.3.2 Configuration

Toutes les interactions utilisateurs que les scripts produisent lors de la thèse [*Modélisation prédictive des interactions entre bactéries et virus bactériophages - Leite Diogo*] ont été remplacées par une valeur dans le fichier de configuration.

Lancement des configurations

La classe *Main* de l'application (app.py) est faite pour lancer la fonction *Core.run()* pour chaque fichier de configuration se trouvant dans le dossier *inphinity/v_{0.5}/configs*. Il faut préciser que seuls les fichiers finissant par l'extension *.ini* sont considérés comme des fichiers de configuration. Donc pour garder une configuration, tout en empêchant qu'elle soit utilisée, il suffit d'ajouter *.old* à la fin, par exemple.

```
if __name__ == '__main__':
    os.chdir("inphinity/v_0.5/configs")
    configs = glob.glob("*.ini")

    for config_file in configs:
        print('Config file: %s' % config_file)

        c = Core(config_file)
        c.run()
```

FIGURE 5.11 – Main

Voici un fichier de configuration standard :

```

[INFORMATION]
verbose = 0
detailed_logs = 0
testing = 0
[ENV]
path_to_core = /home/kamyh/projects/master/developpement/dockers/core
#Dile to parse sequences
temp_file_pseqs = /tmp/tmpMF.txt
reset_db_at_start = 0
#if = 0 —> auto
#if = -1 —> nbr of core
#if = x —> x
process = -1
# Smaller the chunk are, the more time is lost
chunk_size_multiplier = 4
phases_to_run = 1,2,3,4,5
[PHASE_1]
analyze_phage = 1
analyze_bacteria = 1
[DOMS_SELECTION]
use_e_value_selection = 0
min_e_value = 9.8e-25
max_e_value = 1500

use_score_selection = 0
min_score = 0
max_score = 1500

use_biais_selection = 0
min_biais = 0
max_biais = 1500
[DATASET_GENERATION]
grades_file_pseqs = /inphinity/grades/gradesDict.p
ds_dir = /inphinity/datasets/
normalisation = 0
# 1 —> Number of bins
# 2 —> Vector of bins
type_bins = 1
number_of_bins = 1
space_between_bins = 1

```

FIGURE 5.12 – Fichier de configuration

Étant donné l'importance des fichiers de configuration dans la production de dataset intéressants, nous allons passer en revue les différents éléments y étant définis.

Premièrement, il faut savoir que l'on trouve deux types d'éléments dans un fichier de configuration, des sections et de valeurs. Les sections sont des mots clés entre crochets, eg. [INFORMATION], et en majuscules. Les valeurs sont en minuscules, et se rapportent chacune à une section. En effet, toutes les valeurs après une section font partie de cette section. Cela permet d'organiser le fichier de configuration au mieux.

Section - [INFORMATION]

Ces valeurs de configuration servent lors des phases de développement et tests.

- *verbose*, permet de désactiver la plupart des affichages console.
- *detailed_logs*, permet d'inscrire dans les logs des informations détaillées sur l'exécution de l'application.
- *testing*, permet de limiter le nombre d'organismes traités par l'application, à des fins de développement ou de tests.

Section - [ENV]

Les configurations se trouvant sous cette section concernent des variables d'environnement de l'application.

- *path_to_core*, afin de fonctionner, l'application a besoin de connaître l'emplacement du dossier contenant la description Docker de l'image du contrôleur principal.
- *temp_file_pseqs*, permet de spécifier le fichier temporaire qui sera utilisé afin de parser les séquences multi-fasta.
- *reset_db_at_start*, permet de réinitialiser la base de données au lancement de l'application.
- *process*, spécifie le nombre de coeurs qui sera alloué à la *pool*.
 - fixé à "0", laisse la librairie automatiquement définir le nombre de coeurs, normalement le nombre maximum disponible.
 - fixé à "-1", fixe le nombre de coeurs au maximum disponible moins 1. Cette méthode permet de laisser un coeur libre pour les processus de la machine hôte.
 - fixé à "x", laisse choisir le nombre de coeurs, mais attention, en mettre davantage que le maximum ralentit l'exécution de l'application.
- *chunk_size_multiplier*, définit la taille de *chunk* pour le traitement des séquences lors de la phase 1.
- *phases_to_run*, permet de définir quelles phases on souhaite exécuter. Par exemple, si la valeur est égale à "1,2,3,4,5", toutes les phases seront exécutées. On peut choisir de n'exécuter que les deux dernières phases en inscrivant la valeur de ce paramètre à "4,5".

Section - [PHASE₁]

Configurations se rapportant directement à l'exécution de la phase 1 du traitement.

- *analyze_phage*, permet de définir si les phages doivent être traités.
- *analyze_bacteria*, permet de définir si les bactéries doivent être traitées.

Section - [DOMS_SELECTION]

Une analyse de séquence par *hmmsearch* donne des résultats semblables à la figure 5.13. On voit que chaque domaine trouvé possède une valeur de score, une e-value et un biais.

```
#
# target name      accession  query name      accession  --- full sequence ---  --- best 1 domain ---  --- domain number estimation ---
#               E-value  score  bias  E-value  score  bias  exp reg clu  ov  env  dom  rep  inc  description of target
#
lcl|CP014196.1_prot_AUT26_00005_1 - AAA PF00004.27 3.7e-12 33.1 0.0 4.6e-11 29.5 0.0 2.7 3 0 0 3 3 3 1 -
lcl|CP014196.1_prot_AUT26_00005_1 - AAA_16 PF13191.4 9.5e-08 18.7 0.1 7e-07 15.9 0.0 2.5 2 1 0 2 2 2 1 -
lcl|CP014196.1_prot_AUT26_00005_1 - AAA_21 PF13304.4 1.6e-06 14.2 0.1 7.7e-06 11.9 0.0 2.0 2 0 0 2 2 2 1 -
lcl|CP014196.1_prot_AUT26_00005_1 - AAA_22 PF13401.4 7.1e-08 19.0 0.9 0.00011 8.7 0.0 2.9 2 1 1 3 3 3 2 -
lcl|CP014196.1_prot_AUT26_00005_1 - AAA_24 PF13479.4 3.9e-06 12.9 0.3 3.5e-05 9.8 0.1 2.3 1 1 1 2 2 2 1 -
lcl|CP014196.1_prot_AUT26_00005_1 - ATPase_2 PF01637.16 6.1e-08 18.9 0.0 1.3e-07 17.9 0.0 1.5 1 0 0 1 1 1 1 -
lcl|CP014196.1_prot_AUT26_00005_1 - Bac_DnaA PF00308.16 3.6e-102 326.9 0.0 7.9e-102 325.8 0.0 1.6 2 0 0 2 2 2 1 -
lcl|CP014196.1_prot_AUT26_00005_1 - Bac_DnaA_C PF08299.9 3.9e-35 106.1 0.1 7.6e-35 105.1 0.1 1.5 1 0 0 1 1 1 1 -
lcl|CP014196.1_prot_AUT26_00005_1 - IstB_IS21 PF01695.15 5.9e-18 51.4 0.0 1.2e-16 47.1 0.0 2.2 2 0 0 2 2 2 2 -
lcl|CP014196.1_prot_AUT26_00005_1 - NACHT PF05729.10 3.6e-06 13.2 0.1 1.6e-05 11.1 0.0 2.1 2 1 0 2 2 2 1 -
lcl|CP014196.1_prot_AUT26_00005_1 - RNA_helicase PF00910.20 1.6e-06 14.8 0.0 4e-06 13.5 0.0 1.7 1 0 0 1 1 1 1 -
lcl|CP014196.1_prot_AUT26_00005_1 - TnIB PF05621.9 1.6e-06 13.6 0.0 4.7e-05 8.8 0.0 2.4 1 1 1 2 2 2 2 -
#
# Program:      hmmsearch
# Version:      3.1b2 (February 2015)
# Pipeline mode: SEARCH
# Query file:   /data-hmm/Pfam-A.hmm
# Target file:  /data-hmm/fasta/seq_diogo_03102016.fasta
# Option settings:  hmmsearch --tblout /data-hmm/results/hits.txt /data-hmm/Pfam-A.hmm /data-hmm/fasta/seq_diogo_03102016.fasta
# Current dir:  /usr/local/bin
# Date:        Mon Feb 6 10:56:36 2017
# [ok]
```

FIGURE 5.13 – Résultats de hmmersearch

- *use_e_value_selection*, *min_e_value*, *max_e_value*, ces trois valeurs permettent de filtrer la sélection des résultats de l'analyse HMMER en utilisant la *e-value*.
- *use_score_selection*, *min_score*, *max_score*, ces trois valeurs permettent de filtrer la sélection des résultats de l'analyse HMMER en utilisant le *score*.
- *use_biais_selection*, *min_biais*, *max_biais*, ces trois valeurs permettent de filtrer la sélection des résultats de l'analyse HMMER en utilisant la valeur de *biais*.

Section - [DATASET_GENERATION]

- *grades_file_pseqs*, emplacements du dictionnaire des scores pour la création du dataset.
- *ds_dir*, emplacement du dossier dans lequel placer le dataset créé. Le nom du dataset est défini en ajoutant au nom de la configuration la date et l'heure.
- *normalisation*, spécifie si les valeurs du dataset doivent être normalisées.
- *type_bins*, sélectionne quel type de *bins* contient le dataset produit.
 - fixé à "1", signifie que l'on souhaite spécifier le nombre de *bins* à produire.
 - fixé à "2", signifie que l'on souhaite un vecteur de *bins*.
- *number_of_bins*, permet de définir le nombre de *bins*.
- *space_between_bins*, permet de définir l'espacement entre les *bins*.

5.3.3 Loggs

Afin de faciliter le développement de ce travail et les futurs développements et modifications de l'application, une classe permettant la génération d'un fichier de *log* (compte-rendu) a été réalisée. Comme on le voit sur la figure 5.6, il y a cinq niveaux de *log*. Cela permet d'organiser les informations que l'on souhaite enregistrer.

Chacun des niveaux de *log* possède un mot clé qui est placé au début de la ligne du message dans le fichier compte-rendu.

```
DETAILS 2017-01-22 12:35:18| <<very detailed message!>>
DETAILS 2017-01-22 12:35:34| <<very detailed message!>>
DETAILS 2017-01-22 12:35:56| <<very detailed message!>>
ERROR 2017-01-22 12:36:18| <<error message!>>
DEBUG 2017-01-22 12:37:18| <<debug message!>>
DEBUG 2017-01-22 12:37:19| <<debug message!>>
DEBUG 2017-01-22 12:37:20| <<debug message!>>
DEBUG 2017-01-22 12:37:21| <<debug message!>>
DEBUG 2017-01-22 12:37:22| <<debug message!>>
DEBUG 2017-01-22 12:37:23| <<debug message!>>
```

FIGURE 5.14 – Logger

Grâce au mot clé en début de ligne, il est très facile de consulter les *logs* au niveau de l'on souhaite. La commande suivante permet d'obtenir les 1000 dernières lignes contenant le mot "NORMAL" :

```
$ tail -1000 /tmp/logs_inphinity.txt |grep NORMAL
```

6 | Déploiement et execution

Dans ce chapitre, nous allons aborder la manière dont l'environnement et l'application doivent être déployés.

6.1 Obtention des sources

Le projet réalisé durant cette thèse a été développé en utilisant les gestionnaires de code source *git* par l'intermédiaire de la plateforme *Github.com*. Le projet est en accès public à l'adresse, <https://github.com/kamyh/master.git>.

```
$ git clone https://github.com/kamyh/master.git
$ cd master/developpement/dockers/core/data-hmm/
$ sh get_pfam_hmm.sh
$ cd ../../database/data/
$ wget https://www.dropbox.com/s/mzt9pxpfvxl3wa/bacteriaVD.sql?dl=0
$ mv bacteriaVD.sql?dl=0 bacteriaVD.sql
```

FIGURE 6.1 – Obtention des sources

Le code 6.1 permet d'obtenir les sources, mais également la base de données *Pfam* et la base de données des bactéries.

6.2 Edition de la configuration

Il faut tout d'abord créer au moins un fichier de configuration en copiant le fichier de configuration par défaut.

```
$ cd ../../../../inphinity/v_0.5
$ cp configs/config.ini.example config.ini
$ sudo nano configs/config.ini
```

FIGURE 6.2 – Edition de la configuration

Il est impératif de modifier la valeur de *path_to_core* afin qu'elle soit correcte par rapport à la machine hôte.

6.3 Execution de la composition Docker

Il est à présent possible d'exécuter l'environnement de notre application.


```
$ cd ../../compose/
$ sudo sh run.sh
$ docker ps
```

FIGURE 6.3 – Execution de la composition Docker

Une fois les commandes 6.3 exécutées, *docker ps* devrait retourner ceci :

CONTAINER ID	IMAGE	COMMAND	CREATED
↪	STATUS	PORTS	
↪	NAMES		
6817333ab720	compose_core	"/bin/bash"	11
↪ seconds ago	Up 7 seconds		
↪		inphinity-core	
b494735f1440	compose_database	"tini_/bin/bash/o"	12
↪ seconds ago	Up 10 seconds	3309/tcp , 0.0.0.0:3309->3306/	
↪ tcp	inphinity-database		

FIGURE 6.4 – Execution de la composition Docker vérification

L'application n'est pas exécutée automatiquement pour le moment, mais en cas de déploiement en production, cela est facile à modifier.

Pour le moment, il faut se connecter au conteneur du contrôleur et exécuter l'application :

```
$ docker exec -it inphinity-core /bin/bash
$ python3 inphinity/v_0.5/app.py
```

FIGURE 6.5 – Execution de l'application

7 | Simplification d'usage

Le projet étant relativement complexe, il est important de connaître certaines astuces permettant de simplifier son utilisation.

7.1 Commandes et alias

La commande de la figure 7.1, permet de se connecter à un conteneur, par exemple, celui des bases de données.

```
$ docker exec -it <<CONTAINER NAME/ID>> /bin/bash
$ docker exec -it inphinity-core /bin/bash
$ docker exec -it inphinity-database /bin/bash
```

FIGURE 7.1 – Commande docker exec

Le première alias de la figure 7.2, permet d'afficher des informations sur l'état du conteneur de base de données.

```
alias inphinitydbstate="docker_exec-it_inphinity-database_mysql-h_
↳ inphinity-database_u_admin-proot-e \"use phage_bact; SELECT (
↳ SELECT count(ProtDomId) as doms_found from phage_bact.PROTDOM) as
↳ doms_found, (SELECT count(id) as doms_done from phage_bact.
↳ progress) as doms_done, (SELECT count(Score_Inter_Id) from
↳ phage_bact.Score_interactions) as interactions_found;\"

alias inphinitylogs="docker_exec-it_inphinity-core_tail-100_/tmp/
↳ logs_inphinity.txt;_echo_'"
alias inphinitylogsnormal="docker_exec-it_inphinity-core_tail-1000_/
↳ tmp/logs_inphinity.txt_|grep_NORMAL;_echo_'"
```

FIGURE 7.2 – Alias

Les deux autres alias de la figure 7.2, permettent d'afficher certaines informations des *logs*, directement depuis la machine hôte.

7.2 Scripts

Le dossier *developpement/scripts/*, contient des scripts utiles à l'utilisation de ce projet.

- *install_docker.sh*, permet d'installer *Docker* et *Docker Compose* automatiquement sur n'importe quelle machine Linux.

- *deploy.sh*, permet de déployer l'environnement de l'application, de récupérer les codes source, de lancer l'environnement et de se connecter au conteneur Docker du contrôleur.
- *docker_ps_lookup.sh*, est simplement un script infini, permettant de surveiller l'état de conteneurs *Docker* en cours d'exécution.
- *infos_lookup.sh*, est simplement un script infini, permettant de surveiller l'état de l'exécution de l'application.

8 | Améliorations

Dans ce chapitre se trouve une liste non exhaustive de pistes d'amélioration pouvant encore être investiguées.

8.1 Parallélisation

La transformation d'un code séquentiel en code linéaire est intéressante dès le moment où l'on a à appliquer le même traitement à un certain nombre de données de même type. Dans le cas de ce travail, c'est la recherche des domaines avec *hmmscan* que l'on a pu rendre parallèle.

Rapidement on peut noter un certain nombre d'autres sections du code présentant le potentiel d'être rendues parallèles.

1. Dans la phase 2 - *Count Score Interaction* - il serait intéressant de rendre parallèle le traitement des interactions. Soit au niveau de la boucle sur les interactions (`core.py`, ligne 220). Soit au niveau du calcul des scores PPI (`core.py`, ligne 242).
2. Dans la phase 3 - *Freq Qtd Scores* - il serait intéressant de paralléliser le calcul du score d'interaction (`core.py`, ligne 410).

8.2 Machines Amazone

Pour le moment, le déploiement et l'utilisation de l'application ont été testés sur des machines hôtes avec un maximum de 8 coeurs disponibles. Mais également avec une machine Amazone EC2, <https://aws.amazon.com/fr/ec2/instance-types>, *t2.large*. Cela signifie que si l'on souhaite améliorer davantage la rapidité d'exécution du code, on peut déployer l'application sur une machine Amazone avec un grand nombre de coeurs telle qu'une *m4.16xlarge*.

9 | Conclusion

Ce projet propose une solution qui prend le parti d'utiliser la plateforme *Docker*, qui offre beaucoup d'avantages. Notamment en termes de déploiement sur différentes machines hôtes (différents développeurs/utilisateurs). De plus, même si pour le moment l'application n'est que dans une optique de démonstration, le fait de développer en utilisant *Docker* permet de passer très rapidement en phase de production. Il faut également noter que *Docker* est souvent utilisé pour faire du déploiement continu.

Le code, précédemment (thèse [*Modélisation prédictive des interactions entre bactéries et virus bactériophages - Leite Diogo*]) développé sous forme de scripts qui nécessitaient de nombreuses entrées utilisateurs, est maintenant complètement automatique. De plus, l'application peut être exécutée plusieurs fois, en fournissant autant de fichiers de configuration que l'on souhaite. Chacune de ces exécutions est susceptible de produire un *dataset* spécifique. Mais on peut aussi n'exécuter que certaines phases du traitement ([*Modélisation prédictive des interactions entre bactéries et virus bactériophages - Leite Diogo*]).

Travailler sur un code déjà existant est une expérience très intéressante, même si cela présente également des désavantages. Un avantage important est le fait d'avoir la liberté de se concentrer sur certains aspects uniquement et de cette manière de les approfondir davantage. C'est ce qui a été fait durant ce travail avec l'idée de parallélisation et l'utilisation de la plateforme *Docker*.

A | Annexes

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.
